Constructing Optimal Whole-Bit Recycling Codes[1]

Danny Dubé
Département d'informatique et de génie logiciel
Université Laval, Canada
Email: Danny.Dube@ift.ulaval.ca

Vincent Beaudoin
Département d'informatique et de génie logiciel
Université Laval, Canada
Email: Vincent.Beaudoin.1@ulaval.ca

***Abstract*—Bit recycling aims at improving the rates achieved by compression techniques, such as LZ77, that suffer from the redundancy caused by the multiplicity of the encodings. The performance of bit recycling depends crucially on the recycling codes that it uses. A recipe for the construction of optimal recycling codes has been mentioned in previous work. However, no efficient algorithm and proof of optimality were given. We present both here.**

I. INTRODUCTION

Bit recycling aims at improving the rates achieved by compression techniques, such as LZ77 [14], that suffer from the redundancy caused by the multiplicity of the encodings [4], [6], [13]. Bit recycling takes the pragmatic approach of detecting and exploiting the multiplicity in order to recover a compensation instead of trying to eliminate the multiplicity at the source (e.g., Kawabata's approach for LZ77 [8]). The performance of bit recycling depends crucially on the recycling codes that it uses. For instance, flat recycling and proportional recycling have been presented in previous papers. However, neither bit recycling technique is optimal, even if proportional recycling comes pretty close. A recipe for the construction of optimal recycling codes has been mentioned in previous work. Still, no efficient algorithm was given. Neither was any proof of optimality. We present both in this paper.

The paper is divided as follows. Sections II, III, and IV review the notion of multiplicity of the encodings, the ability to embed data in a compressed file when multiple encodings exist, and bit recycling itself. Sections V and VI progressively introduce the concepts needed to build recycling codes that lead to optimal bit recycling and present an algorithm to construct optimal recycling codes. Section VII contains the proof that the codes built by the algorithm are indeed optimal. The paper ends with a discussion in Section VIII.

II. MULTIPLICITY OF THE ENCODINGS

LZ77 [14] is a lossless data compression technique that compresses a file, say F, by transmitting a description of F in the form of a *sequence* of *messages*. A message is either a *literal*, denoted by $[z]$, which explicitly indicates that the next character is z, or a *match*, denoted by $\langle l, d\rangle$, which indirectly indicates that the next l characters are a copy of the l characters that appear d characters before in F.[2] Variable M ranges over messages of either kind. Also, we use $\mathcal{C}$ and $\mathcal{D}$ to refer to the compressor and the decompressor, respectively. When transmitted from $\mathcal{C}$ to $\mathcal{D}$, the messages are encoded using a prefix-free code c that $\mathcal{C}$ and $\mathcal{D}$ have agreed upon.

The LZ77 compression technique does not completely specifies the way files are compressed in the sense that an original file F may have many corresponding compressed files $\mathsf{G}_1, \ldots, \mathsf{G}_n$ and any of these decompresses to F. This multiplicity of the compressed files contributes to fill up the space of compressed files too quickly. A compression technique that allows multiple encodings for most of the original files suffers from what we call the *redundancy from the multiplicity of the encodings*. We also use the term *parse* to refer to a particular sequence of messages for an original file. LZ77 compression provides multiple encodings for a file F because it can parse F in multiple ways.

In this paper, we only consider parses in which longest matches are selected in a greedy fashion. Although parsing issues are essentially orthogonal to bit-recycling code construction issues, we choose to consider greedily selected longest-match parses only, for reasons of simplicity. **Example 1.** In Figure 1, although file F_1^{ex} could be parsed in 6 different ways, there exist two ways to parse it using greedily selected longest matches.

III. DATA EMBEDDING VIA MATCH SELECTION

Many authors have observed that the multiplicity of the encodings offers an opportunity to open a side-channel of communication from $\mathcal{C}$ to $\mathcal{D}$. Let us give an overview of how this is possible. When facing multiple longest matches $\langle l, d_1\rangle, \ldots, \langle l, d_n\rangle$, $\mathcal{C}$ may choose any $\langle l, d_i\rangle$ and still describe the same l characters of F. For instance, in Example 1, the last message selected by $\mathcal{C}$ may be $\langle 3, 4\rangle$ or $\langle 3, 8\rangle$; in either case, '`abc`' gets described. When the longest matches that $\mathcal{C}$ is facing are $\langle l, d_1\rangle, \ldots, \langle l, d_n\rangle$, we say that $\mathcal{C}$ has n *options*. In a conventional implementation of LZ77, $\mathcal{D}$ does not pay attention to the particular $\langle l, d_i\rangle$ that is chosen. However, in a *non-conventional* implementation of LZ77, $\mathcal{D}$ can be brought to notice it. In such an implementation, $\mathcal{C}$ is programmed to make *eye winks* when facing multiple options and $\mathcal{D}$ is programmed to notice these eye winks. The eye winks are *information* and this information can take the form of bit sequences. For instance, in Example 1, selecting one of $\langle 3, 4\rangle$ and $\langle 3, 8\rangle$ can be associated to an eye-wink (single-) bit sequence, i.e. '`0`' or '`1`', respectively. While only the codeword for $\langle 3, d\rangle$ is *explicitly* transmitted, an additional bit is *implicitly*

[1]This work was supported by NSERC of Canada.

[2]In this paper, we require matches to be at least 3-character long.

F_1^{ex}: abc1abc2abc
Parse 1a: [a] [b] [c] [1] $\langle 3,4\rangle$ [2] $\langle 3,4\rangle$
Parse 1b: [a] [b] [c] [1] $\langle 3,4\rangle$ [2] $\langle 3,8\rangle$

Fig. 1. Greedily selected longest-match parses.

Instant I: 0110101101001100010101...
Instant II: 01100010101...
Instant III: 10101100010101...

Fig. 2. Illustration of bit recycling.

transmitted. Naturally, when more than 2 options are available, bit sequences of more than 1 bit may be used. When $\mathcal{C}$ frequently makes eye winks, a *side-channel* is established between $\mathcal{C}$ and $\mathcal{D}$. A complete transmission through the side-channel is $w = w_1 \dots w_m$, where w_i is due to the ith eye wink. This side-channel has a limited capacity which is dependent on occurrences of $\mathcal{C}$ facing multiple options. More frequent occurrences and occurrences with more numerous options increase the capacity of the side-channel.

The side-channel is general purpose and it may carry any useful data. In previous work, it has been used for information hiding, or *steganography* [2], for authentication [1], for error detection or correction [9], and for bit recycling [4], [6], [13]. This last application is described in greater detail next. It is important to notice that, in any of these uses of the side-channel, $\mathcal{C}$ is not free to arbitrarily select matches among the available options. This is because specific information needs to be transmitted through the side-channel and $\mathcal{C}$ does not decide what this information is. Since this information is made up of bit sequences sent using eye winks, each eye wink has to be made in such a way that the appropriate bits are sent through the side-channel. Consequently, each time it faces multiple matches, $\mathcal{C}$ *has* options, but it cannot select one arbitrarily. Instead, it has to select one that causes the appropriate bits to be transmitted by the eye wink.

IV. BIT RECYCLING

Bit recycling is a technique that aims at reducing the size of the compressed files. It uses the side-channel to transmit as many bits of compressed data as possible, reducing the size of the compressed file by doing so. **Example 2**. Figure 2 illustrates the workings of a bit-recycling implementation. The figure shows the changes that happen to $\mathcal{D}$'s input bit stream. At instant I, $\mathcal{D}$ is ready to decode a message. At instant II, a message M_i has been decoded. Next, $\mathcal{D}$ realizes that $\mathcal{C}$ had many options and that any of M_1, …, M_n could have been sent. At instant III, the bit sequence associated to M_i, say '101', gets *recycled*. $\mathcal{D}$ is then ready to decode the next message and possibly recycle bits again.[3]

Note that the bits that get transmitted through the side-channel are compressed data bits, which means that they are (virtually) random. It means that, in the context of bit recycling, options get selected by a random process. In contexts other than bit recycling, the embedded data need not be random in nature and so it would be doubtful to make any hypothesis about the selection probabilities of the options.

V. CONSTRUCTION OF RECYCLING CODES

The previous sections shortly describe the workings of bit recycling. However, there remains to describe a crucial part of the technique which is the way (recycled) bit sequences are associated to the options faced by $\mathcal{C}$. We adopt a progressive approach where notions are introduced one after the other and definitions get refined in multiple steps. In this section, we always presume that options M_1, …, M_n ($n > 1$) are available.

Let us give a **definition** of a recycling code. A *recycling code* for a step during compression consists in a function r that maps $\{M_i\}_{i=1}^n$ to bit sequences. Bit sequence $r(M_i)$ has to be defined, for $1 \leq i \leq n$.

Approach 1. Code r may associate fixed-length bit sequences to the options. This is one of the simplest of the conventional encodings so we might as well try it in building recycling codes. Taking into account the fact that n might not be a power of 2, we define $r(M_i)$ to be $i-1$ written in binary using k bits, where $k = \lceil \log_2 n \rceil$. **Problem**. This definition of a recycling code conflicts with the expected behavior of bit recycling. Let us present a counter-example. **Example 3**. Let $n = 3$ and r be defined as: $r(M_1) = 00$, $r(M_2) = 01$, and $r(M_3) = 10$. If we consider the operations on $\mathcal{D}$'s side, we have that, at the start of this step, the bit stream looks like $c(M_i) \cdot w$; then, $\mathcal{D}$ decodes M_i and the bit stream looks like w; finally, the bit sequence $r(M_i)$ gets recycled and the bit stream now looks like $r(M_i) \cdot w$. But what happens if the resulting bit stream needs to have the form $11 \cdot w$? **Lesson**. A recycling code has to be a complete code. **Definition**: a *complete* code $r : \Sigma \to \{0,1\}^*$ is such that, for any infinite bit sequence σ, there exists a symbol s in Σ such that $\sigma = r(s) \cdot \sigma'$. This brings us to **redefine** *recycling code*: it is a function r that maps $\{M_i\}_{i=1}^n$ to bit sequences such that $r(M_i)$ is defined, $1 \leq i \leq n$, *and* r is a complete code.

Approach 2. Code r maps $\{M_i\}_{i=1}^n$ to bit sequences of length k, where $k = \lfloor \log_2 n \rfloor$, while making sure that all k-bit sequences are in the image of r. Clearly, r is complete. **Problem**. **Example 4**. Let $n = 3$ and r be defined as $r(M_1) = r(M_2) = 0$ and $r(M_3) = 1$. When bit '0' needs to be recycled, either M_1 or M_2 may be selected. So, in such circumstances, there remain multiple encodings that are not exploited by bit recycling. **Lesson**. A recycling code ought to be a varying-length complete code.

At this point, we need of few additional **definitions**: in particular, the *net cost* of a match. Note that emitting a match M_i *costs* $|c(M_i)|$ bits; the compressed file grows. On the other hand, recycling its associated bit sequence *saves* $|r(M_i)|$ bits; the compressed file shrinks. The *net cost* of match M_i is $|c(M_i)| - |r(M_i)|$. Note that, while $r(M_i)$ is under the control of the bit recycling technique, $c(M_i)$ is not. Due to the randomness of $\mathcal{D}$'s input bit stream, option M_i's *probability* of being selected is $2^{-|r(M_i)|}$. Finally, at some step, given a recycling code r, we define the *expected cost* of the step using

[3]Note that the recycled bits could be used differently, see [13].

function BUILD($[M_1, \ldots, M_n]$):
  if $n = 1$ **then**
    return M_1
  else
    let $(\overline{M}, t') = \text{CUT}(M_{n-1}, M_n)$
    let $K(\overline{M}) = K(t')$
    let $L = \text{INSERT}([M_1, \ldots, M_{n-2}], \overline{M})$
    let $t = \text{BUILD}(L)$
    if $\overline{M}$ appears in t **then**
      return $\text{REPLACE}(t, \overline{M}, t')$
    else
      return t
function CUT(M_a, M_b):
  let $\overline{M}$ = a new "placeholder" option
  execute a branch whose condition is satisfied **in**
    1. **condition** $K(M_a) + 2 \leq K(M_b)$:
      return $(\overline{M}, M_a)$ // *drop* M_b
    2. **condition** $K(M_a) + 2 \geq K(M_b)$:
      return $(\overline{M}, M_a \otimes M_b)$ // *keep* M_b

Fig. 3. Optimal recycling code construction algorithm.

r as:

$$\sum_{i=1}^{n} 2^{-|r(M_i)|} \left(|c(M_i)| - |r(M_i)|\right).$$

In order to improve the compression as much as possible, bit recycling should build an r that minimizes the expected cost. **Lessons**. Clearly, bit recycling has to take the costs of the options into account. Moreover, it is wasteful to have a recycled bit sequence that is the prefix of another. If we go back to Example 4 and further suppose that $|c(M_1)| = |c(M_2)| = |c(M_3)|$, the expected cost would be reduced if r were rather defined as $r(M_1) = \texttt{00}$, $r(M_2) = \texttt{01}$, and $r(M_3) = \texttt{1}$.

Approach 3. In previous work, *proportional recycling* was proposed to build a good r [4]. Frequency $2^{-|c(M_i)|}$ is assigned to M_i, for $1 \leq i \leq n$, and Huffman's algorithm is used to build r [7]. **Example 5**. Let $\{M_i\}_{i=1}^{3}$ be such that $|c(M_1)| = 12$, $|c(M_2)| = 11$, and $|c(M_3)| = 15$. The assigned frequencies are 2^{-12}, 2^{-11}, and 2^{-15}, respectively, and r would likely be $r(M_1) = \texttt{00}$, $r(M_2) = \texttt{1}$, and $r(M_3) = \texttt{01}$. The expected cost of the step is $\frac{1}{4}(12-2) + \frac{1}{2}(11-1) + \frac{1}{4}(15-2) = 10.75$. **Problem**. A costly option like M_3 is not compensated enough by its associated recycled bit sequence and it pushes the expected cost up. **Lesson**. There is no need to consider *all* the available options in every step; too-costly options should be dropped. Here, we need to review two **definitions**. A *recycling code* r maps a non-empty subset of $\{M_i\}_{i=1}^{n}$ to bit sequences and it has to be a prefix-free (and complete) code. The *expected cost* of a recycling code r, $K(r)$, is:

$$\sum_{i \in \text{Dom}(r)} 2^{-|r(M_i)|} \left(|c(M_i)| - |r(M_i)|\right).$$

Approach 4. (This is the optimal approach.) It consists in selecting the "right" non-empty subset of options and in building a proportional code for them. If we come back to Example 5, optimal recycling is achieved by dropping M_3

Input: $[M_2, M_1, M_3]$ $\overline{M}$: M_α t': M_1 $K(M_\alpha) = 12$
  Input: $[M_2, M_\alpha]$ $\overline{M}$: M_β t': $M_2 \otimes M_\alpha$ $K(M_\beta) = \frac{21}{2}$
    Input: $[M_\beta]$
    Result: M_β
  Result: $M_2 \otimes M_\alpha$
  Result: $M_2 \otimes M_1$

Fig. 4. Trace of the algorithm's execution.

and building r as $r(M_1) = \texttt{0}$, $r(M_2) = \texttt{1}$, which leads to an expected cost of $\frac{1}{2}(12-1) + \frac{1}{2}(11-1) = 10.5$. From this point, there remain two tasks to be performed. First, we need to provide an efficient algorithm to build optimal recycling codes. Second, we need to prove that the algorithm effectively builds optimal recycling codes.

In the next section, we use an alternative representation of the recycling codes: one using recycling trees. A *recycling tree* t for the options $\{M_i\}_{i=1}^{n}$ is either a single leaf, M_i, or an internal node with two sub-trees, $t_1 \otimes t_2$, provided that the sets of options that appear in t_1 and t_2 are disjoint. Note that there is a one-to-one mapping between the recycling trees and the valid recycling codes. We define the *expected cost* $K(t)$ of a recycling tree t as follows:

$$\begin{aligned} K(M_i) &= |c(M_i)| \\ K(t_1 \otimes t_2) &= (K(t_1) + K(t_2))/2 - 1 \end{aligned}$$

Note that this definition is equivalent to one where the expected cost $K(t)$ of a recycling tree t would be the expected cost $K(c)$ of the recycling code c to which t corresponds.

VI. CONSTRUCTION ALGORITHM

Figure 3 presents the pseudo-code of a linear-time algorithm that builds optimal recycling codes. The main function is BUILD, which takes a list of options already ordered by cost (from the cheapest to the costliest) and returns an optimal recycling tree. $\text{INSERT}(L, M)$ inserts M in the ordered list L of options and returns an ordered list. $\text{REPLACE}(t, \overline{M}, t')$ replaces $\overline{M}$ by t' in t. Note that function CUT has a non-deterministic behavior when $K(M_a) + 2 = K(M_b)$. However, non-determinism is used only to get an algorithm that encompasses any concrete (deterministic) implementation.

The algorithm processes the options from the costliest to the cheapest. In each step, CUT decides whether the costliest one, M_n, deserves to be dropped or joined to the second costliest one, M_{n-1}. During the subsequent recursive calls, an artificial option $\overline{M}$ serves as a placeholder for M_{n-1} or $M_{n-1} \otimes M_n$, respectively. After the recursive calls, $\overline{M}$ is replaced by the sub-tree it stood for. Let us come back to Example 5. The construction of a recycling code for these options proceeds as depicted in Figure 4. Note that match M_3 is indeed dropped by the algorithm.

VII. PROOF OF OPTIMALITY

Here, we establish the optimality of the recycling trees computed by BUILD using five lemmas and a main theorem. But, first, we need a few definitions. We say that a list $[M_1, \ldots, M_n]$ of options is *ordered* if $j < i$ implies

$K(M_j) \leq K(M_i)$. The notion of *depth* of a node in a recycling tree is the usual one. A *level* in a recycling tree contains all the internal nodes and the leaves (options) that appear at a particular depth. The *lowest* level is the deepest one that is non empty.

In each of the lemmas, we let $L_{\text{in}} = [M_1, \ldots, M_n]$ be an ordered list of options and t, an optimal recycling tree for L_{in}.

Lemma 1. Let i be such that M_i appears in t. For all j such that $1 \leq j < i$, M_j also appears in t.

PROOF. Instead, let us suppose that we have i and j, with $j < i$, such that M_i appears in t but not M_j. If we replace M_i by $M_j \otimes M_i$ in t, we obtain a recycling tree for L_{in} that is strictly cheaper than t, which contradicts the optimality of t, since:

$$\begin{array}{rcl} K(M_i) & = & \left(K(M_i) + K(M_i)\right)/2 \\ & \geq & \left(K(M_j) + K(M_i)\right)/2 \\ & > & \left(K(M_j) + K(M_i)\right)/2 - 1 \\ & = & K(M_j \otimes M_i). \end{array}$$

□

Lemma 2. Let i and j such that M_i and M_j both appear in t. If M_j appears on a strictly higher level than M_i, then $K(M_j) \leq K(M_i)$.

PROOF. Instead, suppose that $K(M_j) > K(M_i)$. Let d_i and d_j be the depths of M_i and M_j in t, respectively. By hypothesis, $d_j < d_i$. Let t' be the recycling tree obtained from t by swapping M_i and M_j. Note that the bit sequence associated to every option except M_i and M_j is the same in t and t'. If we compare the expected costs of the recycling *codes* that correspond to t and t', we need only take the net costs of M_i and M_j into account and we have that:

$$\begin{array}{rl} & K(t) - K(t') \\ = & 2^{-d_i}(K(M_i) - d_i) + 2^{-d_j}(K(M_j) - d_j) \\ & -2^{-d_i}(K(M_j) - d_i) - 2^{-d_j}(K(M_i) - d_j) \\ = & 2^{-d_i}K(M_i) + 2^{-d_j}K(M_j) \\ & -2^{-d_i}K(M_j) - 2^{-d_j}K(M_i) \\ = & \left(2^{-d_j} - 2^{-d_i}\right)\left(K(M_j) - K(M_i)\right) \quad > \quad 0. \end{array}$$

This contradicts the optimality of t. □

We omit the proofs of both following, trivial lemmas.

Lemma 3. Suppose that M_i and M_j both appear in t, for $i \neq j$. If M_i and M_j appear on the same level in t, then swapping them does not change the cost of the tree.

Lemma 4. Suppose that M_i and M_j both appear in t, for $i \neq j$. If M_i and M_j have the same cost, then swapping them does not change the cost of the tree.

Lemma 5. If M_{n-1} and M_n both appear in t, then we can reshape t without changing its expected cost and have M_{n-1} and M_n be the children of the same internal node.

PROOF. The reshaping process is performed in three steps. The first step consists in making sure that M_n appears on the lowest level. If not, let us pick an option M_i that appears on the lowest level. We have that $K(M_i) \leq K(M_n)$ because L_{in} is ordered. Moreover, we have that $K(M_n) \leq K(M_i)$ because M_n appears on a strictly higher level than M_i (by Lemma 2). Since $K(M_n) = K(M_i)$, we can swap them (by Lemma 4) without changing the cost of t. The second step consists in making sure that M_{n-1} also appears on the lowest level. If not, we proceed in a similar fashion by swapping it with an option other than M_n that appears on the lowest level (by Lemmas 2 and 4). The third step consists in making sure that M_{n-1} and M_n are the children of the same internal node. If not, we swap M_{n-1} with the *brother* of M_n (by Lemma 3). □

Theorem. Given an ordered list $L_{\text{in}} = [M_1, \ldots, M_n]$ of options, BUILD returns an optimal recycling tree.

PROOF. We prove the theorem by induction on n, the number of options. In the base case, $n = 1$, there is only one option, M_1, and consequently only one possible recycling tree, M_1, which is necessarily optimal. Since BUILD returns M_1, it returns the optimal recycling tree. Now, let us consider the case where $n > 1$ and make the hypothesis that BUILD returns an optimal recycling tree when given an ordered list of options of length $n-1$. The first step that is performed by BUILD is to ask CUT to make a decision about M_{n-1} and M_n. One of two decisions is made by CUT: M_n is either dropped or kept. Let us examine each case.

First, let us consider the case where CUT decides to drop M_n. Note that $K(M_{n-1}) + 2 \leq K(M_n)$. We have that $\overline{M}$ is a placeholder for $t' = M_{n-1}$, $K(\overline{M}) = K(M_{n-1})$, L is some permutation of $[M_1, \ldots, M_{n-2}, \overline{M}]$, and t is an optimal recycling tree for L. Note that we say that L is *some* permutation of $[M_1, \ldots, M_{n-2}, \overline{M}]$ because $\overline{M}$ need not be *strictly* costlier than M_{n-2}. Note also that t is optimal for L, which has length $n-1$, by induction hypothesis. BUILD returns t'' which is identical to t except that the eventual appearance of $\overline{M}$ in t is replaced by M_{n-1}. Clearly, t'' is optimal for $[M_1, \ldots, M_{n-1}]$. We claim that t'' is also optimal for L_{in}. Now, suppose that t'' is not and let t''' be a recycling tree for L_{in} such that $K(t''') < K(t'')$. We need to consider two sub-cases: one where M_n does not appear in t''' and the other where M_n does. If M_n does not appear in t''', then we replace the eventual appearance of M_{n-1} in t''' by $\overline{M}$ and we obtain a tree for L that costs less than t. This contradicts the optimality of t for L. On the other hand, if M_n appears in t''', so does M_{n-1} (by Lemma 1). Since both M_{n-1} and M_n appear in t''' and they are (the) two costliest options, we can modify the shape of t''' without changing its cost to make them children of the same internal node (by Lemma 5). If we replace $M_{n-1} \otimes M_n$ by M_{n-1} in t''', we obtain a tree that is not costlier than t''', because

$$\begin{array}{rcl} K(M_{n-1} \otimes M_n) & = & \left(K(M_{n-1}) + K(M_n)\right)/2 - 1 \\ & \geq & \left(K(M_{n-1}) + K(M_{n-1}) + 2\right)/2 - 1 \\ & = & K(M_{n-1}), \end{array}$$

and, by replacing M_{n-1} by $\overline{M}$, we obtain a tree for L that is less costly than t. Contradiction.

Second, let us consider the case where CUT decides to keep M_n. Note that $K(M_{n-1}) + 2 \geq K(M_n)$. We have that $\overline{M}$ is a placeholder for $t' = M_{n-1} \otimes M_n$, $K(\overline{M}) = \left(K(M_{n-1}) + K(M_n)\right)/2 - 1$, L is some permutation of $[M_1, \ldots, M_{n-2}, \overline{M}]$, and t is an optimal recycling tree for L. BUILD returns t'' which is obtained by replacing the eventual appearance of $\overline{M}$ in t by $M_{n-1} \otimes M_n$. We claim that t'' is

optimal for L_{in}. Now, suppose that t'' is not and let t''' be a recycling tree for L_{in} such that $K(t''') < K(t'')$. We need to consider three sub-cases: one where M_n appears in t''', another where M_{n-1} appears in t''' but not M_n, and the last one where none of M_{n-1} and M_n appears in t'''. In the first sub-case, since M_n appears in t''', M_{n-1} appears too (by Lemma 1). We can modify the shape of t''' without changing its cost to make M_{n-1} and M_n the children of the same internal node (by Lemma 5). By replacing $M_{n-1} \otimes M_n$ by $\overline{M}$, we obtain a recycling tree for L that is as costly as t''', which contradicts the optimality of t for L. In the second sub-case, we can replace the leaf M_{n-1} by $M_{n-1} \otimes M_n$ in t''' without increasing its cost, because

$$\begin{aligned} K(M_{n-1}) &= (K(M_{n-1}) + K(M_{n-1}) + 2)/2 - 1 \\ &\geq (K(M_{n-1}) + K(M_n))/2 - 1 \\ &= K(M_{n-1} \otimes M_n), \end{aligned}$$

and the remainder of the reasoning proceeds as in the first sub-case. In the third sub-case, we immediately have that t''' is a recycling tree for L and its cost is lower than that of t, which contradicts the optimality of t for L. □

VIII. DISCUSSION AND FUTURE WORK

We presented the multiplicity of the encodings using LZ77. However, many other data compression techniques suffer from the same problem. For instance, Volf and Willems [11] present a character-wise *predict-and-encode* technique where not one but two predictors compete to best predict the next character. The encoding of each character consists in a flag that indicates which predictor has been chosen followed by the encoding of the character using that predictor's predictions. They show that, although some space is lost by sending the flags, the loss is more than compensated by the ability to let the best predictor describe each character. As is obvious, this technique systematically offers *two* encodings per character, which makes it susceptible to benefit from bit recycling. Other techniques showing the multiplicity of the encodings exist or could be naturally devised. Variants of Prediction by Partial Matching (PPM) [3] with explicit escape characters frequently offer multiple encodings for characters because it is possible to deliberately escape to a lower-order model even though a higher-level one could handle the characters. Similarly, it is possible to devise variants of dictionary-based technique, like LZ78 [15], in which the decisions to add new words in the dictionary or not are communicated to the decompressor as flags, creating multiple encodings for each emitted word.

The complexity of the algorithm that we present in this paper is linear in the number of options. This is the same complexity as Huffman's algorithm [7]. This is not surprising as both algorithms essentially perform the same sequence of operations except that ours drops options (or symbols) under certain circumstances. The difference in cost does not come from the algorithms themselves but in their uses. While Huffman's algorithm may be used once to build a code that is used for the encoding of a large number of symbols, ours has to be used each time the compressor is facing multiple options. In the context of bit-recycling LZ77 compression, it would be worthless to cache the recycling codes that our algorithm builds since the exact same set of options would rarely appear twice. Moreover, we do not observe a gradual evolution of the set of options or their costs that would make an *adaptive* variant of our algorithm useful like an adaptive Huffman's algorithm in contexts where the statistics of the symbols slowly evolve [10].

As future work, we intend to develop a fast version of our algorithm, similar to the method for fast construction of disposable prefix-free codes [5]. The method would be faster only by a constant factor, not asymptotically, and would exploit the fact that the options have integer costs. Also, we intend to make bit recycling compatible with arithmetic encoding [12]. In this paper and previous work on bit recycling, encoding messages always involves sequences of *whole* bits and recycling, too. Adapting bit recycling for arithmetic encoding would allow recycled *intervals* (instead of bit sequences) to be tailored perfectly to the probabilities at hand, without the constraint of using powers of two. The flexibility offered by the arithmetic encoding setting would make the abandon of costly options unnecessary. However, a difficulty is that a naïve scheme would rely on the use of arbitrary-precision numbers. Indeed, practical implementations of arithmetic encoding use fixed-precision integers only. Arithmetic recycling would have to do the same.

REFERENCES

[1] M. J. Atallah and S. Lonardi. Augmenting LZ-77 with authentication and integrity assurance capabilities. *Concurrency and Computation: Practice and Experience*, 16(11):1063–1076, 2004.

[2] A. Brown. `gzip-steg`, 1994.

[3] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. on Communications*, 32(4):396–402, 1984.

[4] D. Dubé and V. Beaudoin. Improving LZ77 data compression using bit recycling. In *Proc. of ISITA*, Seoul, South Korea, October 2006.

[5] D. Dubé and V. Beaudoin. Fast construction of disposable prefix-free codes. In *Proc. of the International Colloquium on Signal Processing and its Applications*, Kuala Lumpur, Malaysia, March 2008.

[6] D. Dubé and V. Beaudoin. Improving LZ77 bit recycling using all matches. In *Proc. of ISIT*, Toronto, ON, Canada, July 2008.

[7] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proc. of the Institute of Radio Engineers*, volume 40, pages 1098–1101, September 1952.

[8] T. Kawabata. Enumerative implementation of Lempel-Ziv 77 algorithm. In *Proc. of ISIT*, pages 990–994, Toronto, ON, Canada, July 2008.

[9] S. Lonardi, W. Szpankowski, and M. Ward. Error resilient LZ'77 data compression: Algorithms, analysis, and experiments. *IEEE Trans. on Information Theory*, 53(5):1799–1813, 2007.

[10] J. S. Vitter. Design and analysis of dynamic Huffman codes. *Journal of the ACM*, 34(4):825–845, October 1987.

[11] P. A. J. Volf and F. M. J. Willems. Switching between two universal source algorithms. In *Proc. of DCC*, pages 491–500, March 1998.

[12] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, June 1987.

[13] H. Yokoo. Lossless data compression and lossless data embedding. In *Proc. of the Asia-Europe Workshop on Concepts in Information Theory*, Jeju, South Korea, October 2006.

[14] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. on Information Theory*, 23(3):337–342, 1977.

[15] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. on Information Theory*, 24(5):530–536, September 1978.