# Arbitrarily Low Redundancy Construction of Balanced Codes Using Realistic Resources

Danny Dubé
Department of Computer Science and Software Engineering, Université Laval
Quebec City, Quebec, Canada
Email: Danny.Dube@ift.ulaval.ca

***Abstract*—Recently, Dubé and Mechqrane presented a construction technique to encode plain data into balanced codewords. The construction is based on permutations, the well known arcade game Pacman, and limited-precision integers. The redundancy that is introduced by their construction is particularly low. The results are noticeably better than those of previous work. Still, the resources required by their technique remain modest: there is no need for large lookup tables, no need to perform costly calculations using large integers, and yet the time and space complexity for encoding or decoding a block is linear. Although their technique allows one to achieve the best per-block redundancy, the per-block aspect of the technique prevents the achievement of the optimal redundancy, in a global sense. In this paper, we extend the technique so that we can achieve a better redundancy than the best per-block one and arbitrarily approach the optimal global redundancy.**

## I. Introduction

### A. Balanced Blocks

A block $B$ of $M$ bits is said to be *balanced* if it contains an equal number of zeros and ones. Note that $M$ has to be an even number. Applications of balanced codes are mentioned in Subsection I-B. Transforming arbitrary input data into balanced blocks is performed using an *encoding function* 'Enc'. Such an encoding necessarily introduces redundancy. Indeed, only $\binom{M}{M/2}$ of the $2^M$ blocks of $M$ bits happen to be balanced. Let $\mathcal{B}_M$ be the set of the $M$-bit balanced blocks.

In this work, we assume that the application imposes a specific size $M$ for the balanced blocks. Moreover, we assume that the input data is binary and purely random. Finally, we consider that only a strict balance is acceptable for our balanced blocks. Other work sometimes adopts a looser definition of balanced blocks; e.g., by allowing odd values for $M$.

Let $Q$ be the size of each block of input data that gets transformed into a balanced block. Then the encoding function is $\mathrm{Enc} : 2^Q \to \mathcal{B}_M$. Let $\mathrm{Dec} : \mathcal{B}_M \to 2^Q$ be the corresponding decoding function. To ensure decodability, 'Enc' has to be injective, which implies that $2^Q \leq \binom{M}{M/2}$. Let $R = M - Q$ be the number of bits of redundancy that 'Enc' introduces per block. Obviously, the smaller $R$ is, the better 'Enc' is. 'Enc' is a fixed-to-fixed code. Figure 1 shows lookup tables for two different choices of $M$ and $Q$.

### B. Motivation

Balanced codes have many applications. They can be used to detect unidirectional errors [1], to detect errors due to low-frequency disturbances in magnetic storage [2], to reduce noise in VLSI integrated circuits [3], and for many other purposes.

### C. Previous Work

*1) Lookup Tables:* Mathematically, devising optimal functions 'Enc' and 'Dec' is a trivial task. First, one determines $Q$ from $M$; so we let $Q$ be $\left\lfloor \log \binom{M}{M/2} \right\rfloor$.[1] Second, one may use enumerative coding to define 'Enc' (and 'Dec') [4]. To do so, one enumerates the $2^Q$ unconstrained input blocks in lexicographic order and the first $2^Q$ $M$-bit balanced blocks also in lexicographic order and then lets 'Enc' be the one-to-one mapping from the former to the latter. The mapping that defines 'Enc' (and 'Dec') may be stored in a *lookup table*, like those shown in Figure 1. Unfortunately, the strategies based on such lookup tables are not practical because they do not scale well. The size of lookup tables increases exponentially with $Q$. For example, input blocks that are merely 1/16-th of a kilobyte ($Q = 512$) would require a lookup table that has much more entries than there are atoms in the universe.

*2) Enumerative Coding via Calculations:* Alternatively, the same mapping may be implemented using a pair of procedures that build, by calculations, the $i$-th balanced block when presented the $i$-th input block, and vice versa. Unfortunately, these procedures, which are based on calculations, require the manipulation of large integers and this is costly in time. The impracticality of enumerative coding has lead many researchers to develop faster, approximate strategies.

*3) Knuth's Construction Technique:* Knuth presented the first practical construction technique for balanced blocks [5]. His technique is quite simple and it is based on the following observation: an arbitrary block $w$ of bits can be made balanced by *inverting* the bits in an appropriate prefix of $w$. Let us denote by $\overline{\cdot}$ the inversion operator; i.e. $\overline{0} = 1$ and $\overline{1} = 0$ and extend the operator so that it operates bitwise on sequences. Given an arbitrary block $w$ of *even* length $Q$, Knuth's technique consists in splitting $w$ into a prefix $u$ and a suffix $v$, where $0 \leq |u| < Q$, such that $\overline{u} \cdot v$ is balanced. Knuth showed that such an appropriate prefix always exists. Merely transforming input blocks that way would not make a valid (i.e. reversible) 'Enc' function. The length $|u|$ has to be encoded somewhere in the transformed block. To do so, Knuth's technique recursively relies on a shorter balanced

[1] In this paper, all logarithms are to the base 2.

| Input | Balanced | Input | Balanced | Input | Balanced | Input | Balanced |
|---|---|---|---|---|---|---|---|
| 00 | 0011 | 01 | 0101 | 10 | 0110 | 11 | 1001 |

| Input | Balanced | Input | Balanced | Input | Balanced | Input | Balanced |
|---|---|---|---|---|---|---|---|
| 0000 | 000111 | 0100 | 010011 | 1000 | 011010 | 1100 | 100110 |
| 0001 | 001011 | 0101 | 010101 | 1001 | 011100 | 1101 | 101001 |
| 0010 | 001101 | 0110 | 010110 | 1010 | 100011 | 1110 | 101010 |
| 0011 | 001110 | 0111 | 011001 | 1011 | 100101 | 1111 | 101100 |

Fig. 1. Lookup tables for (a) $Q = 2$, $M = 4$ and (b) $Q = 4$, $M = 6$.

| Input | Balanced | Input | Balanced | Input | Balanced |
|---|---|---|---|---|---|
| 00000 | 0011·0011 | 01011 | 0101·1100 | 10110 | 1001·1010 |
| 00001 | 0011·0101 | 01100 | 0110·0011 | 10111 | 1001·1100 |
| 00010 | 0011·0110 | 01101 | 0110·0101 | 11000 | 1010·0011 |
| 00011 | 0011·1001 | 01110 | 0110·0110 | 11001 | 1010·0101 |
| 00100 | 0011·1010 | 01111 | 0110·1001 | 11010 | 1010·0110 |
| 00101 | 0011·1100 | 10000 | 0110·1010 | 11011 | 1010·1001 |
| 00110 | 0101·0011 | 10001 | 0110·1100 | 11100 | 1010·1010 |
| 00111 | 0101·0101 | 10010 | 1001·0011 | 11101 | 1010·1100 |
| 01000 | 0101·0110 | 10011 | 1001·0101 | 11110 | 1100·0011 |
| 01001 | 0101·1001 | 10100 | 1001·0110 | 11111 | 1100·0101 |
| 01010 | 0101·1010 | 10101 | 1001·1001 | | |

Fig. 2. Lookup table for $Q = 5$, $N = 2$, $M = 4$.

code. The codewords of the latter have length $R$, where $R$ is large enough to encode $|u|$; i.e. $\binom{R}{R/2} \geq Q$. Typically, $R$ is small enough to use a lookup table and avoid deeper recursion. So Knuth's technique encodes $w$ by returning $\mathrm{Enc}(|u|) \cdot \overline{u} \cdot v$.

Knuth estimated the redundancy added by his technique to be $R \approx \log Q \approx \log M$ bits, which is about twice the optimal one: $M - \log \binom{M}{M/2} \approx \frac{1}{2} \log M$.

*4) Variants of Knuth's Technique:* Indeed, much research has been conducted to reduce the redundancy of Knuth's algorithm. Weber and Immink noted that an input block $w$ may have multiple (between 1 and $Q/2$) adequate prefixes [6]. This freedom in selecting encodings is the cause for part of the extra redundancy introduced by Knuth's algorithm. The same authors also noted that, in theory, this selection freedom could be used to convey information and they showed that, on average, the amount of information that could be conveyed per block this way is $A_{\mathrm{SF}} \approx \frac{1}{2} \log Q - 0.916 \approx \frac{1}{2} \log M - 0.916$ [7]. They devised a scheme that significantly reduces the redundancy compared to Knuth's algorithm. Still, they did not succeed to fully exploit the selection freedom.

Al-Rababa'a et al. noticed that this selection freedom is a good candidate for bit recycling [8]. Their technique achieved a better improvement by transmitting almost $A_{\mathrm{SF}}$ extra bits per balanced block, on average.

*5) Construction Using Permutations and Limited-Precision Integers:* Recently, Dubé and Mechqrane presented a completely different construction technique, after being brought to the belief that it was not possible to improve variants of Knuth's technique further [9]. This new technique was primarily inspired by the observation that, inside of any permutation of the first $M$ naturals, hides a balanced block of $M$ bits. The rest of the machinery used by the technique consists in performing calculations similar to those of enumerative coding but without ever manipulating large integers. Indeed, one of the tools of the machinery is a special Pacman[2] that consumes *and produces* "pills of information"; see Section II.

### D. Contributions

The contribution in this paper is made of two parts.

First, we improve the redundancy that is achieved by the construction of balanced blocks, with respect to that of the Dubé-Mechqrane technique. Note that the Dubé-Mechqrane technique can be used to efficiently achieve the best possible redundancy for a *per-block* encoding. However, we improve the technique to approach the optimal redundancy in a global sense; i.e. when transforming a large amount of input data into a large number of balanced blocks. In order to illustrate our strategy, we compare Figures 1(a) and 2. We first note that, since a single balanced block of size 4 may only be one of 6 codewords, we can encode at most 2 (integral) input bits into it; see Figure 1(a). On the other hand, if we consider two balanced blocks of size 4 at once, the pair of codewords can be any of 36 pairs. This selection freedom allows up to 5 (integral) input bits to be encoded into a pair of balanced blocks. This leads to an average of 2.5 input bits per balanced block; see Figure 2.

Second, we try to clarify the presentation of the technique, which is based on permutations, Pacman, and limited-precision integers. The authors of the previous technique, Dubé and Mechqrane, received feedback from readers that indicated that the technique was not presented clearly enough.

## II. The Dubé-Mechqrane Technique

The technique uses a variety of tools. In this section, we introduce a minimal set of notions, just enough to describe the contribution of the paper. The complete presentation of the Dubé-Mechqrane technique includes the processes of initialization, termination, and decoding, as well as the notion of valid programming. In order to save space, we skip these and we refer the reader to the original paper [9].

### A. The Main Encoding Algorithm

The operations performed by the encoding function 'Enc' of the Dubé-Mechqrane technique are pictured in Figure 3. An invocation of 'Enc' takes $w \in 2^Q$ as input and emits $B \in \mathcal{B}_M$ as output. Note that 'Enc' holds a state which is preserved from one invocation to the next. This implies that $B$ is not built from $w$ alone; also, not all the information that $w$ contains gets transferred into $B$ at once. During an invocation, the information contained in $w$ gets blended with the state that was preserved in memory. This results in a blob of information that then gets separated into $B$ and a new state, which

[2]The name is inspired by the well known PAC-MAN video game. The trademark PAC-MAN is a property of BANDAI NAMCO.

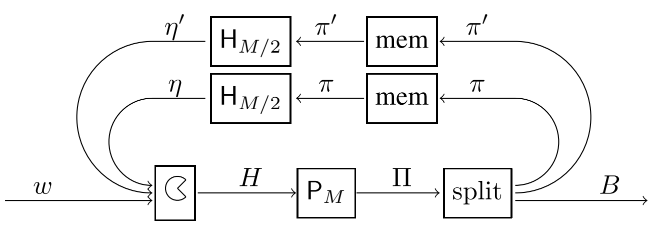


Fig. 3. Information flow inside of function 'Enc'.

| **Convent.** | **Indexed** | **Convent.** | **Indexed** | **Convent.** | **Indexed** |
|---|---|---|---|---|---|
| $(1,2,3,4)$ | $\langle 1,2,3,4\rangle$ | $(2,3,1,4)$ | $\langle 1,1,2,4\rangle$ | $(3,4,1,2)$ | $\langle 1,2,1,2\rangle$ |
| $(1,2,4,3)$ | $\langle 1,2,3,3\rangle$ | $(2,3,4,1)$ | $\langle 1,1,2,3\rangle$ | $(3,4,2,1)$ | $\langle 1,1,1,2\rangle$ |
| $(1,3,2,4)$ | $\langle 1,2,2,4\rangle$ | $(2,4,1,3)$ | $\langle 1,1,3,2\rangle$ | $(4,1,2,3)$ | $\langle 1,2,3,1\rangle$ |
| $(1,3,4,2)$ | $\langle 1,2,2,3\rangle$ | $(2,4,3,1)$ | $\langle 1,1,2,2\rangle$ | $(4,1,3,2)$ | $\langle 1,2,2,1\rangle$ |
| $(1,4,2,3)$ | $\langle 1,2,3,2\rangle$ | $(3,1,2,4)$ | $\langle 1,2,1,4\rangle$ | $(4,2,1,3)$ | $\langle 1,1,3,1\rangle$ |
| $(1,4,3,2)$ | $\langle 1,2,2,2\rangle$ | $(3,1,4,2)$ | $\langle 1,2,1,3\rangle$ | $(4,2,3,1)$ | $\langle 1,1,2,1\rangle$ |
| $(2,1,3,4)$ | $\langle 1,1,3,4\rangle$ | $(3,2,1,4)$ | $\langle 1,1,1,4\rangle$ | $(4,3,1,2)$ | $\langle 1,2,1,1\rangle$ |
| $(2,1,4,3)$ | $\langle 1,1,3,3\rangle$ | $(3,2,4,1)$ | $\langle 1,1,1,3\rangle$ | $(4,3,2,1)$ | $\langle 1,1,1,1\rangle$ |

Fig. 4. Correspondence between the permutations in $\mathcal{P}_4$ and $\mathcal{H}_4$.

is saved in memory. 'Enc' manipulates permutations under two representations: the *conventional* one and the *indexed* one; see Subsections II-B and II-C. The first step during an invocation consists in recovering the state $\pi, \pi' \in \mathcal{P}_{M/2}$ from memory and converting these permutations to the indexed representation, giving $\eta, \eta' \in \mathcal{H}_{M/2}$. The second step is performed by Pacman, which transfers all the information contained in $w$, $\eta$, and $\eta'$ to a new permutation $H \in \mathcal{H}_M$; see Subsections II-E and II-F. The third step converts $H$ to the conventional representation, giving $\Pi \in \mathcal{P}_M$. In the final step, 'split' extracts $B$ from $\Pi$, as well as two new permutations $\pi, \pi' \in \mathcal{P}_{M/2}$; see Subsection II-D. Figure 3 makes it clear that no permutations are ever input or output; they are only part of the internal state of 'Enc'. We point out that all operations performed by 'Enc' are injective. Even more: all operations except the one performed by Pacman are bijective. The injectivity of the operations makes decoding possible. Decoding has to proceed backwards.

### B. Conventional Representation of Permutations

We denote a (conventional) *permutation* of $n$ elements by by $(a_1, \ldots, a_n)$, where $a_i \neq a_j$ whenever $1 \leq i < j \leq n$. We define $\mathcal{P}_n$ to be the set of permutations of $\{1, \ldots, n\}$.

### C. Indexed Representation of Permutations

The indexed representation indicates the *relative position* of each of the numbers that appear in a permutation $\pi \in \mathcal{P}_n$. The leftmost position is 1. We use the term "relative" because the indexed representation indicates, for each number $a$, the position of $a$ in the permutation that remains if we remove the larger numbers $a+1, \ldots, n$ from $\pi$. We denote an indexed permutation $\eta$ by $\langle \iota_1, \ldots, \iota_n \rangle$, where $1 \leq \iota_i \leq i$, for $1 \leq i \leq n$. That is, $\iota_1$ is necessarily 1, $\iota_2$ can be 1 or 2, $\iota_3$ can be 1, 2 or 3, and so on. Let $\mathcal{H}_n$ be the set of indexed permutations with $n$ indices. The conversion of permutations from the conventional representation to the indexed representation is performed using a family of bijective functions, $\{\mathsf{H}_n\}_{n=1}^{\infty}$, where $\mathsf{H}_n : \mathcal{P}_n \rightarrow \mathcal{H}_n$, which are inductively defined as follows.

$$\begin{aligned} \mathsf{H}_1((1)) &= \langle 1 \rangle \\ \mathsf{H}_n((a_1, \ldots, a_{i-1}, n, a_{i+1}, \ldots, a_n)) &= \\ \mathsf{H}_{n-1}((a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n)) \cdot \langle i \rangle & \end{aligned}$$

Note that we overload the operator '$\cdot$' to also denote the extension of a permutation. The reverse conversion is performed using the family of functions $\{\mathsf{P}_n\}_{n=1}^{\infty}$, where $\mathsf{P}_n : \mathcal{H}_n \rightarrow \mathcal{P}_n$. Figure 4 illustrates the correspondence between the conventional permutations of $\mathcal{P}_4$ and the indexed ones of $\mathcal{H}_4$. As an example, let us explain why the conventional permutation $\pi = (3, 1, 4, 2)$ corresponds to the indexed permutation $\eta = \langle 1, 2, 1, 3 \rangle$; i.e. why $\eta$ is $\mathsf{H}_4(\pi)$. First, 4 is located in the third position in $\pi$, so $\eta = \mathsf{H}_3((3, 1, 2)) \cdot \langle 3 \rangle$. Second, 3 is located in the first position in $(3, 1, 2)$, so $\mathsf{H}_3((3, 1, 2)) = \mathsf{H}_2((1, 2)) \cdot \langle 1 \rangle$. Third, $\mathsf{H}_2((1, 2)) = \mathsf{H}_1((1)) \cdot \langle 2 \rangle$. Finally, $\mathsf{H}_1((1)) = \langle 1 \rangle$. So we conclude that $\eta = \langle 1, 2, 1, 3 \rangle$. An interesting property of the indexed permutations is that every index is independent from the others, which is not the case with conventional permutations. This property is extremely useful in the reconstruction of a large permutation from an input block and two small permutations; see the Pacman operation in Figure 3 as well as Subsection II-E.

### D. Permutations and Balanced Blocks

Permutations have some relation to balanced codes. Let $M$ be an even integer. Let us suppose further that we have at hand a permutation $\Pi \in \mathcal{P}_M$. Then we can extract a balanced block $B \in \mathcal{B}_M$ from $\Pi$ by keeping the *parity* of the elements of $\Pi$. Let us denote this operation by $B = \Pi \bmod 2$. For example, if $\Pi$ is $(5, 4, 2, 7, 1, 8, 3, 6)$, then $B = \texttt{10011010}$ can be extracted.

This extraction process can be put to good use in an encoding procedure. If we could transform some input data into a permutation like $\Pi$, then we would be able to extract a balanced block $B$ from $\Pi$, and $B$ would carry some information about that input data. However, these operations can only hope to constitute a part of an encoding procedure. The key word here is that $B$ would only carry "some" information about the input data. Most of the information about the input data would remain in $\Pi$. Note that the remaining information in $\Pi$ cannot be discarded, in order to allow an eventual decoder to recover the original input data.

Let us characterize the information that remains in $\Pi$ once $B$ has been extracted. In order to do so, let us take the point of view of the decoder and assume that $B$ is known but not $\Pi$. $B$ describes the positions of the even and odd numbers inside of $\Pi$. However, nothing is divulged about the order of the even numbers relative to each other, neither regarding the odd numbers. If the decoder were to receive the relative order of the even numbers and that of the odd numbers, then the

$$
\begin{array}{llllll}
 & w & \eta & \eta' & \mapsto & H \\
\textit{Before}: & b_1\ b_2\ b_3 \ldots\ b_Q & \left\langle\, \iota_1,\ \iota_2,\ \iota_3, \ldots,\ \iota_{M/2} \right\rangle & \left\langle\, \iota'_1,\ \iota'_2,\ \iota'_3, \ldots,\ \iota'_{M/2} \right\rangle & \mapsto & \left\langle \Box_1, \Box_2, \Box_3, \ldots, \Box_M \right\rangle \\
\textit{During}: & b_1\, \Box_2\ b_3 \ldots\ b_Q & \left\langle \Box_1,\ \iota_2, \Box_3, \ldots,\ \iota_{M/2} \right\rangle & \left\langle \Box_1, \Box_2,\ \iota'_3, \ldots, \Box_{M/2} \right\rangle & \mapsto & \left\langle \Box_1,\ \iota''_2,\ \iota''_3, \ldots, \Box_M \right\rangle \\
\textit{After}: & \Box_1\, \Box_2\, \Box_3 \ldots \Box_Q & \left\langle \Box_1, \Box_2, \Box_3, \ldots, \Box_{M/2} \right\rangle & \left\langle \Box_1, \Box_2, \Box_3, \ldots, \Box_{M/2} \right\rangle & \mapsto & \left\langle\, \iota''_1,\ \iota''_2,\ \iota''_3, \ldots,\ \iota''_M \right\rangle
\end{array}
$$

Fig. 5. Progressive information transfer during the execution of Pacman's programming.

decoder would hold full information about $\Pi$. These orders are equivalent, up to renumbering, to permutations of $M/2$ elements. So there is a one-to-one mapping between permutations like $\Pi \in \mathcal{P}_M$ and triples like $(B, \pi, \pi') \in \mathcal{B}_M \times \mathcal{P}_{M/2} \times \mathcal{P}_{M/2}$. We define $\mathrm{split} : \mathcal{P}_M \to \mathcal{B}_M \times \mathcal{P}_{M/2} \times \mathcal{P}_{M/2}$ as that bijective function.

In the example of $\Pi = (5,\ 4,\ 2,\ 7,\ 1,\ 8,\ 3,\ 6)$, we have $\mathrm{split}(\Pi) = (B, \pi, \pi')$, where $B = \mathtt{10011010}$, $\pi = (2,\ 1,\ 4,\ 3)$, and $\pi' = (3,\ 4,\ 1,\ 2)$. Note that $\pi$ and $\pi'$ are the renumberings of $(4,\ 2,\ 8,\ 6)$ and $(5,\ 7,\ 1,\ 3)$, respectively.

### *E. Rebuilding a Large Permutation from Small Permutations*

Since the information that remains in $\pi$, $\pi' \in \mathcal{P}_{M/2}$ should not be discarded, it should be injected into a new $\Pi \in \mathcal{P}_M$. The indexed representation of permutations is helpful, here. Indeed, each index can be seen as a small and independent piece of information.

Dubé and Mechqrane took inspiration from the famous video game Pacman. The original Pacman character consumes pills with the intent to accumulate points. In the Dubé-Mechqrane setting, pills are small pieces of information and Pacman not only consumes but also *produces* them. It does so with the intent to transfer the information from $\eta$ and $\eta'$, plus $w$, into $H$. During the reconstruction of a large permutation, Pacman consumes all the indices of $\eta$ and $\eta'$ and all the bits of $w$ and produces all the indices of $H$, in some order. Figure 5 illustrates the process of transforming the information that is contained in $w \in 2^Q$ and $\eta$, $\eta' \in \mathcal{H}_{M/2}$ into the information that is to be contained in $H \in \mathcal{H}_M$. There are $M + Q$ pills to consume—$M/2$ for $\eta$, $M/2$ for $\eta'$, and $Q$ for $w$—and $M$ pills to produce—for $H$. Empty boxes depict pills that have been consumed or pills that have yet to be produced. Note that, since $Q$ is chosen such that $2^Q \leq \binom{M}{M/2}$, then $Q + \log(M/2)! + \log(M/2)! \leq \log M!$, which means that there is enough capacity in $H$ to accommodate for all the information that is initially contained in $w$, $\eta$, and $\eta'$.

Pacman has a memory. Its memory enlarges when it consumes a pill and its memory shrinks when it produces a pill, as suggested by the pictures in Figure 6. In particular, if Pacman consumes many indices in a row, its memory enlarges considerably. It is preferable to have Pacman alternate between consumption and production, preventing its memory to expand too much.

We say that Pacman has a *small memory*. A small memory is intended to hold a single value: an integer in the range 1, …, $\sigma$, where $\sigma \geq 1$. While the classical way of measuring the size of a memory would consider it to be $(\log \sigma)$-bits wide, we choose to consider it to have size $\sigma$ (units).

| Consumption ($\mathsf{B}_i$, $\mathsf{E}_i$, or $\mathsf{O}_i$): | | Production ($\mathsf{L}_i$): | |
|---|---|---|---|
| *Before* : | … ᗧ $b_i$ … | *Before* : | … ᗧ $\Box$ … |
| *After* : | … $\Box$ ᗧ … | *After* : | … $\iota''_i$ ᗧ … |

Fig. 6. Effect of the instructions on Pacman's memory size.

When Pacman processes (i.e. either consumes or produces) a pill, the value in Pacman's memory gets modified. But, more importantly, the *size* of Pacman's memory also gets modified. Let $\sigma$ and $\sigma'$ be the sizes of Pacman's memory before and after processing a pill, respectively. When Pacman processes a pill, it is the *range* of the pill that causes the change from $\sigma$ to $\sigma'$. The range of the pill is the number of different values the pill may take. An input bit has range 2 and a permutation index $\iota_i$ (or $\iota'_i$ or $\iota''_i$) has range $i$. When Pacman consumes a pill of range $\rho$, Pacman's new memory size is $\sigma' = \sigma \cdot \rho$. When Pacman produces a pill of range $\rho$, $\sigma' = \lceil \sigma / \rho \rceil$. We say that the consumption of a pill introduces no redundancy while the production of a pill may introduce redundancy, due to the rounding operation. Note that $\sigma'$ is the minimal size such that there exists an injective function of type $\{1, \ldots, \sigma\} \times \{1, \ldots, \rho\} \to \{1, \ldots, \sigma'\}$, in the case of consumption, or one of type $\{1, \ldots, \sigma\} \to \{1, \ldots, \sigma'\} \times \{1, \ldots, \rho\}$, in the case of production.

### *F. Pacman's Programming*

The same sequence $\mathbb{P}$ of operations for Pacman is used each time 'Enc' is invoked. Sequence $\mathbb{P}$ is called *Pacman's programming*. A programming $\mathbb{P}$ is a sequence of $2 \times M + Q$ instructions. Each of the following instructions has to appear exactly once in $\mathbb{P}$:

- $\mathsf{E}_1$, …, $\mathsf{E}_{M/2}$ (for the consumption of an index of $\eta$),
- $\mathsf{O}_1$, …, $\mathsf{O}_{M/2}$ (for the consumption of an index of $\eta'$),
- $\mathsf{B}_1$, …, $\mathsf{B}_Q$ (for the consumption of a bit of $w$), and
- $\mathsf{L}_1$, …, $\mathsf{L}_M$ (for the production of an index of $H$).

The semantics of the instructions are described in the original paper, as well as the notion of a *valid* programming [9].

## III. Improved Construction Technique

Let us point out that $C = \log \binom{M}{M/2}$ is never an integer, except for $M = 2$. So, in general, the best per-block encoding that one can devise fixes $Q$ to be $\lfloor C \rfloor$. This means that the best per-block encoding is generally suboptimal, in a global sense, because $Q < C$. There is room for improvement, if we abandon per-block encoding. In particular, it is possible to choose a fraction that lies between the rounded capacity and the real capacity of the balanced blocks, as follows: $\lfloor C \rfloor \leq$

$Q/N \leq C$. Fraction $Q/N$, if we view it as an average number of embedded input bits per block, is an improvement over the best per-block encoding. Moreover, in principle, it is possible to choose fractions that get arbitrarily close to the real capacity.

Fortunately, the existence of such a fraction is not just a mathematical consideration. In fact, rewriting the inequality on the right-hand side gives $Q \leq N \cdot C$, which suggests that one should be able to embed $Q$ bits of information inside of a group of $N$ balanced blocks. At least, it holds in terms of theoretical encoding capacity. It also holds in terms of effective encoding procedures, as we may extend the Dubé-Mechqrane technique quite simply to implement these better embedding rates. The encoding procedure now has to build balanced blocks in groups of $N$. We redefine Pacman's task so that it now converts $N$ pairs of small permutations, $\eta_1$, $\eta'_1$, …, $\eta_N$, $\eta'_N$, as well as the block $w$ of input bits, into $N$ large permutations, $H_1, \ldots, H_N$, in a way that is similar to that depicted in Figure 5. We only need to adapt the instruction set and then use the same machinery as in the Dubé-Mechqrane technique to devise programmings in this new, generalized setting. Most concepts remain identical as in the per-block setting. The instruction set is extended and programming $\mathbb{P}$ has to contain each of the following instructions exactly once:

- $\mathsf{L}_{1,1}, \ldots, \mathsf{L}_{N,M}$ (for each index of each of $H_1, \ldots, H_N$),
- $\mathsf{E}_{1,1}, \ldots, \mathsf{E}_{N,M/2}$, $\mathsf{O}_{1,1}, \ldots, \mathsf{O}_{N,M/2}$ (similarly), and
- $\mathsf{B}_1, \ldots, \mathsf{B}_Q$ (unchanged, except for the value of $Q$).

## IV. Experimental and Theoretical Results

In order to measure the redundancy $R$ that we incur in our encoders, we need to make some crucial distinctions. We separate $R$ into what we call the *inherent* redundancy $R_I$ and the *extra* redundancy $R_X$; i.e. $R = R_I + R_X$. The inherent redundancy comes from the fact that balanced blocks of size $M$ cannot carry as much as $M$ bits of information but rather only $C = \log \binom{M}{M/2}$ bits. The difference $M - C$ is the inherent redundancy $R_I$. Redundancy $R$ is the average number of redundant bits in our encoding; i.e. $R = M - Q/N$. The extra redundancy is the one that our encoding introduces on top of the inherent redundancy; i.e. $R_X = R - R_I$. Figure 7 shows results for balanced blocks of different sizes $M$ with different averages $Q/N$ of embedded bits per block. For each selection of the parameters $M$, $Q$, and $N$, a programming $\mathbb{P}_{M,Q,N}$ has been devised. The number indicated in the column labelled with $\sigma_{\max}$ is the maximal size of Pacman's memory that is reached during the execution of $\mathbb{P}_{M,Q,N}$. Note that the experimental results show that we can reduce $R_X$ significantly by handling not so large ($N$) groups of balanced blocks. Moreover, note that the values of $\sigma_{\max}$ guarantee that conventional CPUs with 32-bit registers are more than adequate to run implementations of our technique. This is the case, despite the fact that the programmings that we devised are not especially well optimized, in terms of $\sigma_{\max}$.

| $M$ | $C$ | $R_I$ | $Q/N$ | $R_X$ | $\sigma_{\max}$ |
|---|---|---|---|---|---|
| 4 | 2.585 | 1.415 | 2/1 | .585 | 2 |
| | | | 5/2 | .0850 | 8 |
| | | | 18/7 | .0135 | 80 |
| 8 | 6.129 | 1.871 | 6/1 | .129 | 24 |
| | | | 49/8 | .00428 | 1,336 |
| 16 | 13.652 | 2.348 | 13/1 | .652 | 32 |
| | | | 27/2 | .152 | 64 |
| | | | 68/5 | .0517 | 464 |
| | | | 109/8 | .0267 | 1,264 |
| | | | 150/11 | .0154 | 2,048 |
| 32 | 29.163 | 2.837 | 29/1 | .163 | 288 |
| | | | 204/7 | .0201 | 5,312 |
| 64 | 60.669 | 3.331 | 60/1 | .669 | 320 |
| | | | 121/2 | .169 | 1,164 |
| | | | 182/3 | .00195 | 142,928 |
| 128 | 124.194 | 3.806 | 124/1 | .194 | 4,352 |
| | | | 745/6 | .0276 | 257,440 |
| 256 | 251.673 | 4.327 | 251/1 | .673 | 5,632 |
| | | | 503/2 | .173 | 24,064 |
| | | | 755/3 | .00618 | 619,744 |
| 512 | 507.174 | 4.826 | 507/1 | .174 | 78,816 |
| | | | 3043/6 | .00688 | 2,604,992 |
| 1,024 | 1,018.674 | 5.326 | 1018/1 | .674 | 74,592 |
| | | | 2037/2 | .174 | 330,752 |
| | | | 3056/3 | .00723 | 8,071,168 |

Fig. 7. Balanced blocks along with various numbers of embedded bits.

## V. Future Work

As far as we know, devising a programming is NP-hard. A programming $\mathbb{P}$ is better than another one, $\mathbb{P}'$, if $\mathbb{P}$ allows Pacman to execute with a maximal memory size $\sigma_{\max}$ that is smaller than that, $\sigma'_{\max}$, incurred by $\mathbb{P}'$. Devising a good programming is a costly process; a fast procedure would be desirable. Some theoretical foundations that underpin the technique need to be made explicit and demonstrated; e.g., the proof that valid programmings necessarily exist.


## Acknowledgment

The authors wish to thank the reviewers for their valuable comments.


## References


[1] S. J. Piestrak, "Design of self-testing checkers for unidirectional error detecting codes," *Scientific Papers of the Institute of Technical Cybernetics of the Technical University of Wroclaw*, 1995.

[2] K. A. S. Immink, "Coding techniques for the noisy magnetic recording channel: A state-of-the-art report," *IEEE Trans. on Communications*, vol. 37, no. 5, pp. 413–419, 1989.

[3] J. F. Tabor, "Noise reduction using low weight and constant weight coding techniques," CS and AI Lab, MIT, Tech. Rep. AITR-1232, May 1990.

[4] T. Cover, "Enumerative source encoding," *IEEE Trans. on Information Theory*, vol. 19, no. 1, pp. 73–77, 1973.

[5] D. E. Knuth, "Efficient balanced codes," *IEEE Trans. on Information Theory*, vol. 32, no. 1, pp. 51–53, 1986.

[6] K. A. S. Immink and J. H. Weber, "Very efficient balanced codes," *IEEE Journal on Selected Areas in Communications*, vol. 28, no. 2, pp. 188–192, 2010.

[7] J. H. Weber and K. A. S. Immink, "Knuth's balanced codes revisited," *IEEE Trans. on Information Theory*, vol. 56, no. 4, pp. 1673–1679, 2010.

[8] A. Al-Rababa'a, D. Dubé, and J.-Y. Chouinard, "Using bit recycling to reduce Knuth's balanced codes redundancy," in *Canadian Workshop on Information Theory*, 2013, pp. 6–11.

[9] D. Dubé and M. Mechqrane, "Almost minimum-redundancy construction of balanced codes using limited-precision integers," in *Canadian Workshop on Information Theory*, Quebec City, Canada, June 2017.