References

[1] E. F. Assmus, Jr., and V. S. Pless, "On the covering radius of extremal self-dual codes," *IEEE Trans. Inform. Theory*, vol. IT-29, pp. 359–363, May 1983.

[2] R. A. Brualdi and V. S. Pless, "Orphans of the first order Reed-Muller codes," *IEEE Trans. Inform. Theory*, vol. 36, pp. 399–401, Mar. 1990.

[3] ——, "Weight enumerators of self-dual codes," *IEEE Trans. Inform. Theory*, vol. 37, pp. 1222–1225, July 1991.

[4] J. H. Conway and N. J. A. Sloane, "A new upper bound on the minimal distance of self-dual codes," *IEEE Trans. Inform. Theory*, vol. 36, pp. 1319–1333, Nov. 1990.

[5] H. P. Tsai, "Existence of certain extremal self-dual codes," *IEEE Trans. Inform. Theory*, vol. 38, pp. 501–504, Mar. 1992.

[6] ——, "Existence of some extremal self-dual codes," *IEEE Trans. Inform. Theory*, vol. 38, pp. 1829–1833, Nov. 1992.

Cascading Methods for Runlength-Limited Arrays

Tuvi Etzion, *Member, IEEE*

***Abstract*— Runlength-limited sequences and arrays have found applications in magnetic and optical recording. While the constrained sequences are well studied, little is known about constrained arrays. In this correspondence we consider the question of how to cascade two arrays with the same runlength constraints horizontally and vertically, in such a way that the runlength constraints will not be violated. We consider binary arrays in which the shortest run of a symbol in a row (column) is $d_1(d_2)$ and the longest run of a symbol in a row (column) is $k_1(k_2)$. We present three methods to cascade such arrays. If $k_1 > 4d_1 - 2$ our method is optimal, and if $k_1 \geq d_1 + 1$ we give a method which has a certain optimal structure. Finally, we show how cascading can be applied to obtain runlength-limited error-correcting array codes.**

***Index Terms*— Cascading, merging arrays, runlength-limited arrays, runlength-limited sequences.**

I. Introduction

Runlength-limited (RLL) codes are binary codes whose minimum and maximum runlengths of consecutive zeroes or ones in its codewords are constrained. Such codes have found applications in magnetic and optical recording, partial response channels, line coding, and bar codes [6], [7], [11]. The one-dimensional case of RLL sequences is well studied, while the two-dimensional case, which has horizontal and vertical constraints, has received attention from only a few authors such as Orcutt and Marcellin [9], [10] who studied multitrack or stacked RLL codes. Two-dimensional RLL codes were considered by Etzion and Wei [5]. These arrays will also be considered in this correspondence. We will study one of the fundamental questions about RLL arrays: how to cascade two constrained arrays in such a way that the constraints of the runlength will not be violated. This question is important in studying encoding, decoding, and error correction of RLL arrays, and in studying the capacity rate of the corresponding channels.

The problem of cascading RLL sequences has been studied by Tang and Bahl [12], Beenker and Immink [2], and Weber and Abdel-Ghaffer [14]. But the problems in cascading RLL arrays are more involved than the ones in cascading RLL sequences. The reason is that we have constraints in both directions, horizontally and vertically, and these constraints have some dependency.

Manuscript received September 29, 1995; revised June 21, 1996. This work was supported in part by the fund of the promotion of sponsored research. Part of this work was performed while the author was visiting Bellocore, Morristown, NJ.

The author is with the Computer Science Department, Technion–Israel Institute of Technology, Haifa 32000, Israel.

Publisher Item Identifier S 0018-9448(97)00180-6.

All sequences and arrays in this correspondence are binary. There are many types of constraints found in the literature and applications [6]. The most popular ones are the (d, k) constraints, which are sets of binary sequences in which any runlength of consecutive zeroes is between d and k, inclusive. In this correspondence we consider a more general class of runlength-limited sequences. We adopt the following notation: a (d_1, k_1, d_2, k_2) sequence is a sequence in which the length of the shortest run of consecutive zeroes (ones) is at least d_1 (d_2), and the length of the longest run of consecutive zeroes (ones) is at most k_1 (k_2). If $d_1 = d_2$ and $k_1 = k_2$ then it is called a (d_1, k_1) sequence. We make the natural assumption that $1 \leq d_i \leq k_i$, for $i = 1, 2$. In some literature, a (d, k) code refers to a set of sequences whose runlengths of consecutive zeroes are between d and k inclusively. It is easy to verify that this is equivalent to specifying that the runlengths, whether the run consists of zeroes or of ones, are between $d + 1$ and $k + 1$, inclusively. Therefore, a (d, k) code is equivalent to a set of $(d + 1, k + 1, d + 1, k + 1)$ sequences. A $(d_1, k_1, d_2, k_2; d_3, k_3, d_4, k_4)$ array of size $n_1 \times n_2$ is a binary array with n_1 rows and n_2 columns such that every row is a (d_1, k_1, d_2, k_2) sequence and every column is a (d_3, k_3, d_4, k_4) sequence. If the horizontal runlength constraints are the same as the vertical runlength constraints, i.e., $d_1 = d_3, d_2 = d_4,\ k_1 = k_3, k_2 = k_4$, it is called a (d_1, k_1, d_2, k_2) array. If, furthermore, the runlength constraints on the zeroes and ones are the same in each dimension, i.e., $d_1 = d_2$ and $k_1 = k_2$, then it is called a (d_1, k_1) array. If only the runlength constraints on the zeroes and the ones are the same, i.e., $d_1 = d_2, k_1 = k_2,\ d_3 = d_4$, and $k_3 = k_4$, then it is called a $(d_1, k_1; d_3, k_3)$ array.

Definition 1: Assume we are given an $n_1 \times n_2$ $(d_1,\ k_1,\ d_2,\ k_2;\ d_3,\ k_3,\ d_4,\ k_4)$ array X and an $n_1 \times n_3$ $(d_1,\ k_1,\ d_2,\ k_2;\ d_3,\ k_3,\ d_4,\ k_4)$ array Y. An $n_1 \times n_4$ array Z is called a *merging array* if XZY is a $(d_1, k_1, d_2, k_2; d_3, k_3, d_4, k_4)$ array.

In this correspondence we consider the following two questions.

(Q1) Given an $n_1 \times n_2$ $(d_1, k_1, d_2, k_2; d_3, k_3, d_4, k_4)$ array X and an $n_1 \times n_3$ $(d_1, k_1, d_2, k_2; d_3, k_3, d_4, k_4)$ array Y, does there exists an $n_1 \times n_4$ merging array Z such that XZY is a $(d_1, k_1, d_2, k_2; d_3, k_3, d_4, k_4)$ array?

(Q2) If the answer to (Q1) is yes, we ask how can we find such Z, and what is the narrowest merging array?

(Q1) and (Q2) are questions on the horizontal cascading. We have similar questions and answers on the vertical cascading. Without loss of generality we will only consider the horizontal cascading. The rest of this correspondence is devoted for answering these questions for certain constraints. In Section II we will give the main results on cascading constrained arrays, i.e., we will give some answers to (Q1) and (Q2). In Section III we will give some applications of cascading constrained arrays. The conclusion is given in Section IV.

II. Cascading Constrained Arrays

In this section we will show how to generate merging arrays in order to cascade constrained arrays, without violating the constraints. We will always assume that the vertical size of the arrays in this section is n_1.

Definition 2: Given a $(d_1, k_1, d_2, k_2; d_3, k_3, d_4, k_4)$ array X, a column that can be cascaded to the right of X without violating the vertical and the horizontal constraints, with a possible exception of a run shorter than d_1 or d_2 as the rightmost run, is called a *merging column*.

The answer for (Q1) is not always positive. Assume we have the constraint $(1, 2, 1, 2; d_3, k_3, d_4, k_4), k_3 < k_4$, and c is a column which starts with k_3 zeroes followed by k_4 ones, and let $X = cc$. By the horizontal constraint the next (merging) column must start with k_3 ones followed by k_4 zeroes, but this is impossible by the vertical constraint on the zeroes. This is one of the reasons that we will consider in this section only $(d_1, k_1; d_2, k_2)$ arrays. We will show that in this case the answer to our two questions is positive. If $d_1 = k_1$ or $d_2 = k_2$ the solution is either trivial or can be transferred to the one-dimensional case. Henceforth, we will assume $k_1 > d_1$ and $k_2 > d_2$.

We will make the assumption that all the arrays in this section have width at least d_1, unless otherwise stated. Also, we will denote arrays by upper case letters and columns by lower case letters.

Definition 3: An *R run (L run)* in any row of an array is the rightmost (leftmost) run in the row.

Definition 4: An array X is called a *valid* $(d_1, k_1; d_2, k_2)$ array if it satisfies the $(d_1, k_1; d_2, k_2)$ constraint, with the possible exception of R runs or L runs smaller than d_1.

Without loss of generality we can consider in (Q1) arrays X and Y which are valid arrays.

Definition 5: A valid $(d_1, k_1; d_2, k_2)$ array X is called R (L) d_1 *balanced* if the last (first) d_1 columns of X are equal and the column before (after) these d_1 columns is the complement of each of these d_1 columns.

The importance of R (L) d_1 balanced arrays comes from the observation that if we have an $n_1 \times n_2$ R (L) d_1 balanced array X then for any (d_2, k_2) RLL sequence c of length n_1, as a column vector, Xc (cX) is a valid $(d_1, k_1; d_2, k_2)$ array. For a binary value b, let $\bar{b}$ denote the binary *complement* of b. For a column c, let $\bar{c}$ denote the column obtained by complementing all the entries of c. For a column c, let c^t denote t consecutive copies of c. For an array X, let X^R denote the *reverse* of X, i.e., the columns of X taken from the last to the first.

Definition 6: For a valid $(d_1, k_1; d_2, k_2)$ array Xc, where c is the last column, the *merge one operator* results in a column $\hat{m}$, which is defined as the complement of the entry in c in all rows where Xc has R runs of length greater than or equal to d_1, and the same value as in c in all rows where Xc has R runs less than d_1.

Definition 7: We define $X[1(\hat{m})] = X\hat{m}$ and if $X[r(\hat{m})] = XY$ then $X[(r+1)(\hat{m})] = XY\hat{m}$, i.e., $[(r+1)(\hat{m})]$ is $r+1$ consecutive applications of the merge one operator.

It is important to understand that $\hat{m}$ is dependent in the d_1 columns which are preceeding it. Note that $X(\hat{m})^t$ is X followed by t identical columns which are equal to $\hat{m}$, and usually $X(\hat{m})^t$ is different from $X[t(\hat{m})], t \geq 2$. $X(\hat{m})^{t_1}(\bar{\hat{m}})^{t_2}$ is X followed by t_1 identical columns which are equal to $\hat{m}$ and t_2 identical columns which are the complements of the previous t_1 columns.

In the results which follow we will give a partial answer to our two questions. The first lemma is an immediate observation from Definition 6.

Lemma 1: If X is a valid $(d_1, k_1; d_2, k_2)$ array then $X\hat{m}$ is an array with no R runs greater than d_1.

Lemma 2: If XcY is a valid $(d_1, k_1; d_2, k_2)$ array, where the width of Y is $d_1 - 1$, then $\hat{m} = \bar{c}$ in $XcY\hat{m}$.

Proof: Since by Lemma 1, in $XcY\hat{m}$ we do not have an R run with more than d_1 symbols, and no runs in a row, with a possible exception of the first or the last run, can have length less than d_1, it follows that $\hat{m}$ must be different in all the positions from the column preceding it in exactly d_1 positions horizontally, i.e., $\hat{m} = \bar{c}$. □

Corollary 1: If X is a valid $(d_1, k_1; d_2, k_2)$ array then $X\hat{m}$ is a valid $(d_1, k_1; d_2, k_2)$ array.

Given a valid $n_1 \times n_2$ $(d_1, k_1; d_2, k_2)$ array Xc, what is the minimum number of columns that we have to cascade to the right of Xc in order that the resulting array will be R balanced? How many merging columns do we have to cascade to the right of Xc before we can cascade any given (d_2, k_2) RLL sequence e of length n_2? There are a few simple cases.

Case 1: If all the R runs in Xc are of length greater or equal d_1 then by Lemma 2, $\hat{m} = \bar{c}$. In this case $Xc(\bar{c})^{d_1}$ is R d_1 balanced. If all the runs are also less than k_1 then Xce is a valid $(d_1, k_1; d_2, k_2)$ array for any (d_2, k_2) RLL sequence e of length n_1.

In Cases 2 and 3 which follow, we assume that the shortest R run in Xc is of length less than d_1.

Case 2: If the shortest R run of a symbol is t_1, the longest R run is t_2, and $t_2 - t_1 \leq k_1 - d_1$, then $Xc^{d_1 - t_1 + 1}$ is a valid $(d_1, k_1; d_2, k_2)$ array and $Xc^{d_1 - t_1 + 1}(\bar{c})^{d_1}$ is an R d_1 balanced $(d_1, k_1; d_2, k_2)$ array. If $t_2 - t_1 < k_1 - d_1$ then for any (d_2, k_2) RLL sequence e, $Xc^{d_1 - t_1 + 1}e$ is a valid $(d_1, k_1; d_2, k_2)$ array.

Case 3: If the longest R run, which is less than d_1, of a symbol is t and $d_1 + t \leq k_1$, then in $Xc\hat{m}$ the longest R run is $t+1$. Therefore, $Xc(\hat{m})^{d_1}(\bar{\hat{m}})^{d_1}$ is an R d_1 balanced array. If $d_1 + t < k_1$ then for any (d_2, k_2) sequence $e, Xc(\hat{m})^{d_1}e$ is a valid $(d_1, k_1; d_2, k_2)$ array.

Lemma 3: If Xc is a valid $(d_1, k_1; d_2, k_2)$ array, where $k_1 \geq 2d_1 - 1$, then $Y = Xc(\hat{m})^{d_1}$ is a valid $(d_1, k_1; d_2, k_2)$ array.

Proof: By Lemma 2, $\hat{m}$ is a (d_2, k_2) RLL sequence and hence Y has the vertical constraint. Since $\hat{m}$ has the complement of the entries of c in all the rows in which Xc has R runs of lengths greater or equal d_1 and $k_1 \geq 2d_1 - 1$, it follows that $Xc(\hat{m})^{d_1}$ does not have an R run of more than $2d_1 - 1$ symbols, and hence it is a valid $(d_1, k_1; d_2, k_2)$ array. □

Definition 8: Let X, Y, and XZ_1Y be valid $(d_1, k_1; d_2, k_2)$ arrays. Z_1 is called an *optimal merging array* if there is no merging array Z_2 of width less than the width of Z_1, such that XZ_2Y is a valid $(d_1, k_1; d_2, k_2)$ array.

Definition 9: A cascading method for $(d_1, k_1; d_2, k_2)$ arrays is called *optimal* if

1) For any given valid $(d_1, k_1; d_2, k_2)$ arrays X and Y, it produces a merging array Z of width less than or equal to w such that XZY is a valid $(d_1, k_1; d_2, k_2)$ array.
2) There exist two valid $(d_1, k_1; d_2, k_2)$ arrays X_1 and Y_1 such that there is no merging array Z_1 of width less than w for which $X_1Z_1Y_1$ is a valid $(d_1, k_1; d_2, k_2)$ array.

Note that an optimal cascading method does not have to produce optimal merging arrays in all cases.

Corollary 2: If X is a valid $(d_1, k_1; d_2, k_2)$ array, where $k_1 \geq 2d_1$, and e is any (d_2, k_2) RLL sequence e of length n_1 then there exists a merging array Z of width d_1, such that XZe is a valid $(d_1, k_1; d_2, k_2)$ array.

Proof: We generate $X\hat{m}$ and take $Z = (\hat{m})^{d_1}$ to obtain the required merging array. □

Corollary 3: If X and Y are valid $(d_1, k_1; d_2, k_2)$ arrays, where $k_1 \geq 4d_1 - 2$, then there exists a merging array Z of width $2d_1$ such that XZY is a valid $(d_1, k_1; d_2, k_2)$ array.

Proof: Let $\hat{m}_1$ be the resulting column from applying the merge one operator on X and let $\hat{m}_2$ be the resulting column from applying the merge one operator on Y^R. Now, take $Z = (\hat{m}_1)^{d_1}(\hat{m}_2)^{d_1}$ to obtain the required merging array. □

The cascading method presented in Corollary 3 for $k_1 \geq 4d_1 - 2$ is optimal as proved in the following Lemma.

Lemma 4: For any given nonnegative integers, d_1, k_1, d_2, k_2, such that $k_1 \geq 4d_1 - 2$ and $k_2 > d_2$, there exist valid $(d_1, k_1; d_2, k_2)$ arrays X and Y, which do not have a merging array Z of width less than $2d_1$, for which XYZ is a valid $(d_1, k_1; d_2, k_2)$ array.

Proof: We construct valid $(d_1, k_1; d_2, k_2)$ arrays X and Y such that in the first two rows of X there are R runs of length k_1 of zeroes, and in the first (second) row of Y there is an L run of length k_1 of zeroes (ones). But, if one columnis added, then because of the horizontal constraint for the first row, at least d_1 columns are needed, in the merging array, with ones in the first row. Because of the horizontal constraint for the second row, $2d_1$ columns are needed in the merging array. □

Corollary 4: The cascading method of Corollary 3 is optimal.

Lemma 5: If X and Y are valid $(d_1, k_1; d_2, k_2)$ arrays, $k_1 \geq 2d_1$, then there exists a merging array Z of width $4d_1$ such that XZY is a valid $(d_1, k_1; d_2, k_2)$ array.

Proof: Let $\hat{m}_1$ be the resulting column from applying the merge one operator on X and let $\hat{m}_2$ be the resulting column from applying the merge one operator on Y^R. Now, take

$$Z = (\hat{m}_1)^{d_1}(\bar{\hat{m}}_1)^{d_1}(\bar{\hat{m}}_2)^{d_1}(\hat{m}_2)^{d_1}$$

to obtain the required merging array. □

In Corollary 3, we have answered (Q1) for $k_1 \geq 4d_1 - 2$. In Lemma 5, we have answered (Q1) for $4d_1 - 2 > k_1 \geq 2d_1$, but the method used in Lemma 5 is not necessarily optimal. Now, we turn to the most difficult case which is $2d_1 > k_1 \geq d_1 + 1$. We will give a solution for this case in the remainder of this section.

Lemma 6: If Xc is a valid $(d_1, d_1+r; d_2, k_2)$, $1 \leq r \leq d_1 - 1$, array with no R runs greater than d_1 and $0 < t \leq r$, then $Xc^{t+1}[d_1(\hat{m})]$ has R runs of length d_1 in each row where Xc has R runs of length d_1 and R runs of length *minimum* $\{s+t, d_1\}$ in each row where Xc has R runs of length $s < d_1$.

Proof: If row i of Xc has an R run of length d_1, then in Xc^{t+1} row i has an R run of length d_1+t and in $Xc^{t+1}[d_1(\hat{m})]$ row i has an R run of length d_1. If row i of Xc has an R run of length s, $s < d_1$, then in Xc^{t+1} row i has R run of length $s+t$. If $s+t \geq d_1$ then in $Xc^{t+1}[d_1(\hat{m})]$ row i has an R run of length d_1. If $s+t < d_1$ then in $Xc^{t+1}[(d_1 - s - t)(\hat{m})]$ row i has an R run of length d_1 and in $Xc^{t+1}[d_1(\hat{m})]$ row i has an R run of length $s+t$. □

Definition 10: For a valid $(d_1, d_1+r; d_2, k_2)$, $1 \leq r \leq d_1 - 1$, array Xc, with no R runs greater than d_1, and $0 < t \leq r$, the operation $Xc^{t+1}[d_1(\hat{m})]$ is called *t-balancing.*

Definition 11: For a valid $(d_1, d_1+r; d_2, k_2)$, $1 \leq r \leq d_1 - 1$, array X, a *balancing method* is a method which produces a valid $(d_1, d_1+r; d_2, k_2)$ R d_1 balanced array XZ_1.

Definition 12: A balancing method for $(d_1, k_1; d_2, k_2)$ arrays is called *optimal* if

1) For any given valid $(d_1, k_1; d_2, k_2)$ array X, it produces an array Z of width less than or equal to w such that XZ is a valid $(d_1, k_1; d_2, k_2)$ R d_1 balanced array.
2) There exists a valid $(d_1, k_1; d_2, k_2)$ array X_1 such that there is no array Z_1 of width less than w for which X_1Z_1 is a valid $(d_1, k_1; d_2, k_2)$ R d_1 balanced array.

Lemma 7: If Xc is a valid $(d_1, d_1+r; d_2, k_2)$, $1 \leq r \leq d_1 - 1$, array with no R runs greater than d_1 then there exists a valid $(d_1, d_1+r; d_2, k_2)$ R d_1 balanced array XcZ in which the rightmost column of Z is either c or $\bar{c}$.

Proof: By Lemma 6, in the array $Xc^{r+1}[d_1(\hat{m})]$ the R run in each row is either d_1 or greater by r than the run in the same row of Xc, but not exceeding d_1. By Lemma 2, the last column of $Xc^{r+1}[d_1(\hat{m})]$ is $\bar{c}$. If s is the shortest R run in Xc then we apply r-balancing

$$\left\lceil \frac{d_1 - s}{r} \right\rceil - 1$$

times to obtain the array Z' for which the last row is either c or $\bar{c}$. By Lemma 6, the R runs in Z' are of lengths between

$$a = s + r\left\lceil \frac{d_1 - s}{r} \right\rceil - r$$

and d_1. Now, since $0 < d_1 - a \leq r$ it follows that we can apply $(d_1 - a)$-balancing, and by Lemma 6, the new obtained array Z is an R d_1 balanced and by Lemma 2 last column of Z is either c or $\bar{c}$. □

Corollary 5: For a $(d_1, d_1+r; d_2, k_2)$, $1 \leq r \leq d_1 - 1$ array X, with no R runs greater than d_1

$$d_1\left(\left\lceil \frac{d_1 - s}{r} \right\rceil + 1\right) - s$$

merging columns are enough to obtain an R d_1 balanced array, where s is the shortest R run of a symbol in the array.

The balancing implied by Corollary 5 is optimal by considering a valid $(d_1, d_1+r; d_2, k_2)$, $1 \leq r \leq d_1 - 1$ array X, which has all possible R runs between s and d_1. At least

$$d_1\left(\left\lceil \frac{d_1 - s}{r} \right\rceil + 1\right) - s$$

columns are needed to obtain an R d_1 balanced array by adding merging columns to the right of X. We will omit the proof of this claim and leave it to the interested reader.

Lemma 8: If X is a valid $(d_1, d_1+r; d_2, k_2)$, $1 \leq r \leq d_1 - 1$, R d_1 balanced array and eY is a valid $(d_1, d_1+r; d_2, k_2)$ L d_1 balanced array then there exists a merging array Z such that $XZeY$ is a valid $(d_1, d_1+r; d_2, k_2)$ array.

Proof: First note that if the last column of X is $\bar{e}$ then XeY is a valid $(d_1, d_1+r; d_2, k_2)$ array. If the last column of X is e then $X(\bar{e})^{d_1}eY$ is a valid $(d_1, d_1+r; d_2, k_2)$ array. If the last column of X is neither e nor $\bar{e}$ then since X is a valid $(d_1, d_1+r; d_2, k_2)$ R d_1 balanced array it follows that Xe and $X\bar{e}$ are valid $(d_1, d_1+r; d_2, k_2)$ arrays. The shortest R run in Xe is of length 1, and the shortest R run in $X\bar{e}$ is 1. By Lemma 2, $Xe[d_1(\hat{m})]$ and $X\bar{e}[d_1(\hat{m})]$ are valid $(d_1, d_1+r; d_2, k_2)$ arrays with no R runs greater than d_1, and their last column is $\bar{e}$ and e, respectively. The shortest R run of both arrays is of length 1. By Lemma 7 we can form either a valid $(d_1, d_1+r; d_2, k_2)$ array $Xe[d_1(\hat{m})]Z_1e^{d_1}$ or a valid $(d_1, d_1+r; d_2, k_2)$ array $X\bar{e}[d_1(\hat{m})]Z_2e^{d_1}$, which is R d_1 balanced. Hence, either $Xe[d_1(\hat{m})]Z_1eY$ or $X\bar{e}[d_1(\hat{m})]Z_2eY$ is a valid $(d_1, d_1+r; d_2, k_2)$ array. □

Corollary 6: If X is a valid $(d_1, d_1+r; d_2, k_2)$, $1 \leq r \leq d_1 - 1$ R d_1 balanced array and eY is a valid $(d_1, d_1+r; d_2, k_2)$ L d_1 balanced array then there exists an array Z of width at most

$$d_1\left(\left\lceil \frac{d_1 - 1}{r} \right\rceil + 1\right)$$

such that $XZeY$ is a valid $(d_1, d_1+r; d_2, k_2)$ array.

Theorem 1: If X and Y are valid $(d_1, d_1+r; d_2, k_2)$, $1 \leq r \leq d_1 - 1$ arrays, then there exists a merging array Z of width at most

$$3d_1\left(\left\lceil \frac{d_1 - 1}{r} \right\rceil + 1\right)$$

for which XZY is a valid $(d_1, d_1+r; d_2, k_2)$ array.

Proof: By Lemma 1 and Corollary 1, we need to cascade one merging column to the right of X to obtain a valid $(d_1, d_1+r; d_2, k_2)$ array with no R runs greater than d_1. By Corollary 5, at most

$$d_1\left(\left\lceil \frac{d_1 - 1}{r} \right\rceil + 1\right) - 1$$

additional merging columns are needed to obtain a valid R d_1 balanced $(d_1, d_1+r; d_2, k_2)$ array XZ_1. Similarly, at most

$$d_1\left(\left\lceil\frac{d_1-1}{r}\right\rceil+1\right)$$

additional columns are needed to obtain a valid L d_1 balanced $(d_1, d_1+r; d_2, k_2)$ array eZ_2Y. By Corollary 6, at most

$$d_1\left(\left\lceil\frac{d_1-1}{r}\right\rceil+2\right)$$

more merging columns are needed to obtain a valid R d_1 balanced $(d_1, d_1+r; d_2, k_2)$ array $XZ_1Z_3e^{d_1}$. Thus for the valid $(d_1, d_1+r; d_2, k_2)$ array $XZY = XZ_1Z_3eZ_2Y$, the width of Z is

$$3d_1\left(\left\lceil\frac{d_1-1}{r}\right\rceil+1\right). \qquad \square$$

In general, in order to cascade the arrays $X_1, X_2, X_3, \cdots$ by using the merging arrays $Z_1, Z_2, Z_3, \cdots$ to form the global array $X_1Z_1X_2Z_2X_3Z_3\cdots$, we need to identify the merging arrays from the global array. Otherwise, we will not be able to retrieve the information residing in the arrays $X_1, X_2, X_3, \cdots$.

One way to obtain this goal is to use a vector $(i_1, j_1, i_2, j_2, i_3, j_3, \cdots)$, where i_r is the width of X_r and j_r is the width of Z_r.

But typically, this is done by requiring that the arrays $X_1, X_2, X_3, \cdots$ will be of equal width, and the arrays $Z_1, Z_2, Z_3, \cdots$ will be also of equal width. If all the arrays are valid $(d_1, k_1; d_2, k_2)$ arrays we distinguish between three cases.

Case 1: If $k_1 \geq 4d_1-2$ then by Corollary 3 all the merging arrays can have width $2d_1$.

Case 2: If $4d_1-3 \geq k_1 \geq 2d_1$ then by Lemma 5 all the merging arrays can have width $4d_1$.

Case 3: If $2d_1-1 \geq k_1 \geq d_1+1$ then we claim that all the merging arrays can have width

$$3d_1\left(\left\lceil\frac{d_1-1}{r}\right\rceil+1\right)$$

where $r = k_1 - d_1$. Let X and Y be valid $(d_1, d_1+r; d_2, k_2)$, $1 \leq r \leq d_1-1$, arrays. First, we claim that we can obtain a $(d_1, d_1+r; d_2, k_2)$ R d_1 balanced array XT_1 such that the width of T_1 is

$$d_1\left(\left\lceil\frac{d_1-1}{r}\right\rceil+1\right).$$

This is done by constructing $X\hat{m}$, applying r-balancing

$$\left\lceil\frac{d_1-1}{r}\right\rceil-1$$

times and then applying (d_1-a)-balancing, where

$$a = 1 + r\left\lceil\frac{d_1-1}{r}\right\rceil - r.$$

By Lemma 6, the resulting array XT_1 is a valid $(d_1, d_1+r; d_2, k_2)$ R d_1 balanced and the width of T_1 is

$$d_1\left(\left\lceil\frac{d_1-1}{r}\right\rceil+1\right).$$

Similarly, we can obtain a valid $(d_1, d_1+r; d_2, k_2)$ L d_1 balanced array $e^{d_1}T_2Y$ such that the width of $e^{d_1}T_2$ is

$$d_1\left(\left\lceil\frac{d_1-1}{r}\right\rceil+1\right).$$

Finally, we claim that we can obtain a merging array T_3 of width

$$d_1\left(\left\lceil\frac{d_1-1}{r}\right\rceil+1\right)$$

such that $XT_1T_3e^{d_1}T_2Y$ is a valid $(d_1, d_1+r; d_2, k_2), 1 \leq r \leq d_1-1$ array. We define

$$m = \left\lceil\frac{d_1-1}{r}\right\rceil+1, \qquad s = \left\lfloor\frac{d_1}{r}\right\rfloor$$

and distinguish between five cases:

Case 3.1: If the last column of XT_1 is e and m is odd then

$$T_3 = (\bar{e})^{i_1}e^{i_2}(\bar{e})^{i_3}\cdots(\bar{e})^{i_{m-2}}e^{i_{m-1}}(\bar{e})^{i_m}$$

where $i_j = d_1$ for $1 \leq j \leq m$.

Case 3.2: If the last column of XT_1 is $\bar{e}$ and m is even then

$$T_3 = e^{i_1}(\bar{e})^{i_2}e^{i_3}\cdots(\bar{e})^{i_{m-2}}(e)^{i_{m-1}}(\bar{e})^{i_m}$$

where $i_j = d_1$ for $1 \leq j \leq m$.

Case 3.3: If the last column of XT_1 is e and m is even then

$$T_3 = e^r(\bar{e})^{i_1}e^{i_2}(\bar{e})^{i_3}\cdots(\bar{e})^{i_{m-3}}e^{i_{m-2}}(\bar{e})^{i_{m-1}}$$

where $i_j = d_1 + r$ for $1 \leq j \leq s-1$, $i_s = 2d_1 - sr, i_j = d_1$ for $s+1 \leq j \leq m-1$.

Case 3.4: If the last column of XT_1 is $\bar{e}$ and m is odd then

$$T_3 = (\bar{e})^r e^{i_1}(\bar{e})^{i_2}\bar{e}^{i_3}\cdots(\bar{e})^{i_{m-3}}e^{i_{m-2}}(\bar{e})^{i_{m-1}}$$

where $i_j = d_1 + r$ for $1 \leq j \leq s-1$, $i_s = 2d_1 - sr, i_j = d_1$ for $s+1 \leq j \leq m-1$.

Case 3.5: If the last column of XT_1 is neither e or $\bar{e}$ then T_3 is obtained by constructing either $XT_1e[d_1(\hat{m})]$ or $XT_1\bar{e}[d_1(\hat{m})]$, applying r-balancing

$$\left\lceil\frac{d_1-1}{r}\right\rceil-1$$

times and then applying (d_1-a)-balancing, where

$$a = 1 + r\left\lceil\frac{d_1-1}{r}\right\rceil - r.$$

By Lemma 6, both resulting arrays are valid $(d_1, d_1+r; d_2, k_2)$ R d_1 balanced one of them has e as the last column and the second has $\bar{e}$ as the last column. Let $XT_3e^{d_1}$ be the array in which e is the last column.

A simple computation shows that in all these five cases the width of T_3 is $d_1(\lceil\frac{d_1-1}{r}\rceil+1)$ Thus the resulting merging array for $2d_1-1 \geq k_1 \geq d_1+1$ has width

$$3d_1\left(\left\lceil\frac{d_1-1}{r}\right\rceil+1\right).$$

III. Applications of Cascading Constrained Arrays

As stated in the Introduction, cascading is important in encoding and decoding of constrained arrays and in the computation of the capacity rate of the corresponding channels. In this section we will briefly discuss applications of cascading in error correction. Error-correction RLL sequences were considered in [1], [8], [15]. Some interesting methods for error-correction of other constrained codes, e.g., DC-free block codes are discussed in van Tilborg and Blaum [13], Calderbank, Herro, and Telang [3], and Etzion [4]. We now discuss two generalizations to constrained arrays.

The first method is the method of Etzion [4] which was used for DC-free block codes. Assume we have a code A with M distinct $n_1 \times n_2$ $(d_1, k_1, d_2, k_2; d_3, k_3, d_4, k_4)$ arrays and minimum Hamming distance D_1. Assume further that we have a cascading method for $(d_1, k_1, d_2, k_2; d_3, k_3, d_4, k_4)$ arrays. We want to generate a code A' with M $(d_1, k_1, d_2, k_2; d_3, k_3, d_4, k_4)$ arrays, of size $n_1 \times n_3$, and minimum Hamming distance $D_2, D_2 > D_1$, such that n_3 is small as possible.

Let S be the smallest integer such that $2^S \geq n_1 n_2$, and let α be a primitive element in GF(2^S). For a given array $C \in A$, let c_{ij}, $0 \leq i \leq n_1 - 1$, $0 \leq j \leq n_2 - 1$, denote the value of C in row i and column j. We compute the following $D_2 - D_1$ functions:

$$f_m(C) = \sum_{c_{ij}=1} (\alpha^{in_1+j})^{2m-1}, \quad 1 \leq m \leq D_2 - D_1.$$

Assume we have an encoding algorithm E for $(d_1, k_1, d_2, k_2; d_3, k_3, d_4, k_4)$ arrays. Let r be the smallest integer such that E encodes at least 2^S $n_1 \times r$ $(d_1, k_1, d_2, k_2; d_3, k_3, d_4, k_4)$ arrays, i.e., for each integer q, $0 \leq q \leq 2^S - 1$, E encodes q into an $n_1 \times r$ $(d_1, k_1, d_2, k_2; d_3, k_3, d_4, k_4)$ array. For each function $f_m(C), 1 \leq m \leq D_2 - D_1$, we encode $\log_\alpha f_m(C)$, where the logarithm is in GF(2^S), into an $n_1 \times r$ $(d_1, k_1, d_2, k_2; d_3, k_3, d_4, k_4)$ array with the encoding algorithm E. Let $e(f_m(C))$ be the resulting $n_1 \times r$ array of this encoding procedure. Let P_m be the array obtained by cascading $D_2 - 1$ identical copies of $e(f_m(C))$ with appropriate merging arrays between these copies of $e(f_m(C))$. We form the array C' by cascading $CZ_1P_1Z_2P_2 \cdots Z_{D_2-D_1}P_{D_2-D_1}$, where $Z_1, Z_2, \cdots, Z_{D_2-D_1}$ are appropriate merging arrays obtained by the cascading method.

Similarly to the proof in [4] we can show that after applying this procedure on all the M codewords of A we have obtained a code A' with M $(d_1, k_1, d_2, k_2; d_3, k_3, d_4, k_4)$ arrays and minimum Hamming distance D_2. Given a transmitted array from A' the decoding of this array obtained by this method will be done in a very similar way to the decoding procedure in [4]. By considering the arguments in [4] we can take as P_m the array obtained by cascading $D_2 - m$ copies of $e(f_m(C))$. The result is again a code with M $(d_1, k_1, d_2, k_2; d_3, k_3, d_4, k_4)$ arrays and minimum Hamming distance D_2 and shorter horizontal width. The only advantage of the method we have presented is that its decoding algorithm can be presented in a simpler way.

The second method is based on the existence of error-correcting codes for RLL sequences and the existence of "good" error-correcting codes over arbitrary alphabets. We will consider only $(d_1, k_1; d_2, k_2)$ arrays. Assume we have

1) a code C of length n and minimum Hamming distance D_1, over an alphabet Σ with σ symbols, and M codewords;
2) a code with σ (d_2, k_2) RLL sequences of length $n_1, s_1, \cdots, s_\sigma$, and minimum Hamming distance D_2;
3) a $1-1$ mapping f from Σ to the σ (d_2, k_2) sequences.

Given a codeword

$$c \in C, \quad c = (c_{i_1}, c_{i_2}, \cdots c_{i_n})$$

we apply the mapping f and obtain n (d_2, k_2) RLL sequences of length $n_1, (s_{i_1}, s_{i_2}, \cdots, s_{i_n})$, where $f(c_{i_j}) = s_{i_j}, 1 \leq j \leq n$. Now, we distinguish between three cases:

Case 1: $k_1 = qd_1 + r$, $3 \leq q$, $0 \leq r \leq d_1 - 1$, we generate the array

$$\begin{aligned}&(s_{i_1})^{d_1}(s_{i_2})^{d_1} \cdots (s_{i_q})^{d_1}(\bar{s}_{i_q})^{d_1}(s_{i_{q+1}})^{d_1} \cdots \\ &(s_{i_{2q-1}})^{d_1}(\bar{s}_{i_{2q-1}})^{d_1}(s_{i_{2q}})^{d_1} \cdots \\ &(s_{i_{3q-1}})^{d_1}(\bar{s}_{i_{3q-1}})^{d_1}(s_{i_{3q}})^{d_1} \cdots (s_{i_n})^{d_1}.\end{aligned}$$

Clearly, this is a valid $(d_1, k_1; d_2, k_2)$ array of size

$$n_1 \times \left(nd_1 + \left\lceil \frac{n-q}{q-1} \right\rceil d_1\right).$$

Using this method on all the codewords of C we obtain a code with M $(d_1, k_1; d_2, k_2)$ arrays and minimum Hamming distance $D_1 D_2 d_1$.

Case 2: $k_1 = 2d_1 + r$, $0 \leq r \leq d_1 - 1$, we generate the array

$$(s_{i_1})^{d_1}(\bar{s}_{i_1})^{d_1}(s_{i_2})^{d_1}(\bar{s}_{i_2})^{d_1} \cdots (s_{i_n})^{d_1}(\bar{s}_{i_n})^{d_1}.$$

Clearly, this is a valid $(d_1, k_1; d_2, k_2)$ array of size $n_1 \times (2nd_1)$. Using this method on all the codewords of C we obtain a code with M $(d_1, k_1; d_2, k_2)$ arrays and minimum Hamming distance $2D_1 D_2 d_1$.

Case 3: $k_1 = d_1 + r$, $1 \leq r \leq d_1 - 1$, we generate the array

$$\begin{aligned}&s^{d_1}(s_{i_1})^r(\bar{s})^{d_1}s^{d_1}(s_{i_2})^r(\bar{s})^{d_1}s^{d_1}(s_{i_3})^r \cdots \\ &(\bar{s})^{d_1}s^{d_1}(s_{i_n})^r(\bar{s})^{d_1}\end{aligned}$$

where s is any (d_2, k_2) RLL sequence of length n_1. Note that another way to write the same array is

$$\begin{aligned}&s^{d_1}(s_{i_1})^r(\hat{m})^{d_1}(\hat{m})^{d_1}(s_{i_2})^r(\hat{m})^{d_1}(\hat{m})^{d_1} \cdots \\ &(\hat{m})^{d_1}(\hat{m})^{d_1}(s_{i_n})^r(\hat{m})^{d_1}.\end{aligned}$$

Clearly, this is a valid $(d_1, k_1; d_2, k_2)$ array of size $n_1 \times (nr + 2nd_1)$. Using this method on all the codewords of C we obtain a code with M $(d_1, k_1; d_2, k_2)$ arrays and minimum Hamming distance rD_1D_2.

IV. Conclusion

Given two valid $(d_1, k_1; d_2, k_2)$ arrays X and Y, with the same vertical size, with $k_1 > d_1$ and $k_2 > d_2$, we have shown how to find a merging array Z such that XZY is a valid $(d_1, k_1; d_2, k_2)$ array. In our construction methods we have distinguished between three cases.

Case 1: $k_1 \geq 4d_1 - 2$, in which our method is optimal.

Case 2: $k_1 \geq 2d_1$.

Case 3: $k_1 \geq d_1 + 1$, in which we have used an optimal balancing method.

It is not clear whether the methods used for Cases 2 and 3 are optimal. Also, we would like to see optimal cascading methods which produce optimal merging arrays in all cases. Another problem for further research is finding cascading methods for other constraints, and to specify exactly when we can cascade two valid constrained arrays and when we cannot.

Acknowledgment

The author wishes to thank V. K. Wei for helpful discussions, and the referees for their comments, suggestions, corrections, and careful reading of this manuscript. He also wishes to thank A. Vardy and R. Talyansky for their helpful comments.

References

[1] K. A. S. Abdel-Ghaffar and J. H. Weber, "Bounds and constructions for runlength-limited error-control block codes," *IEEE Trans. Inform. Theory*, vol. 37, pp. 789–800, 1991.
[2] G. F. M. Beenker and K. A. S. Immink, "A generalized method for encoding and decoding run-length-limited binary sequences," *IEEE Trans. Inform. Theory*, vol. IT-29, pp. 751–754, 1983.
[3] A. R. Calderbank, M. A. Herro, and V. Telang, "A multi-level approach to the design of dc-line codes," *IEEE Trans. Inform. Theory*, vol. 36, pp. 579–583, 1989.
[4] T. Etzion, "Constructions of error-correcting DC-free block codes," *IEEE Trans. Inform. Theory*, vol. 36, pp. 899–905, 1990.
[5] T. Etzion and V. Wei, "On two-dimensional run-length-limited codes," presented at the IEEE Int. Workshop on Information Theory, Salvador, Brazil, June 1992.
[6] K. A. S. Immink, *Coding Techniques for Digital Recorders*. London, UK: Prentice-Hall Int., 1991.
[7] ——, "Runlength-limited sequences," *Proc. IEEE*, vol. 78, pp. 1745–1759, 1990.
[8] P. Lee and J. K. Wolf, "A general error-correcting code construction for run-length limited binary channels," *IEEE Trans. Inform. Theory*, vol. 35, pp. 1330–1335, 1989.

[9] E. K. Orcutt and M. W. Marcellin, “Enumerable multi-track (d, k) block codes,” *IEEE Trans. Inform. Theory*, vol. 39, pp. 1738–1744, 1993.

[10] ——, “Redundant multi-track (d, k) codes,” *IEEE Trans. Inform. Theory*, vol. 39, pp. 1744–1750, 1993.

[11] T. Pavlidis, J. Swartz, and Y. P. Wang, “Fundamentals of bar code information theory,” *Computer*, vol. 23, pp. 74–86, 1990.

[12] D. L. Tang and L. R. Bahl, “Block codes for a class of constrained noiseless channels,” *Inform. Contr.*, vol. 17, pp. 436–461, 1970.

[13] H. van Tilborg and M. Blaum, “On error-correcting balanced codes,” *IEEE Trans. Inform. Theory*, vol. 35, pp. 1091–1095, 1989.

[14] J. H. Weber and K. A. S. Abdel-Ghaffar, “Cascading runlength-limited sequences,” *IEEE Trans. Inform. Theory*, vol. 39, pp. 1976–1984, 1993.

[15] Ø. Ytrehus, “Upper bounds on error-correcting runlength-limited block codes,” *IEEE Trans. Inform. Theory*, vol. 37, pp. 941–945, 1991.

Two-Step Trellis Decoding of Partial Unit Memory Convolutional Codes

Marianne Fjelltveit Hole and Øyvind Ytrehus, *Member, IEEE*

***Abstract*— We present a new soft-decision decoding method for high-rate convolutional codes. The decoding method is especially well-suited for PUM convolutional codes. The method exploits the linearity of the parallel transitions in the trellis associated with PUM codes. We provide bounds on the number of operations per decoded bit, and show that this number is dependent on the weight hierarchy of the linear block code associated with the parallel transitions. The complexity of the new decoding method for PUM codes is compared to the complexity of Viterbi decoding of comparable punctured convolutional codes. Examples from a special class of PUM codes show that the new decoding method compare favorably to Viterbi decoding of punctured codes.**

***Index Terms*— Decoding, partial unit memory convolutional codes, weight hierarchy.**

I. Introduction

Consider an application where Viterbi decoding of a high-rate convolutional code is used. A punctured representation of the convolutional code is often selected because it reduces the number of operations per decoded bit significantly compared to a nonpunctured representation. The path memory size is almost the same for the punctured and the nonpunctured representation of the code. If instead we assume that a Partial Unit Memory (PUM) convolutional code is used, the constraint length is often smaller than the constraint length of a comparable punctured convolutional code. Many authors have investigated the class of PUM codes [1]–[4]. It is known that any convolutional code may be represented as a PUM code. We present a modification of the Viterbi algorithm for PUM codes that often results in fewer operations per decoded bit than Viterbi decoding of comparable punctured codes, and which needs a smaller path memory because of the smaller constraint length.

A convolutional code with rate k/n, constraint length ν, and free distance $d_{\rm free}$, is said to be an $(n, k, \nu, d_{\rm free})$ convolutional code. For an $(n, k, \nu, d_{\rm free})$ PUM convolutional code, $k > \nu$. Hence, there are parallel branches between states in the trellis representing the PUM code. The labels on these parallel branches constitute cosets of an $[n, k-\nu, d_{\min}]$ block code. This block code is defined by the labels on the branches starting and ending in state zero of the PUM code trellis. The decoding of the block code and the cosets reduces the $2^{k-\nu}$ parallel branches between any pair of states in the PUM code trellis to only one branch.

Assuming the Viterbi algorithm and using the number of operations per decoded bit as a complexity measure, the trellis with fewest states that represents a block code is the best trellis representation [5], [6]. We show what the best trellis representation is when decoding, in addition to the block code, all the cosets of the block code. Specially, an upper and a lower bound on the total number of operations needed to decode the block code and all its cosets, are given. For block codes satisfying the chain condition [7], [8], it is shown how to determine the best trellis representation which attains the lower bound. We also give examples from a special class of PUM codes where not all words of the cosets of the block code can be found as branch labels in the PUM code trellis. These codes make the most of the new decoding technique.

The remainder of the correspondence is organized as follows. Section II describes punctured convolutional codes and gives a motivating example. In Section III, the new modified Viterbi decoding for PUM codes is introduced. Next, in Section IV, bounds on the number of operations needed to decode a block code and all its cosets are provided. A connection between the weight hierarchy of the block code and the trellis representing the block code and all its cosets, is presented. Section V shows how to design codes well suited for the new decoding technique. In Section VI, the complexity of the new decoding of PUM codes is compared to the Viterbi decoding of punctured codes. Section VII contains a brief conclusion.

It is assumed that the reader is familiar with the theory for convolutional codes and encoding matrices as presented by Forney [9], Johannesson and Wan [10], or Dholakia [11].

II. Punctured Codes and Their Decoding

Consider a trellis representing an $(n, k, \nu, d_{\rm free})$ convolutional code. The trellis has 2^k outgoing branches from each state. The branches have labels consisting of n encoded bits. If the Viterbi algorithm is applied to the trellis representing the $(n, k, \nu, d_{\rm free})$ code, the number of operations needed in one depth of the trellis is $2^\nu \cdot (2^k \cdot n \text{ additions} + (2^k - 1) \text{ comparisons})$. Note that this is the number of operations needed to decode k information bits.

Punctured convolutional codes constitute a subclass of ordinary convolutional codes. The number of operations per decoded bit is significantly less for punctured convolutional codes than for ordinary convolutional codes when the Viterbi algorithm is used. A punctured rate k/n convolutional code can be generated by a rate $1/n'$ convolutional code in the following way [12]:

- Encode k bits by the original rate $1/n'$ encoder. The corresponding output has length $k \cdot n'$.
- By periodically deleting $k \cdot n' - n$ of the encoded bits in this output, the k input bits generate n encoded bits.

The trellis representing the rate $1/n'$ convolutional code has in each depth only two outgoing branches from each state, and each branch label has n' encoded bits. Therefore, only one comparison and $2 \cdot n'$ additions are needed per state when applying the Viterbi algorithm. In depths where bits are being deleted the number of additions per state is less than $2 \cdot n'$. The total number of operations per state is still $2 \cdot n' + 1$ because the additions are replaced by

Manuscript received September 25, 1995; revised June 8, 1996. This work was supported by the Norwegian Research Council, NFR.

The authors are with the Department of Informatics, University of Bergen, N-5020 Bergen, Norway.

Publisher Item Identifier S 0018-9448(97)00098-9.