# Fast Construction of Disposable Prefix-Free Codes

Danny Dubé
Université Laval, Canada
Email: Danny.Dube@ift.ulaval.ca

Vincent Beaudoin
Université Laval, Canada
Email: Vincent.Beaudoin.1@ulaval.ca

***Abstract*— Some data compression techniques use large numbers of prefix-free codes. The following two techniques do so: adaptive Huffman encoding and bit recycling. Adaptive Huffman encoding allows successive symbols to be encoded where each one is encoded according to the statistics of the symbols seen so far. Bit recycling, on the other hand, is a technique that is designed to improve the efficiency of a certain class of compression techniques (that is, the ones that allow for the existence of multiple encodings of the same data) and that repetitively has to build prefix-free codes that are used to encode or decode only one symbol. In the case of adaptive Huffman encoding, the simple but inefficient solution consists in building a prefix-free code from scratch according to the current statistics (using, say, Huffman's algorithm) before encoding each symbol. However, there exist efficient algorithms for adaptive encoding that take advantage of the fact that the statistics evolve only progressively (e.g., Vitter's algorithm). Bit recycling, on the other hand, is unlikely to reuse the same, or even a similar, prefix-free code. Consequently, a lot of prefix-free codes need to be constructed from scratch. What we propose is to use a fast technique to construct prefix-free codes. The technique trades speed in exchange of the optimality of the prefix-free codes it builds. We measured that the technique is 3 to 4 times faster than Huffman's algorithm, while the encodings of the symbols are only 4% or 1.4% longer on average, depending on whether the technique is used in a general context or in a bit-recycling one, respectively.**

## I. Introduction

It is very common for data compression techniques to construct one or a few prefix-free codes that are then used to encode and decode symbols. But less common are the data compression techniques that construct a large number of prefix-free codes and that use most of them only to encode or decode a single symbol. We will refer to prefix-free codes that are built and then used to encode or decode a single (or a few) symbol as *disposable*.

The main reason behind the need for disposable prefix-free codes in a data compression application is the situation where the statistics of the symbols change constantly. An application that requires symbols to be encoded and decoded in an optimal or close-to-optimal manner forces the prefix-free codes to be modified accordingly. A different, but somehow related, reason is when the alphabet from which the symbols are drawn changes constantly. Note that we can choose to view this second reason as a special case of the first if we define a universal alphabet which is the set of all the symbols that could ever be manipulated by the application and adapt the statistics (or the probabilities) of the symbols in order to control the possibility of occurrence of each symbol. In other words, if we denote by $U$ the universal alphabet and by $\Sigma$ the alphabet of currently possible symbols, then it is sufficient to declare each *impossible* symbol as being perfectly *improbable*, i.e. if $a \in U - \Sigma$, then $P(a) = 0$. In Section II, we present applications in which the statistics of the symbols change constantly.

In this work, we present a technique that is well-suited for data compression applications that work with disposable prefix-free codes. The technique offers the means to build a new code and to encode and decode a symbol using such a code. Note that a code built using our technique is *not* limited to the processing of a single symbol. The particularity of the technique is that of being fast when disposable codes are used. The technique features especially fast construction of codes at the cost of building sub-optimal codes, in general. Thus, in a context where a code would be used to encode and decode a lot of symbols, our technique would *not* be appropriate, due to its sub-optimality. Still, it would provide correct encoding and decoding operations. In Section IV, we show that the technique is very fast when it comes to the construction of codes and that the loss in coding efficiency is relatively limited. Moreover, all the operations that it provides have an optimal time complexity.

In the rest of the paper, we use the following vocabulary. We usually refer to prefix-free codes simply as *codes*. We call *symbols* the events, numbers, flags, or whatever pieces of information that the compressor needs to communicate to the decompressor. The symbols are encoded be the compressor and then decoded by the decompressor using some code. The set of symbols that may be transmitted at any given point or in general is called the *alphabet*. A *codeword* is a sequence of bits and it is the encoded version of a symbol. Thus, a code is a mapping between all the symbols of an alphabet and sequences of bits. The set of bit sequences associated to the alphabet by a code obeys the usual restrictions that make the code prefix-free.

The paper is organized as follows. Section II describes the problem that we attack in details. Section III describes the technique that we propose. Section IV presents the experiments that were conducted and the measurements that we obtained. Sections V and VI discuss about related and future work. Section VII is the conclusion.

## II. Description of the Problem

### A. Contexts in which Codes are Used

We distinguish three contexts in which codes are used. They are characterized by the way the statistics according to which they are built evolve. The first context is the one in which the statistics of the symbols do not change or they do but very rarely. The other contexts are the ones in which the statistics of the symbols change all the time or, at least, frequently. The second context is the one in which the statistics change but only progressively. The third context is the one in which the statistics of the symbols are constantly changing and so in an irregular way.

The first context occurs when, for instance, the compressor works first by gathering the statistics of the symbols, by constructing a code, by communicating the code to the decompressor, and finally by sending all the symbols encoded using the code. The Deflate compression technique [1], which is a variant of LZ77 [13] that is used in the `gzip` tool [6], proceeds in this way. Deflate processes one large block of data at a time. It *parses* the bytes in the block in order to determine how it decomposes into matches and literals. It does not emit any compressed data at this step. Based on the statistics of the matches and the literals, it builds two codes. Then, the sequence of matches and literals of the block is encoded using these two codes. In such a context, the statistics remain static for relatively long periods. Consequently, it is worthwhile to take the time to build optimal codes. It is worthwhile both for the coding efficiency and for the fact that fast encoding and fast decoding may be obtained by taking the time to set up helpful data structures. Huffman's algorithm [7] is useful in this context, since it is able to build optimal codes.

The second context occurs, for instance, in *adaptive* Huffman encoding. Adaptive Huffman encoding consists in updating the statistics of the symbols while they are transmitted. Provided that optimal encoding is mandatory, the fact that the statistics change constantly implies that the code has to be changed constantly, too. Note that some changes in the statistics do not necessarily force the code to change but, to the very least, some verification has to be performed to ascertain that the code is still optimal. If the conventional Huffman algorithm were used, it would force a reconstruction of the code each time the statistics are updated (i.e. possibly each time a symbol is encoded). Since the construction of a code is much more costly than the encoding (or decoding) of a symbol, the overall cost of using Huffman's algorithm would be prohibitive. Fortunately, it is not necessary to rebuild a code from scratch each time the statistics are updated. This is because the updates normally involve the increment of one or a few counters. This means that a code that is optimal for the old statistics could be "minimally modified" in order to become optimal for the new statistics. Vitter's algorithm [10] performs optimal adaptive Huffman encoding by efficiently updating the code each time a symbol is encoded.

The third context occurs, for instance, in a technique called *bit recycling* [2], [3], [4], [12]. Bit recycling is explained below. Bit recycling routinely has to build a new code from new statistics. Moreover, the code has to be built from scratch since the statistics are (almost) always totally different.

### B. Overview of Bit Recycling

Bit recycling is a technique that is intended to be added on top of data compression techniques that suffer from the problem of being prone to allow original data to be compressed in many different ways or, in other words, that suffer from the multiplicity of encodings. Multiple encodings for the same data causes redundancy and tends to make the compressed files longer than necessary. LZ77, for instance, suffers from the multiplicity of encodings [13]. LZ77 works by looking for matches between the very next bytes to describe and those that have already been described. A match has a length $l$ (the number of bytes it describes) and a distance $d$ (the number of bytes to the left of the current position where the matching bytes can be found) and is denoted by $\langle l, d\rangle$. In order to maximize compression, an LZ77 compressor looks for matches that are as long as possible, i.e. those that describe as many (clear-text) bytes as possible. For example, let us suppose that the very next bytes to describe are "`definition`" and that it is not possible to find a match longer than 10 bytes. Thus, a longest match has the form $\langle 10, d\rangle$. Note that there may exist more than one longest match.

A key observation is that, if the compressor has found $n$ possible longest matches, $\langle 10, d_1\rangle$, ..., $\langle 10, d_n\rangle$, it may choose any one of these and still describe "`definition`". The possibility for the compressor to choose (as it wishes) any of the matches allows it to make something like an *eye wink* to the decompressor. That is, provided the decompressor is programmed to notice the eye winks. In particular, if there exist two longest matches ($n = 2$), selecting one over the other carries one bit of information. Similarly, if $n = 4$, the selection carries two bits of information. In general, if $n = 2^k$, the selection carries $k$ bits of information. The idea of bit recycling is to send bits of compressed data using eye winks *instead* of doing so through the compressed file. Sending bits using eye winks contributes to reduce the size of the compressed files. That being said, the compressor is limited to send only as many bits using eye winks as the longest matches permit since the compressor does not have control over $n$.

The presentation of bit recycling above is a bit naïve. First, it is not necessary for $n$ to be a power of two. Second, it is important to notice that an important issue has been neglected in the presentation: the cost of using some match $\langle l, d_i\rangle$ over another. The encoding of $\langle l, d_i\rangle$ might require more bits than that of $\langle l, d_j\rangle$, for $i \neq j$. This issue is important since the benefit of using eye winks might get ruined by the occasional selection of costly matches. However, nothing forces bit recycling to associate a fixed number of eye-wink bits to each longest match. In fact, Dubé and Beaudoin [3] found experimentally that a very good way to associate eye-wink bits to matches is to build a prefix-free code using statistics derived from the costs of the encodings of the matches. This way to associate eye-wink bits is called *proportional bit recycling*. In

proportional bit recycling, a match $\langle l, d_i \rangle$, which is encoded in, say, $c_i$ bits, is assigned a "frequency" of $2^{-c_i}$. A prefix-free code is then built, based on these frequencies. The codeword that is associated to each match constitutes its sequence of eye-wink bits. The net effect is that, the more costly a match is, the more numerous the eye-wink bits tend to be.

The operations on codes that bit recycling uses are: the construction of a new code based on a set of longest matches, the conversion of a match to its associated sequence of eye-wink bits, and the inverse conversion. Note that these operations on codes are those that take care of the bits involved in the bit recycling proper. The codes used to encode and decode the matches and the literals need not have any relation with them. For instance, the codes for the matches and the literals may continue to be rebuilt on a per-block basis if this is the case in the original technique (i.e. the technique without bit recycling). So, let us focus on the operations related to bit recycling. On the compressor side, each time there exist multiple longest matches, a code has to be built to associate them to eye-wink bit sequences and the code is used once for the decoding of a match. Note that the compressor has to perform a *decoding* operation because the chosen match gets selected with the intent of transmitting certain eye-wink bits to the decompressor. On the decompressor side, when a longest match is received and when it is established that other matches could have been used instead, a code is built (the same code as that on the compressor side) and it is used once to encode the match in order to recover the eye wink made by the compressor.

Given the way the codes related to bit recycling are used, it is clear that each code is built with the intent of using it only once and it is rather infrequent for a code to be required again. Consequently, it is reasonable to abandon each of the codes after its use instead of trying to store, index, and possibly reuse them.

### C. Cost of the Proposed Operations

Our technique is intended to work for prefix-free codes and symbols in general, i.e. not only for a bit recycling technique. As said above, three operations have to be provided: the construction of a code from a set of symbols and their associated frequencies, the encoding of a symbol, and the decoding of a symbol. Essentially, we would like to have the construction of a code in time proportional to the number of symbols and the encoding and the decoding of a symbol in time proportional to the length of the codeword associated to the symbol. However, we will see that there is an additional value $\Delta$ that must be taken into account. The asymptotic complexity of the three operations grows linearly with $\Delta$ too. $\Delta$ is the logarithm of the quotient between the highest frequency and the lowest non-zero one, i.e.

$$\Delta = \log_2 f_{max} - \log_2 f_{min}$$

In the context of bit recycling, $\Delta$ has an intuitive meaning: it is the difference between the cost of the most costly match and that of the least costly one.

## III. Description of the Technique

### A. From Frequency Counts to Costs

Unlike Huffman's algorithm, our technique does not manipulate probabilities nor frequencies. Instead, it manipulates costs. A cost is an estimate of the number of bits that would be required to encode each symbol of an alphabet. A cost is a logarithmic measure in the sense that a symbol that is twice as improbable ought to be one bit higher. Our technique is insensitive to the absolute costs of the symbols; only their relative costs matter. It means that uniformly adding or subtracting the same constant to all the costs does not change the resulting code. This is similar to the insensitivity of Huffman's algorithm where multiplying or dividing all the frequencies by the same constant does not change the resulting code.

When an application intends to use our technique by providing the frequencies of the symbols, then the frequencies first have to be translated into costs. Let $\Sigma$ be the alphabet that contains all the provided symbols. Without loss of generality, we make the simplifying assumption that all the frequencies are strictly positive. If $\Sigma$ were to contain improbable symbols then we could redefine $\Sigma$ by ejecting these and keeping only the symbols with strictly positive frequencies. For each symbol $s$ of frequency $f_s$, we compute its cost $c_s$ as $\lceil -\log_2 f_s \rceil$.

Let us recall that our technique sacrifices the optimality of the built codes in order to make the construction faster. It should be clear that this very first conversion alone, when it must be performed, is sufficient to lead to sub-optimal codes. For example, suppose that there are three symbols of frequencies $1/5$, $1/6$, and $1/7$. Clearly, the optimal code consists in assigning a 1-bit codeword to the most frequent symbol and two 2-bit codewords to the other symbols. However, once the frequencies are converted into costs (costs of 3, here), then the single 1-bit codeword might be assigned to any one of the three symbols.

### B. Building a Code

Our technique builds a code for an alphabet $\Sigma = \{s_1, \ldots, s_n\}$ whose symbols are assigned costs. The cost of $s_i$ is $c_i$. The construction proceeds in a few steps. The code that is built by our technique is represented in an implicit form. This form contains enough information for a subsequent encoding or decoding operation to proceed smoothly.

The first step consists in counting the number of symbols of each cost. Let $\mathrm{count}[c]$ be the number of symbols of cost $c$. During this step, the technique takes note of the values of $c_{min}$, the minimum cost, and $c_{max}$, the maximum cost. Note that $\Delta$, presented above, is simply $c_{max} - c_{min}$. This step can be performed in time $O(n)$, provided that the array $\mathrm{count}$ contains only zeros, initially. Clearly, at the end of this step, $\mathrm{count}[c] = 0$ for $c < c_{min}$ and for $c > c_{max}$.

The second step consists in computing the shape of the binary tree that implicitly represents the code built by the technique. Before we describe the computations precisely, we intuitively explain the way the tree is built. We need to talk

about the *levels* of the tree. The level in which the deepest leaves are placed is numbered $c_{max}$. Each time we go up a level, the number of the level decreases. Ideally, the tree that is built would have $\mathrm{count}[c]$ leaves on level $c$. Each leaf on level $c$ would correspond to a symbol of cost $c$. On level $c$, there would be exactly half as many *internal nodes* as there would be *nodes* on level $c+1$. On the highest level (one with a cost no more than $c_{min}$), there would be the root of the tree. We say "ideally" because, in reality, the number of nodes on a level might not be even. In order to compute the (real) shape of the tree, the array 'carry' is used. The first entry is computed like this:

$$\mathrm{carry}[c_{max}] = 0.$$

Then, the entries of progressively lower cost are computed like this:

$$\mathrm{carry}[c] = \left\lceil \frac{\mathrm{count}[c+1] + \mathrm{carry}[c+1]}{2} \right\rceil.$$

Since $\mathrm{count}[c] = 0$ for $c < c_{min}$, then there is a cost $c_{root}$ ($c_{root} \leq c_{min}$) such that $\mathrm{carry}[c] = 1$ for $c <= c_{root}$. Since we do not want to manipulate an infinite array, what we do in practice is to progressively compute the entries of array 'carry' until we reach a cost $c \leq c_{min}$ such that $\mathrm{count}[c] + \mathrm{carry}[c] = 1$. This cost is called $c_{root}$. Intuitively, $\mathrm{carry}[c]$ is roughly the number of internal nodes at level $c$. The second step is performed in time $O(c_{max} - c_{root})$. Since $c_{root}$ is not an input of the code construction algorithm, but only an indirect consequence, we prefer to replace it by a lower bound. The minimal value that $c_{root}$ can take is $c_{min} - \lceil \log_2 n \rceil$ and it happens when (almost) all the symbols have cost $c_{min}$. Consequently, $O(c_{max} - c_{root}) \subseteq O(\Delta + \log_2 n)$ and the worst case happens when a single symbol has cost $c_{max}$ and all the others have cost $c_{min}$.

The third step consists in establishing a relation between the symbols and their positions in the tree. The array 'base' is indexed by cost and it gives the first position in the tree where symbols of at least a certain cost can be found. The entries of 'base' are computed for costs $c_{root}$ to $c_{max}$ this way:

$$\begin{aligned} \mathrm{base}[c_{root}] &= 0, \\ \mathrm{base}[c] &= \mathrm{base}[c-1] + \mathrm{count}[c-1]. \end{aligned}$$

Roughly speaking, $\mathrm{base}[c]$ counts the number of symbols that appear in the levels numbered less than $c$, in an ideal tree. The array 'position' gives the position of each symbol in the tree. Entry $\mathrm{position}[s_i]$ is defined as $\mathrm{base}[c]$ if $s_i$ is the first symbol of cost $c$ in $\Sigma$ or as $\mathrm{position}[s_j] + 1$ if $s_j$ is the closest predecessor of cost $c$ in $\Sigma$. Finally, array 'element' is the inverse of array 'position', that is:

$$\mathrm{element}[p] = s_i \text{ if and only if } \mathrm{position}[s_i] = p.$$

Filling array 'base' takes $O(\Delta + \log_2 n)$ in time and filling arrays 'position' and 'element' takes $O(n)$ in time.

If we sum up the time required to perform all three steps, we obtain $O(n + \Delta)$. Recall that the first step requires that the array 'count' is filled with zeros. In order to obey this condition for the next construction of a code, the array can be cleaned to nullify all the entries that might have been changed, i.e. entries $\mathrm{count}[c_{min}]$ to $\mathrm{count}[c_{max}]$. This cleaning can be performed in time $O(\Delta)$, which does not change the overall complexity. Note that, in practice, $\Delta$ is likely to be a pretty small constant; especially if the costs come from the lengths of the codewords, as in bit recycling, and the compression technique imposes a bound on the lengths of the codewords.

### C. Encoding and Decoding Using the Code

The encoding and decoding operations are relatively similar. Both proceed by going up or down in the tree, level by level, and by behaving according to the nodes they reach. There are three different situations. The first one is when a terminal node is reached: either the root or a leaf. The two other situations involve internal nodes. Let us consider the internal nodes on level $c$ and all the nodes on level $c+1$. If there is an even number of nodes on level $c+1$, then each internal node on level $c$ is the parent of two children. With a node that is the parent of two children comes the need to distinguish between the children. A bit (either encoded or decoded) needs to be involved in the process of distinguishing the children. On the other hand, when there is an odd number of nodes on level $c+1$, there is an internal node on level $c$ that is the parent of a single child. In the case of this internal node, there is no ambiguity in the identity of its child and no bit needs to be involved. If we come back to the three situations, then the second one applies when an internal node has a single child and the third one, when an internal node has two children. Note that we arbitrarily choose to let the first internal node of level $c$ be the parent of a single child when level $c+1$ contains an odd number of nodes.

We begin with the description of the decoding process because it manipulates bits in the usual order. The decoding process starts by pointing on the root (on level $c_{root}$). We let pointer $p$ be 0 since we start the numbering of the nodes on a level at 0. Then, the routine is the same on each level. Let $c$ be the current level. If $0 \leq p < \mathrm{count}[c]$, then the node pointed by $p$ is a leaf, i.e. it represents a symbol. The position of the symbol is $\mathrm{base}[c] + p$ and, using array 'element', the corresponding symbol can be recovered. Otherwise, the node pointed by $p$ is internal. If there is an even number of nodes on level $c+1$, then a bit $b$ has to be read, and the new value of the pointer on level $c+1$ is $2*(p - \mathrm{count}[c]) + b$. Otherwise, there is an odd number of nodes on level $c+1$. If $p = \mathrm{count}[c]$, then the new value of the pointer on level $c+1$ is 0. Otherwise, we have that $p > \mathrm{count}[c]$ and a bit $b$ has to be read and the new value of the pointer on level $c+1$ is $1+2*(p - \mathrm{count}[c] - 1) + b$.

The encoding process is more complicated only because it generates the bits in the reverse order. Let $s_i$ be the symbol to encode. The encoding process starts at level $c = c_i$ ($s_i$'s cost) and by initializing $p$ to $\mathrm{position}[s_i] - \mathrm{base}[c_i]$. Then, the routine is the same on each level. If $c = c_{root}$, then the encoding process is finished. Otherwise, the pointed node has a parent. If there is an even number of nodes on level $c$, then

| Cost | ... | 1 | 2 | 3 | 4 | 5 | 6 | ... |
|---|---|---|---|---|---|---|---|---|
| count | ... | 0 | 0 | 0 | 1 | 3 | 0 | ... |
| carry | ... | — | 1 | 2 | 2 | 0 | — | ... |
| base | ... | — | 0 | 0 | 0 | 1 | — | ... |

| | $s_1$ | $s_2$ | $s_3$ | $s_4$ |
|---|---|---|---|---|
| $c_i$ | 5 | 5 | 4 | 5 |
| position | 1 | 2 | 0 | 3 |

| Position | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| element | $s_3$ | $s_1$ | $s_2$ | $s_4$ |

Fig. 1. Arrays that form the implicit representation of a code.

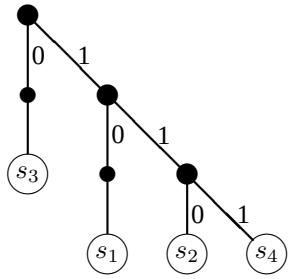


Fig. 2. Picture of the tree that is implicitly represented by the arrays.

the bit $b = p \bmod 2$ is emitted and the new value of the pointer on level $c-1$ is $\text{count}[c-1] + (p-b)/2$. Otherwise, there is an odd number of nodes on level $c$. If $p = 0$, then the new value of the pointer on level $c-1$ is $\text{count}[c-1]$. Otherwise, the bit $b = (p-1) \bmod 2$ is emitted and the new value of the pointer on level $c-1$ is $\text{count}[c-1] + 1 + (p-1-b)/2$.

Both the encoding operation and the decoding operation involve traversing the full height of the tree, in the worst case. The operations that need to be performed on each level can be done in constant time. Consequently, encoding or decoding a symbol can be done in time $O(\Delta + \log_2 n)$.

### *D. An Example*

Let us illustrate the technique with a small example. Let $\Sigma = \{s_1, s_2, s_3, s_4\}$ where the symbols have costs 5, 5, 4, and 5, respectively. Figure 1 presents the various arrays that get filled by the construction of a code for $\Sigma$. Note that $c_{min} = 4$, $c_{max} = 5$, and $c_{root} = 2$.

Suppose that we want to encode $s_1$. We have to start by converting $s_1$ into its position, which is 1. The cost of $s_1$ is 5. We let $p = \text{position}[s_1] - \text{base}[5] = 1 - 1 = 0$. Since there is an odd number of nodes on level 5 and $p$ points on the first one, then this node is the only child of its parent. Consequently, no bit gets emitted and the new $p$ on level 4 is $\text{count}[4] = 1$. Since there is an odd number of nodes on level 4 but $p$ does not point on the first node, then the parity of $p-1$ (0) has to be emitted and the new $p$ on level 3 is $\text{count}[3]+1+(1-1-0)/2 = 1$. Since there is an even number of nodes on level 3, then the parity of $p$ (1) has to be emitted and the new $p$ on level 2 is $\text{count}[2] + (1-1)/2 = 0$. Finally, level 2 has been reached and the encoding is finished. The bits that have been emitted have been so in the inverse order, so the codeword generated for $s_1$ is 10. Note that decoding the codeword would involve similar decision making.

Figure 2 illustrates the shape of the tree that is implicitly represented by the arrays filled by the code construction algorithm. Each symbol appears at the level that corresponds to its cost. Except for the root, every node has a parent. When a level contains an odd number of nodes, the first internal node of the upper level has only one child. A bit needs to be emitted (received) when one of the arms of a parent with two children is traversed.

## IV. Experimental Results

In the experiments, we compared two techniques for the construction of prefix-free codes: Huffman's algorithm and our technique. The implementation of Huffman's algorithm that we used comes from `gzip` [6]. More precisely, it is the one in the implementation of the Deflate technique [1]. This implementation is coded for speed. However, it performs additional operations like trimming the codewords to make sure they fit in 15 bits and converting the code to make it canonical. We removed these unnecessary operations and kept only the heap-based implementation of Huffman's algorithm. We added child fields in the nodes of the tree that it builds so that the tree could be traversed downwards as much as upwards and so, as soon as it is built.

We used two kinds of benchmarks: some where the symbols are associated with frequencies and the others where the symbols are associated with costs. Both kinds of benchmarks originate from the processing of a file using a prototype by Dubé and Beaudoin, which is a version of `gzip` augmented with bit recycling. The frequencies (of the benchmarks with frequencies) are taken from instances of codes that `gzip` has to generate in order to send matches, literals, and canonical codes to the decompressor. The costs (of the benchmarks with costs) are taken from instances of codes that are used by the bit recycling proper. These costs are the lengths of the encodings of the set of longest matches when there are many of these. The file whose processing generated all the benchmarks is `book1`, from the Calgary Corpus [11]. There were 18 benchmarks with frequencies and 59 046 benchmarks with costs that were generated by the processing of `book1`. These numbers demonstrate how frequent the construction of codes becomes when bit recycling is used. Fast construction of codes is truly needed in such an application.

We had two techniques for code construction and two sets of benchmarks. We tried the four combinations. Both techniques were very quick. So, in any combination, we had to run the technique on the set of benchmarks many times. When the benchmarks with frequencies were used, we had codes built for the whole set 100 000 times. When the benchmarks with costs were used, we had codes built for the whole set 1000 times. The times that we measured for the four combinations are

| Kind of bench. | # of instances | # of iterations | Time w/ Huffman | Time w/ ours |
|---|---|---|---|---|
| With freq. | 18 | 100 000 | 11.337 s | 3.676 s |
| With costs | 59 046 | 1000 | 42.747 s | 11.881 s |

Fig. 3. Execution times for the repetitive construction of codes for all benchmarks of each kind.

presented in Figure 3. We can see the our technique is 3 to 4 times faster than Huffman's algorithm. Note that there is a conversion between frequencies and costs when our technique is used on the benchmarks with frequencies and the same when Huffman's algorithm is used on the benchmarks with costs. We also measured the increase in the expected length of the codewords caused by the sub-optimality of our technique. On the benchmarks with frequencies, we observed an increase of about 4% while on the benchmarks with costs, we observed an increase of about 1.4%. The latter measurements is particularly satisfying given that our technique performs the best (from the coding efficiency point of view) when its speed is most needed, i.e. in the context of bit recycling.

## V. Related Work

We compared our technique to the best-known code construction technique: Huffman's algorithm [7]. However, there exist other techniques that are likely to be fast or even that have been designed with that intent. Namely, there is the encoding of Shannon-Fano [9] and the Fyffe codes [5]. In order to have a more complete comparison, the speed and, if applicable, the loss in coding efficiency of these techniques ought to be measured.

The particular case in which the statistics change constantly but in a progressive has been studied on many occasions [8], [10].

## VI. Future Work

It would be interesting to try to develop a variant of our technique that could accommodate finer variations in the costs of the symbols than whole bits. Arguably, the conversion process from frequencies to costs in the current work may be responsible for a significant part of the loss in coding efficiency that we observed.

We did not take the time to explain bit recycling in more details in this paper but it has an important effect on the way matches get selected and how the coding efficiency must be evaluated. We refer the reader to the papers on bit recycling [2], [3], [4], [12]. It would be interesting to look for a technique that optimizes the coding efficiency in the context of bit recycling, while remaining very fast.

## VII. Conclusion

We have presented a fast technique for the construction of disposable codes. In order to achieve its goal, it tries to make the construction of the codes as lightweight a process as possible, even if encoding and decoding symbols becomes heavier. We measured the performances of our technique compared to a very fast implementation of Huffman's algorithm. Our technique is 3 to 4 times faster than Huffman's algorithm. However, it does not necessarily generate optimal codes. In a general context, we measured that the average length of the codewords gets increased by about 4%. However, in the context of bit recycling, the increase reduces to about 1.4%, which is rather satisfying since it is precisely the bit recycling techniques that need to build numerous disposable codes.

## References


[1] P. Deutsch. Request for comments: 1051, 1996. `http://www.ietf.org/rfc/rfc1051.txt`.

[2] D. Dubé and V. Beaudoin. Recycling bits in LZ77-based compression. In *Proceedings of the Conférence des Sciences Électroniques, Technologies de l'Information et des Télécommunications (SETIT 2005)*, Sousse, Tunisia, mar 2005.

[3] D. Dubé and V. Beaudoin. Improving LZ77 data compression using bit recycling. In *Proceedings of the International Symposium on Information Theory and Applications (ISITA)*, Seoul, South Korea, oct 2006.

[4] D. Dubé and V. Beaudoin. Bit recycling with prefix codes. In *Proc. of DCC*, page 379, Snowbird, Utah, USA, mar 2007. Poster.

[5] Graham Fyffe. Fyffe codes for fast codelength approximation, 1999. `http://www.geocities.com/g_fyffe/fastselect.htm`.

[6] J. L. Gailly and M. Adler. The GZIP compressor. `http://www.gzip.org`.

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

[8] D. E. Knuth. Dynamic Huffman coding. *Journal of Algorithms*, 6:163–180, 1985.

[9] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27, 1948.

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

[11] I. Witten, T. Bell, and J. Cleary. The Calgary corpus, 1987. `ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus`.

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

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