# Optimizing the Free Distance of Error-Correcting Variable-Length Codes

Amadou Diallo, Claudio Weidmann, Michel Kieffer

# Optimizing the Free Distance of Error-Correcting Variable-Length Codes

Amadou Diallo [1], Claudio Weidmann [2], Michel Kieffer [1,3]

[1] *L2S - CNRS - SUPELEC - Univ Paris-Sud*
*3 rue Joliot-Curie, 91192 Gif-sur-Yvette cedex, France*
amadou.diallo@lss.supelec.fr
kieffer@lss.supelec.fr

[2] *INTHFT, Vienna University of Technology, 1040 Vienna, Austria.*
claudio.weidmann@ieee.org

[3] *on sabbatical leave at LTCI - CNRS - Telecom ParisTech 75013 Paris, France.*

***Abstract*—This paper considers the optimization of Error-Correcting Variable-Length Codes (EC-VLC), which are a class of joint-source channel codes. The aim is to find a prefix-free codebook with the largest possible free distance for a given set of codeword lengths, $\boldsymbol{\ell} = (\ell_1, \ell_2, \ldots, \ell_M)$. The proposed approach consists in ordering all possible codebooks associated to $\boldsymbol{\ell}$ on a tree, and then to apply an efficient branch-and-prune algorithm to find a codebook with maximal free distance. Three methods for building the tree of codebooks are presented and their efficiency is compared.**

## I. Introduction

The transmission of multimedia contents over an error-prone channel with scarce bandwidth usually requires lossless or lossy compression of the content to remove redundancy. The introduction of structured redundancy via a channel code is also required to improve the robustness of the compressed stream to transmission errors, which are unavoidable when considering transmission over a wireless channel. This scheme is well known as *separated (tandem) scheme* and is motivated by Shannon's separation principle [1], [2], which states that source and channel coding may be optimized separately, without loss in optimality compared to a joint design. Nevertheless, this result has been obtained under the hypothesis of a stationary channel, with well-known characteristics at the transmitter and receiver, which is seldom the case in wireless communication systems.

These limitations have motivated the development of joint source-channel (JSC) coding techniques, which aim at designing low-complexity codes simultaneously providing data compression and error correction capabilities. The hope is to get joint codes outperforming separate codes when the length of the codes is constrained, see [3].

Compression efficiency is measured by the ratio of the average code length to the source entropy [2], while the error-correction performance may be predicted with an union bound using the *distance properties* of the code, *i.e.*, its *free distance* and *distance spectrum*, see [4].

JSC coding using error-correcting arithmetic coding (EC-AC) was introduced in [5]. First techniques for optimizing the free distance of EC-AC are presented in [6], whereas a global optimization of the free distance using a tree data structure and an efficient branch-and-prune algorithm is reported in [7]. The present work aims to apply the ideas of [7] to the optimization of error-correcting variable-length codes (EC-VLC).

Early work on JSC coding using EC-VLCs includes [8]. In [9], two methods of constructing EC-VLCs with a desired free distance are proposed, namely the "code anti-code" construction and a heuristic construction algorithm. The main problem of these methods is that the obtained codeword lengths are not matched to the source statistics. In addition, if the smallest codeword length is less than the desired free distance, then no EC-VLC is found.

In this paper, we propose an alternative approach for building EC-VLCs with large free distance. For a given set of codeword lengths, $\boldsymbol{\ell} = (\ell_1, \ell_2, \ldots, \ell_M)$, which satisfies Kraft's inequality $\sum_{i=1}^{M} 2^{-\ell_1} \leq 1$, and where we assume w.l.o.g. $\ell_i \leq \ell_{i+1}$, the proposed algorithms aim to design a prefix-free codebook with maximal free distance.

Our approach consists in ordering all possible codebooks associated to $\boldsymbol{\ell}$ in a tree such that leaves correspond to EC-VLC codebooks, and (internal) parent nodes correspond to partially defined codebooks, from which child nodes may be obtained by specific rules. By construction, the free distance of a parent node will be an upper bound on the free distances of its child nodes, so that we may apply an efficient *branch-and-prune* algorithm to explore only a part of the tree, thus reducing the time needed to find the best EC-VLC [7]. Three methods to structure the search tree are proposed. The first method, *construction by codewords*, adds one codeword at time to a parent node to obtain a child node. It is described in Section III-B. The second method, *construction by bit planes*, is described in Section III-C. It successively determines bitplanes, *i.e.*, chooses the first bits of all codewords, then the second bits, and so on. The third method separates the

*MMSP'10, October 4-6, 2010, Saint-Malo, France. 978-1-4244-8112-5/10/$26.00* 

structure of a prefix-free code tree from the actual labeling with 0 and 1, by first enumerating canonical trees representing tree isomorphism classes, see Section III-D.

Before detailing these construction methods, Section II details some properties of EC-VLCs and recalls the basis of the tools used to compute the free distance. Experimental results are provided in Section IV, before drawing some conclusions.

## II. Computing the free distance of an EC-VLC

In this section, we briefly recall EC-VLCs and show how their free distance can be computed. Consider a memoryless source $X$ with alphabet $\mathcal{X} = \{a_1, a_2, \ldots, a_M\}$ and associated probabilities $\boldsymbol{p} = (p_1, p_2, \ldots, p_M)$. To each symbol $a_i$ in $\mathcal{X}$, one associates a codeword $c_i$ in a set of codewords $\mathcal{C} = \{c_1, c_2, \ldots, c_M\}$. The length in bits of $c_i$ is $\ell_i$, $i = 1, \ldots, M$. The codebook $\mathcal{C}$ is prefix-free iff the codeword lengths $\boldsymbol{\ell} = (\ell_1, \ell_2, \ldots, \ell_M)$ satisfy Kraft's inequality

$$\sum_{i=1}^{M} 2^{-\ell_1} \leq 1, \tag{1}$$

see [2]. Henceforth we assume (1) is satisfied and call $\boldsymbol{\ell}$ a *Kraft vector*.

The performance of an EC-VLC is determined by its *redundancy* and its *error correcting capability*. The redundancy $R_{\mathrm{c}}$ is the difference between the average codeword length $\ell_{\mathrm{av}} = \sum_{i=1}^{M} p_i \ell_i$ and the source entropy $H_{\mathrm{c}} = -\sum_{i=1}^{M} p_i \log_2 p_i$. Thus

$$R_{\mathrm{c}} = \ell_{\mathrm{av}} - H_{\mathrm{c}} = \sum_{i=1}^{M} p_i \ell_i + \sum_{i=1}^{M} p_i \log_2 p_i. \tag{2}$$

The error correcting capability is primarily characterized by the *free distance* $d_{\mathrm{free}}$ (a finer characterization is possible through the *distance spectrum*). To evaluate the distance properties of an EC-VLC, a graphical representation of the code is better suited than a list of codewords. The code $\mathcal{C}$ can be represented as a directed graph $\Gamma(\mathcal{S}, \mathcal{T})$, where $\mathcal{S}$ is a set of states (vertices) and $\mathcal{T}$ is a set of transitions (directed edges). Each transition is labeled with an input symbol in $\mathcal{X}$ and a sequence of output bits. $\Gamma$ is also called a *finite-state encoder* (FSE). In the simple case of an EC-VLC, $\mathcal{S}$ contains a single state $s_0$, from/to which all transitions leave/lead, so $\mathcal{T}$ has $M$ transitions associated to the elements of $\mathcal{X}$. Each transition $u_i$ has an input label $I(u_i) = a_i$, an output label $O(u_i) = c_i$ and an associated probability $P(u_i) = p_i$. Hence

$$\mathcal{T} = \{u_i = a_i/c_i \ : \ 1 \leqslant i \leqslant M\}. \tag{3}$$

Fig. 1(a) shows the FSE associated to a source $X$ with alphabet $\mathcal{X}_3 = \{a_1 = a, a_2 = b, a_3 = c\}$ encoded using the codebook $\mathcal{C}_3 = \{c_1 = 0, c_2 = 10, c_3 = 111\}$.

A better-suited representation of $\mathcal{C}$ for distance evaluation is the bit-clock FSE (B-FSE) in which each transition is labeled with exactly one output bit and may have an empty input label. Details on how B-FSE can be obtained from the FSE are presented in [6]. Fig. 1(b) shows the B-FSE derived from the FSE of Fig. 1(a).

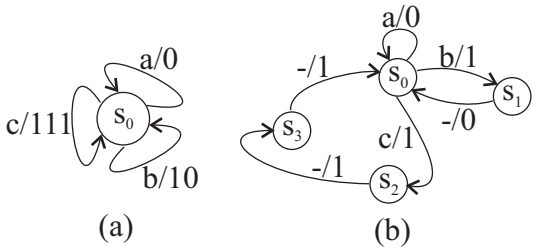


Fig. 1. (a) Example of FSE associated to $\mathcal{X}_3$ and $\mathcal{C}_3$ and (b) its corresponding bit-clock representation

Let $\sigma(u)$ be the originating state of some transition $u \in \mathcal{T}$ and $\tau(u)$ its target state. A path $\boldsymbol{u} = (u_1 \circ u_2 \circ \cdots \circ u_k) \in \mathcal{T}^k$ on the graph is a concatenation of transitions that satisfy $\sigma(u_{i+1}) = \tau(u_i)$ for $1 \leqslant i < k$ (this corresponds to a *walk* of length $k$ on the encoder graph). By extension, we define $\sigma(\boldsymbol{u}) = \sigma(u_1)$ and $\tau(\boldsymbol{u}) = \tau(u_k)$, as well as $I(\boldsymbol{u})$ and $O(\boldsymbol{u})$, which are the concatenations of the input, respectively output, labels of $\boldsymbol{u}$. The probability of a path is $P(\boldsymbol{u}) = \prod_{i=1}^{k} P(u_i)$. Finally, $\ell(\boldsymbol{x})$ is the length (in symbols or bits) of the sequence $\boldsymbol{x}$.

The Hamming distance $d_{\mathrm{H}}$ between two equal-length sequences $\boldsymbol{x}, \boldsymbol{y}$ is equal to the Hamming weight $w_{\mathrm{H}}$, *i.e.*, the number of non-zero entries, of their elementwise difference, $d_{\mathrm{H}}(\boldsymbol{x}, \boldsymbol{y}) = w_{\mathrm{H}}(\boldsymbol{x} - \boldsymbol{y})$. If two paths $(\boldsymbol{u}_1, \boldsymbol{u}_2) \in \mathcal{T}^{k_1} \times \mathcal{T}^{k_2}$ are such that $\ell(O(\boldsymbol{u}_1)) = \ell(O(\boldsymbol{u}_2))$, then we will write $d_{\mathrm{H}}(\boldsymbol{u}_1, \boldsymbol{u}_2) = d_{\mathrm{H}}(O(\boldsymbol{u}_1), O(\boldsymbol{u}_2))$.

*Definition 1:* Let $\mathcal{P}$ be the set of all pairs of paths in $\left(\mathcal{T}^{k_1} \times \mathcal{T}^{k_2}\right)_{1 \leqslant k_1, k_2 < \infty}$ diverging from $s_0$ and converging for the first time in $s_0$ with the same length of output labels. Then $d_{\mathrm{free}}$ is the minimum Hamming distance in $\mathcal{P}$,

$$d_{\mathrm{free}} = \min_{(\boldsymbol{u}_1, \boldsymbol{u}_2) \in \mathcal{P}} d_{\mathrm{H}}(\boldsymbol{u}_1, \boldsymbol{u}_2). \tag{4}$$

*Definition 2:* The distance spectrum [10] in the code domain can be represented with a generating function

$$G(D) = \sum_{d=d_{\mathrm{free}}}^{\infty} A_d D^d, \tag{5}$$

where $A_d$ is the average number of pairs of paths in $\mathcal{P}$ with Hamming distance $d$. In [9], $A_d$ is defined as :

$$A_d = \sum_{\substack{(\boldsymbol{u}_1, \boldsymbol{u}_2) \in \mathcal{P} \\ d_{\mathrm{H}}(\boldsymbol{u}_1, \boldsymbol{u}_2) = d}} P(\boldsymbol{u}_1) \tag{6}$$

In recent work [11], we introduced a *Pairwise Distance Graph* (PDG), which is a modified and reduced *product graph* of the B-FSE and tracks the Hamming distances in $\mathcal{P}$. This PDG is defined such that the free distance can be found by applying Dijkstra's algorithm [12].

The PDG is obtained as follows. Let $\mathcal{S}_b = \{s_i : \ 0 \leqslant i < M_b\}$ be the set of states of the B-FSE. The product graph associated to $\Gamma_b(\mathcal{S}_b, \mathcal{T}_b)$ is the directed graph $\Gamma_b^2(\mathcal{S}_b \times \mathcal{S}_b, \mathcal{T}_b \times \mathcal{T}_b)$ with states $s_{i,j}$ defined as

$$s_{i,j} = (s_i, s_j) \ : \ 0 \leqslant i \leqslant j < M_b. \tag{7}$$

For any pair of transitions $(u, v)$ in the original graph, $\Gamma_b^2$ contains a directed edge $e$ with

$$e = (u, v), \tag{8}$$

$$\sigma(e) = s_{\sigma(u),\sigma(v)} \text{ and } \tau(e) = s_{\tau(u),\tau(v)}. \tag{9}$$

The weight of the edge $e$, $w_{\mathrm{H}}(e)$ is defined as the Hamming distance between the outputs of the two transitions $u$ and $v$,

$$w_{\mathrm{H}}(e) = d_{\mathrm{H}}(u, v). \tag{10}$$

A directed path $\boldsymbol{e}$ in $\Gamma_b^2$ from the state $\mathrm{s}_{i,j}$ to the state $\mathrm{s}_{m,n}$, is a sequence of edges $\boldsymbol{e} = (e_1 \circ e_2 \circ \cdots \circ e_N)$ such that $\sigma(e_{\mu+1}) = \tau(e_\mu)$ for $1 \leqslant \mu < N$. The weight of this directed path, $w_{\mathrm{H}}(\boldsymbol{e})$ is

$$w_{\mathrm{H}}(\boldsymbol{e}) = \sum_{\mu=1}^{N} w_{\mathrm{H}}(e_\mu). \tag{11}$$

Form two sets of states, $\mathcal{S}_{\text{div}}$ and $\mathcal{S}_{\text{conv}}$, in the product graph. $\mathcal{S}_{\text{div}}$ is the set of states of $\Gamma_b^2$ in which the outgoing edges consist of pairs of diverging transitions in $\mathcal{T}_b^2$ having the same originating state in $\mathcal{S}_b$ and $\mathcal{S}_{\text{conv}}$ is the set of states of $\Gamma_b^2$ in which the incoming edges consist of pairs of distinct transitions in $\mathcal{T}_b^2$ converging in the same target state in $\mathcal{S}_b$.

$$\mathcal{S}_{\text{div}} = \left\{ s_{i,i} : \exists\, u \neq v \in \mathcal{T}_b^2 \text{ and } \sigma(u) = \sigma(v) = s_i \right\}, \tag{12}$$

$$\mathcal{S}_{\text{conv}} = \left\{ s_{i,i} : \exists\, u \neq v \in \mathcal{T}_b^2 \text{ and } \tau(u) = \tau(v) = s_i \right\}. \tag{13}$$

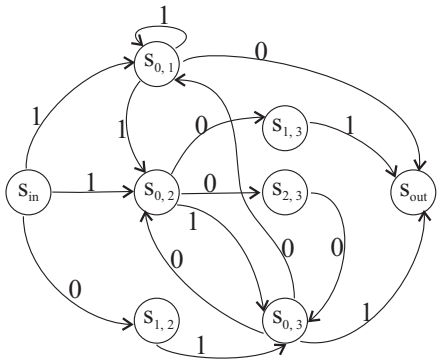


Fig. 2. Pairwise distance graph derived from the B-FSE in Fig. 1(b)

By merging the states in $\mathcal{S}_{\text{div}}$ into a single state $s_{\text{in}}$ and $\mathcal{S}_{\text{conv}}$ into a single state $s_{\text{out}}$ we obtain the PDG. Finding $d_{\text{free}}$ with this PDG is equivalent to finding a directed path from $s_{\text{in}}$ to $s_{\text{out}}$ with minimal weight. This is known as the shortest weighted path problem in graph theory and can be solved efficiently using Dijkstra's algorithm [12], since all weights are non-negative. The PDG derived from the B-FSE in Fig. 1(b) is represented in Fig. 2.

The code optimization using the search tree relies on bounds on the free distance of partially defined codebooks.

*Definition 3:* An EC-VLC is an *incomplete codebook* (IC) if some codewords or parts of codewords are not determined. For $\mathcal{X}_3$, two examples are $\mathcal{C}^1 = \{0, 10, xxx\}$ and $\mathcal{C}^1 = \{0, 1x, 1xx\}$, where $x$ stands for an undetermined bit. A *complete codebook* (CC) is an EC-VLC in which all codewords are determined. $\{0, 10, 110\}$ is an example for $\mathcal{X}_3$.

*Definition 4:* An incomplete (complete) codebook $\mathcal{C}^1$ is derived from an IC $\mathcal{C}^0$ (denoted $\mathcal{C}^0 \subset \mathcal{C}^1$) if it is obtained by specifying some (all) undetermined bits in $\mathcal{C}^0$.

If two (incomplete) codebooks $\mathcal{C}^0$ and $\mathcal{C}^1$ satisfy $\mathcal{C}^0 \subset \mathcal{C}^1$, then upper and lower bounds on the free distance of $\mathcal{C}^1$ can be obtained directly from $\mathcal{C}^0$. For this end, the PDG of $\mathcal{C}^0$ is constructed. To get an upper or a lower bound, the weights of the transitions which contain an undetermined bit ($x$) are replaced by 1 or 0, respectively, and then applying Dijkstra's algorithm.

## III. Structuring the search space

### A. Trees of EC-VLCs

The approach in this work is to arrange all EC-VLCs for a given Kraft vector $\boldsymbol{\ell} = (\ell_1, \ell_2, \ldots, \ell_M)$ in a tree data structure, such that every leaf corresponds to a specific EC-VLC codebook, and (internal) parent nodes correspond to partially defined codebooks, from which children nodes (codebooks) may be obtained by specific rules (to be described). This tree data structure should not be mistaken with the tree representing a prefix-free VLC. Then we explore this tree using an efficient branch-and-prune algorithm to find one of the EC-VLCs with the largest free distance. The key to efficient pruning is the availability of an upper bound on the free distance of partial codebooks that is monotonically nonincreasing when traversing the tree from the root towards the leaves. At each step of the algorithm, a list of nodes to be explored is sorted according to the free distance bound and the node with the largest upper bound is explored first. Thus partial codebooks leading to potentially large free distance are examined first. The pruning efficiency can be further improved by using a lower bound on the free distance as a secondary sorting criterion, *i.e.*, for equal upper bound, the partial codebook with the largest lower bound will be extended first (the sorting criteria should not be inverted, since upper bounds turned out to be much more discriminating for partial codebooks). The same approach has been successfully used for the optimization of error-correcting arithmetic coding [7].

For a given Kraft vector, there is a total of $2^{\sum_{i=1}^{M} \ell_i}$ codebooks, including such that are not uniquely decodable. Here, we will only consider *prefix-free* codebooks and additionally use symmetry properties to discard some codebooks known to have that same free distance than codebooks already considered. Next we outline three methods for structuring trees of EC-VLCs, *i.e.*, for creating hierarchies of partial codebooks.

### B. Construction by codewords

A straightforward method to structure the tree of EC-VLCs is to add one codeword at a time, starting from an empty codebook at the tree root, such that child nodes inherit the partial codebook from their parent node and augment it by one codeword. Thus the tree will have $M$ levels.

Let us first introduce some notations. For any $x \in \mathbb{N}$ and $\ell \in \mathbb{N}$ such that $\ell \geq \lceil \log_2(x+1) \rceil$, $\mathrm{B}_\ell(x)$ is the binary

representation of $x$ using $\ell$ bits. For instance, $\mathrm{B}_3(1) = 001$. Consider some $x$, $\ell \geq \lceil \log_2(x+1) \rceil$ as just defined, and $\mathcal{A}$, a prefix-free codebook. We define by $\mathrm{pref}(\mathcal{A}, x, \ell)$ the function which is zero if $\mathcal{A} \cup \{\mathrm{B}_\ell(x)\}$ is a prefix-free codebook, and one otherwise. For example, $A = \{1, 01\}$. If $x = 1$ and $\ell = 3$, $\mathcal{A} \cup \{\mathrm{B}_\ell(x)\} = \{1, 01, 001\}$, then $\mathrm{pref}(\mathcal{A}, 1, 3) = 0$. If $x = 3$ and $\ell = 3$, then $\mathcal{A} \cup \{\mathrm{B}_\ell(x)\} = \{1, 01, 011\}$ is not prefix, thus $\mathrm{pref}(\mathcal{A}, 3, 3) = 1$. For any $\ell \in \mathbb{N}$, the set $\mathcal{V}_{\mathcal{A},\ell}$ contains all integers $x \in \mathbb{N}$ which satisfy $\mathrm{pref}(\mathcal{A}, x, \ell) = 0$:

$$\mathcal{V}_{\mathcal{A},\ell} = \{x \in \mathbb{N}_0 \ : \ \mathrm{pref}(\mathcal{A}, x, \ell) = 0\} \tag{14}$$

Finally, $v^j_{\mathcal{A},\ell}$ is the $j$th element of $\mathcal{V}_{\mathcal{A},\ell}$.

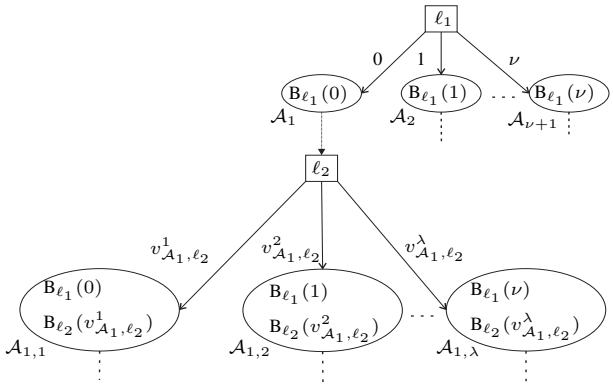


Fig. 3. Tree of EC-VLCs for the Kraft vector $(\ell_1, \ell_2, \ldots, \ell_M)$

Fig. 3 shows how all possible EC-VLCs corresponding to a Kraft vector can be ordered in a tree data structure. Since $\ell_1$ is the smallest length, the tree is initialized with $2^{\ell_1}$ codebooks which consist of a single codeword $\mathrm{B}_{\ell_1}(x)$ of length $\ell_1$ bits. These binary representations are shown in Fig. 3 by $\mathrm{B}_{\ell_1}(0)$ to $\mathrm{B}_{\ell_1}(\nu)$, where $\nu = 2^{\ell_1} - 1$. The obtained codebooks are named $\mathcal{A}_1$ to $\mathcal{A}_{\nu+1}$. Then, each codebook $\mathcal{A}_k$, $1 \leqslant k \leqslant \nu + 1$, is extended with different possible binary sequences $\mathrm{B}_{\ell_2}(v^j_{\mathcal{A}_k,\ell_2})$. In Fig. 3, for example, we suppose that $\mathcal{V}_{A_1,\ell_2}$ has $\lambda$ elements. Then, there are $\lambda$ possibilities to extend the codebook $\mathcal{A}_1$, leading to $\lambda$ new different codebooks denoted by $\mathcal{A}_{1,1}$ to $\mathcal{A}_{1,\lambda}$. Then the obtained new codebooks are extended with codewords with length $\ell_2$ and so on, until the codewords of length $\ell_M$ are added. Hence, all EC-VLCs corresponding to the Kraft vector $\boldsymbol{\ell}$ are obtained as the leaves of the final tree. Fig. 4 gives an example of a tree of EC-VLCs corresponding to the Kraft vector $\boldsymbol{\ell} = (1, 2, 3)$.

Inverting all bits of a EC-VLC does not change its distance properties. This symmetry property helps reducing the complexity of the branch-and-prune search and divide by two the time needed to find the best EC-VLC. To use this property, one can initialize the tree of EC-VLCs with $\mathrm{B}_{\ell_1}(0)$ to $\mathrm{B}_{\ell_1}(\frac{\nu+1}{2} - 1)$. In Fig. 4, only one part of the tree stemming from $\mathcal{A}_1 = \{0\}$ or $\mathcal{A}_2 = \{1\}$ is needed to find the best free distance.

Another useful symmetry property is that if the lengths $\ell_i$ and $\ell_{i+1}$ are equal, exchanging the codewords $i$ and $i+1$ does not change the average distance properties of the corresponding EC-VLC. Hence we can impose a lexicographic order

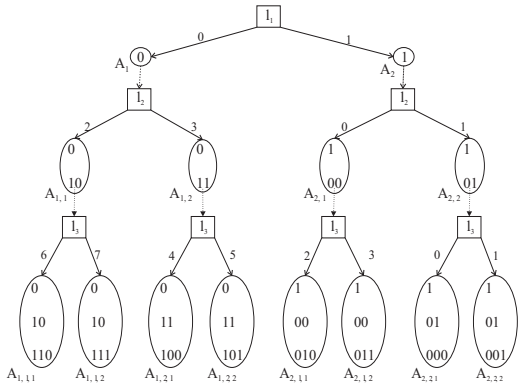

Fig. 4. Example of a tree of EC-VLCs for Kraft vector $\boldsymbol{\ell} = (1, 2, 3)$

in the construction of the codebooks. This may substantially reduce the time needed to find the best EC-VLCs.

### C. Construction by bitplanes

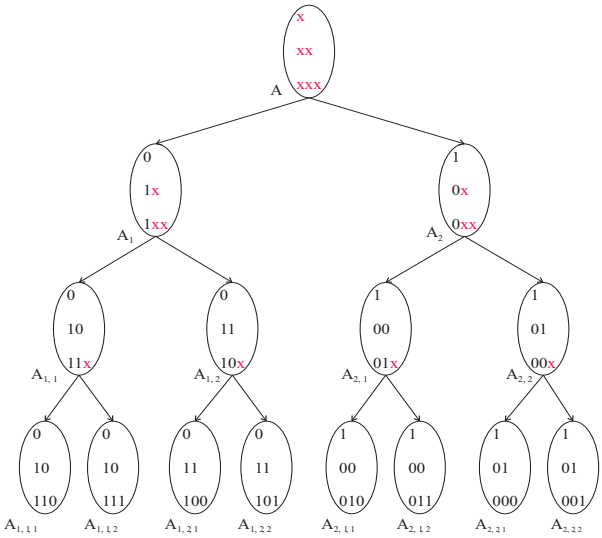

Fig. 5. Example of a tree of EC-VLCs obtained by bit plane construction for Kraft vector $\boldsymbol{\ell} = (1, 2, 3)$

Instead of successively adding codewords, one may build a codebook and thus obtain a tree of EC-VLCs by successively determining bitplanes, *i.e.*, by choosing the first bit for all codewords, then choosing the second bit (where present), and so on. In doing this, one must make sure that the suffixes of the codewords with a common prefix satisfy Kraft's inequality for the overall code to remain prefix-free. If the codewords for $i \in \mathcal{I}$ have a common prefix of length $\ell$, then the condition is $\sum_{i \in \mathcal{I}} 2^{\ell - \ell_i} \leqslant 1$.

Fig. 5 shows an example of generating a tree of EC-VLCs by bitplanes. In this figure, the codewords of each EC-VLC are ordered vertically. The tree is initialized with all possible combinations of the first bit of each codeword (by taking care that the EC-VLC remains prefix-free). The symbol $x$ represents the indeterminate bits of each codeword at a given time. Then the tree is explored by adding all possible

combinations of the next bits. At each step, we check if the suffixes of codewords having the same prefix satisfy Kraft's inequality.

### D. Construction using canonical code trees

Any prefix-free EC-VLC can be represented by a labeled binary code tree with leaves mapped to the $M$ source letters, such that the codeword for a letter can be read off as the concatenation of the (binary) labels from the root to the corresponding leaf. Clearly, if two codewords have a common prefix this will affect the distance between sequences starting with those words, regardless of the labels on this prefix. It can be seen that the structure of the unlabeled code tree already gives some information about the code, which can be used to derive upper and lower bounds on the free distance. To exploit this fact, we group the code trees into isomorphism (equivalence) classes, which can be arranged on a search tree (not to be confused with a code tree). Each isomorphism class can then be explored in turn using variants of the two methods outlined above.

*Definition 5:* Two binary trees are isomorphic if they can be transformed into each other by transposing (flipping) the children of internal nodes, including the root (*i.e.* all nodes stay at the same level, only their horizontal position changes, assuming the tree is drawn top-down from the root).

Fig. 6 shows an example of two isomorphic trees, while Fig. 7 shows an example of two non-isomorphic trees.

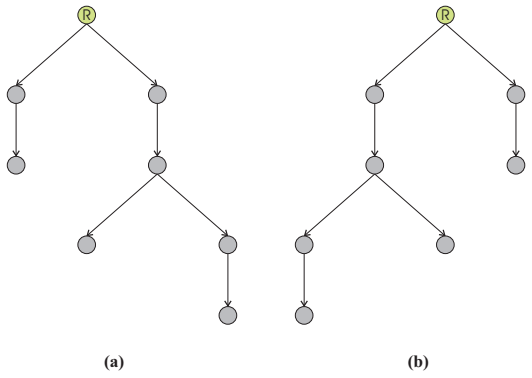


Fig. 6. Two isomorphic trees

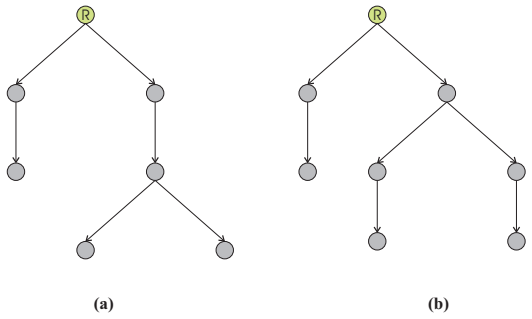


Fig. 7. Two non-isomorphic trees

An isomorphism class can be represented by an appropriately defined *canonical tree*, so the main problem becomes that of enumerating all canonical trees having a given Kraft vector $\boldsymbol{\ell}$. Such enumerations are classic problems in graph isomorphism; however, to the best of our knowledge, no algorithm is directly (and efficiently) applicable to the case when a Kraft vector is given. Thus we outline one below.

Let $T$ a binary tree, $\mathrm{left}(T)$ its left subtree, $\mathrm{right}(T)$ its right subtree (for compactness of notation, we identify the tree with its root; left and right outgoing edges may be thought as labeled 0 and 1, respectively). The function $\boldsymbol{\ell}^*(T)$ yields the Kraft vector in non-increasing order. For example, say $T$ has $\boldsymbol{\ell}(T) = (1,3,3,2)$. Then $\boldsymbol{\ell}^*(T) = (3,3,2,1)$. $\mathrm{Kraft}(\boldsymbol{\ell}^*(T)) = \mathrm{Kraft}(\boldsymbol{\ell}(T)) = \sum_{i=1}^{M} 2^{-\ell_i}$ is the Kraft sum. For example $\mathrm{Kraft}((2,1)) = 0.75$, $\mathrm{Kraft}((0)) = 1$, where $(0)$ stands for a single leaf node.

Define the order $\prec$ as follows:

$$T_1 \prec T_2 \quad \text{iff} \quad \boldsymbol{\ell}^*(T_1) \prec_{\text{lex}} \boldsymbol{\ell}^*(T2), \tag{15}$$

where $\prec_{\text{lex}}$ is the lexicographic order on integer vectors. *E.g.* $(1) \prec_{\text{lex}} (1)$, $(3) \prec_{\text{lex}} (3,2,2)$, $(0) \prec_{\text{lex}} (1)$, where $(0)$ stands for a single-node tree, and $(\emptyset) \prec_{lex} (0)$, where $(\emptyset)$ stands for no tree, *i.e.* an empty branch, which is "smaller" than anything.

*Definition 6:* A binary tree is canonical if it satisfies $\mathrm{left}(T_i) \prec \mathrm{right}(T_i)$ at all its internal nodes $T_i$ (*i.e.* recursively from the root down).

A canonical tree may be represented by traversing it in any well-defined order that visits each internal node $T_i$ once (preorder, inorder, postorder) and listing $(\boldsymbol{\ell}^*(\mathrm{left}(T_i)), \boldsymbol{\ell}^*(\mathrm{right}(T_i)))$.

To obtain a list of all canonical trees, start with the ordered Kraft vector $\boldsymbol{\ell}^*$ and split $\boldsymbol{\ell}^* - 1$ (componentwise subtraction as in Matlab, since we go one level down, we have to subtract one from the lengths) into two parts $\boldsymbol{\ell}_1^*$, $\boldsymbol{\ell}_2^*$ (which are again ordered) such that $\boldsymbol{\ell}_1^* \prec_{lex} \boldsymbol{\ell}_2^*$ and $\mathrm{Kraft}(\boldsymbol{\ell}_i^*) \leqslant 1$ $(i = 1,2)$ with one hitch: $\boldsymbol{\ell}_1^*$ may be empty (no leaf), so we define $\mathrm{Kraft}((\emptyset)) = 1$. Repeat recursively for $\boldsymbol{\ell}_1^*$ (the left subtree) and $\boldsymbol{\ell}_2^*$ (right subtree).

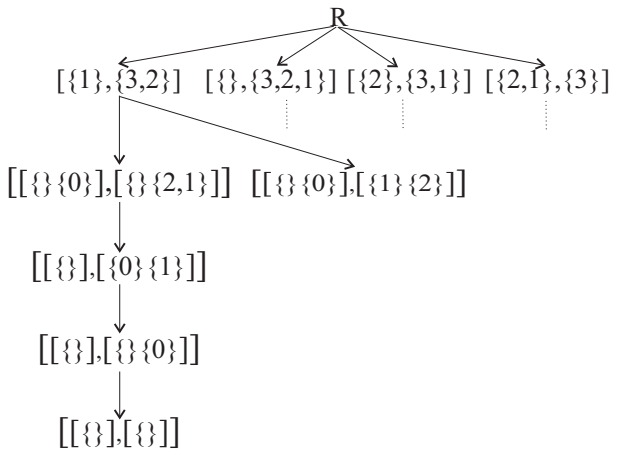


Fig. 8. Generating all canonical trees for $\boldsymbol{\ell} = (2,3,4)$

For example, consider the Kraft vector $\boldsymbol{\ell} = (3, 2, 4)$. Fig. 8 shows how canonical trees may be obtained from $\boldsymbol{\ell}$. We have $\boldsymbol{\ell}^* - 1 = (3, 2, 1)$. At the first level, represented by the node R, we have four possible choices for $(\boldsymbol{\ell}_1^*,\ \boldsymbol{\ell}_2^*)$: $(\boldsymbol{\ell}_1^* = (\emptyset),\ \boldsymbol{\ell}_2^* = (3, 2, 1))$, $(\boldsymbol{\ell}_1^* = (1),\ \boldsymbol{\ell}_2^* = (3, 2))$, $(\boldsymbol{\ell}_1^* = (2),\ \boldsymbol{\ell}_2^* = (3, 1))$ and $(\boldsymbol{\ell}_1^* = (2, 1),\ \boldsymbol{\ell}_2^* = (3))$. Each choice leads to one or more possible canonical trees. At the second level, we have several ways to split $\boldsymbol{\ell}_1^* - 1$ and $\boldsymbol{\ell}_2^* - 1$ for the pair $(\boldsymbol{\ell}_1^* = (1),\ \boldsymbol{\ell}_2^* = (3, 2))$. We repeat this process for all internal nodes. This leads to the first branch on the left of Fig. 8 which is the first obtained canonical tree, represented by Fig. 6 (a).

As mentioned above, the structure of the (canonical) code tree allows to already compute bounds on the free distance. For example, the tree in Fig. 7 (a) contains two codewords of length three, having two common prefix bits. Hence the free distance of any code derived from this tree will be the upper bounded by one. Algorithmically, this can be accomplished by labeling the canonical tree with special labels. Start by numbering the internal nodes. Then, label the left branch leaving node $i$ with "$L_i$" and the right branch with "$R_i$." Now, when constructing the PDG for bounding $d_{\text{free}}$, we may exploit the knowledge that $d_{\text{H}}(L_i, R_j) = 1$ if $i = j$, while for all other distances, we insert either 0 or 1, depending on the type of bound (lower, upper) we want to compute.

For a given Kraft vector $\boldsymbol{\ell}$, we first list all canonical trees and compute an upper bound on the free distance for each tree (this may also be done for partially known trees). The trees with the largest upper bound will then be explored first, using variations of the methods defined in Sections III-B and III-C.

## IV. Experimental Results

Experiments were run to display the time savings over an exhaustive search gained by applying the branch-and-prune algorithm on a search tree. The results shown in Table I are for the Kraft vector $\boldsymbol{\ell} = (3, 4, 5, 6)$, for which the maximal free distance is $d_{\text{free}} = 4$. The three methods for constructing this tree of EC-VLCs described in Sections III-B (*by codeword*), III-C (*by bitplane*) and III-D (using *canonical trees*) are compared. The row "# VLCs" shows the number of intermediate EC-VLCs that were examined by the algorithm. Clearly, the branch-and-prune algorithm is a promising method to find a code with maximal $d_{\text{free}}$ for a given Kraft vector.

The main benchmark is the number of intermediate EC-VLCs that need to be examined, while the execution times are only partially comparable, due to implementation differences. Indeed the complexity of computing (a bound on) $d_{\text{free}}$ is the number of states in the pairwise distance graph (PDG). In our implementation, when bounding $d_{\text{free}}$ of intermediate EC-VLCs using *bitplanes* or *canonical trees*, the PDG is static and has maximal number of states, whereas in the approach *by codewords*, this number of states is dynamic and can be as small as two. The rather disappointing performance of the *canonical trees* method has two origins: on one hand, for simplicity we chose to label the trees one bit at a time (*i.e.* neither by codewords, nor by bitplanes), which likely increases the number of trees that need to be examined. On the other, listing all canonical trees on the first level of the (branch-and-prune) optimization tree is suboptimal, because it does not allow to prune incomplete canonical trees. Other experiments showed that the *bitplane* and *canonical tree* methods are more efficient for codes having many codewords of the same length.

TABLE I
Comparison between exhaustive search and three branch-and-prune algorithms

| Method | **Exhaustive** | **Bitplane** | **Canon. tree** | **Codeword** |
|---|---|---|---|---|
| # VLCs | 72800 | 4070 | 3286 | 1222 |
| Time [s] | 2477 | 222 | 118 | 16 |

## V. Conclusion

In this paper, we propose three methods to build a tree ordering all prefix-free EC-VLCs with a given Kraft vector $\boldsymbol{\ell}$. First results show that using a branch-and-prune algorithm on a tree built with one these methods yields fast algorithms to optimize the free distance, when compared to an exhaustive search algorithm. Qualitatively, we observed that the method *by codewords* works better for Kraft vectors with distinct lengths, while the *bitplane* and *canonical tree* approaches are more efficient when many codeword lengths are equal.

## Acknowledgments

The authors would like to thank Pierre Duhamel for helpful discussions and suggestions.

This work was partly supported by the European Commission in the framework of the FP7 Network of Excellence in Wireless COMmunications NEWCOM++ (contract n. 216715).

## References

[1] C. E. Shannon, "A mathematical theory of communication," *Bell Syst. Tech. J.*, vol. 27, pp. 379–423 and 623–656, 1948.

[2] T. M. Cover and J. M. Thomas, *Elements of Information Theory*. New-York: Wiley, 1991.

[3] Y. Zhong, F. Alajaji, and L. L. Campbell, "On the joint source-channel coding error exponent for discrete memoryless systems," *IEEE Trans. Inform. Theory*, vol. 52, no. 4, pp. 1450–1468, 2006.

[4] A. J. Viterbi and J. Omura, *Principles of Digital Communication and Coding*. New-York: McGraw-Hill, 1979.

[5] C. Boyd, J. Cleary, I. Irvine, I. Rinsma-Melchert, and I. Witten, "Integrating error detection into arithmetic coding," *IEEE Trans. Commun.*, vol. 45, no. 1, pp. 1–3, 1997.

[6] S. Ben-Jamaa, C. Weidmann, and M. Kieffer, "Analytical tools for optimizing the error correction performance of arithmetic codes," *IEEE Trans. Commun.*, vol. 56, no. 9, pp. 1458–1468, September 2008.

[7] A. Diallo, C. Weidmann, and M. Kieffer, "Optimizing the search of finite-state joint source-channel codes based on arithmetic coding," *Eusipco*, 2009.

[8] M. Bernard and B. Sharma, "Some combinatorial results on variable-length error-correcting codes," *ARS Combinatoria*, vol. 25B, pp. 181–194, 1988.

[9] V. Buttigieg, "Variable-length error correcting codes," PhD dissertation, University of Manchester, Univ. Manchester, U.K., 1995.

[10] A. J. Viterbi, "Convolutional codes and their performance in communication systems," *IEEE Trans. Commun. Technol*, vol. 19, no. 5, pp. 751–772, 1971.

[11] A. Diallo, C. Weidmann, and M. Kieffer, "Efficient computation and optimization of the free distance of variable-length finite-state joint source-channel codes," Dec 2009, submitted to IEEE Trans. Commun.

[12] M. Gondran and M. Minoux, *Graphs and algorithms*. Chichester, UK: Wiley, 1984.