Individually Optimal Single- and Multiple-Tree Almost Instantaneous Variable-to-Fixed Codes

Danny Dubé and Fatma Haddad
Université Laval, Quebec City, Canada
Email: Danny.Dube@ift.ulaval.ca, Fatma.Haddad.1@ulaval.ca

***Abstract*—Variable-to-fixed (VF) codes are often based on dictionaries that obey the prefix-free property; e.g., the Tunstall codes. However, correct VF codes need not be prefix free. Removing that constraint may offer the opportunity to build more efficient codes. Here, we come back to the almost instantaneous VF codes introduced by Yamamoto and Yokoo. They considered both single trees and multiple trees to perform the parsing of the source string. We show that, in some cases, their technique builds suboptimal codes. We propose correctives accordingly. We also propose a new, completely different technique based on dynamic programming that builds individually optimal dictionary trees.**

I. Introduction

When it comes to variable-to-fixed (VF) codes, Tunstall's technique is usually mentioned [1], as it is "optimal". In order to be more accurate, we should say that Tunstall's technique selects optimal VF codes among the VF codes whose dictionaries obey the *prefix-free* (PF) *property* (i.e. among PFVF codes). However, the dictionary of a correct VF code does not need to be PF. The dictionary only needs to be *exhaustive*. As such, VF codes with dictionaries that are not PF have been considered by Savari [2] and Yamamoto and Yokoo [3], leading to strictly superior performance in certain cases.

In this paper, we focus on the VF codes considered by Yamamoto and Yokoo, which are called *almost instantaneous* (AI) *VF codes* (AIVF codes) [3]. They present a technique that aims at building VF codes that maximize the average length of the prefix that gets matched on the source string per step. The technique may build AIVF codes for two modes: the *single-tree* and the *multiple-tree* modes. As the name suggests, in single-tree mode, a single dictionary is used while, in multiple-tree mode, a *family* of dictionaries is used, where the latter are specialized for contexts where certain symbols are known *not* to start the remainder of the source string. At any given step, such partial information, denoted by i, about the source string is obtained from the previous step: if the next symbol had been one in $\{a_1, a_2, \ldots, a_i\}$, then a longer match would have been possible, and since the longer match did not happen, it implies that the next symbol is instead one in $\{a_{i+1}, \ldots, a_A\}$. Unfortunately, the Yamamoto-Yokoo construction technique (YY) fails to build optimal VF codes, in certain cases. We show the existence of such failures and propose correctives. As an alternative to YY, we propose a different approach which is based on *dynamic programming* (DP).

The rest of the paper is organized as follows. Section II establishes some notation and definitions. Section III presents the Tunstall and YY techniques. Section IV shows failure cases of YY and proposes correctives. Section V proposes our new DP technique. Note that a preliminary and summarized version of this paper was presented recently [4].

II. Notation, Definitions, and Conventions

Let the source alphabet be $\mathcal{A}$. Let $\mathcal{A}$ contain A symbols: a_1, a_2, …, a_A. The probability of a_i is $p(a_i)$. For convenience, we suppose that $p(a_1) \geq p(a_2) \geq \ldots \geq p(a_A)$. We consider the case where the symbols of the source string are generated independently and using an identical distribution (IID hypothesis). A source string $w = c_1\, c_2 \ldots c_n \in \mathcal{A}^n$ has probability $p(w) = \prod_{i=1}^{n} p(c_i)$ of being generated. Since we consider VF codes, the exact definition of the target alphabet is not relevant here and we keep it implicit. In a context where $i > 0$, $p(w)$ must be divided by $\sum_{j=i+1}^{A} p(a_j)$.

A crucial parameter for any VF code is the desired size M of its dictionaries. A VF code maps each *parseword* of a dictionary $\mathcal{D}$ of M parsewords to a corresponding *codeword* of a dictionary of M codewords. As with the target alphabet, we keep the dictionary of codewords implicit.

The term "parseword" is used because a VF code *parses* the source string w by decomposing it into a concatenation of parsewords from $\mathcal{D}$. In this paper, we ignore the issue of handling the few remaining symbols of w that cannot be parsed exactly into dictionary entries. Source string w gets encoded by repeating the following three operations: some prefix v of w gets matched to a parseword $v \in \mathcal{D}$, the prefix v gets extracted from w, and the codeword corresponding to v gets emitted. The sequence of emitted codewords provides a full description of w and a copy of w may be losslessly rebuilt from that description.

The choice of the parsewords in the dictionary $\mathcal{D}$ of a VF code is crucial as it determines the correctness and the efficiency of the code. In order for a VF code to be correct, $\mathcal{D}$ has to be *exhaustive*. $\mathcal{D}$ is exhaustive if, for any sufficiently long[1] string w, it is always possible to match some prefix of w with a parseword. A VF code is optimized by maximizing the average length of the matched parsewords; that is, we must choose $\mathcal{D}$ such that we maximize:

$$\overline{\text{len}}(\mathcal{D}) = \sum_{v \in \mathcal{D}} p_{\text{selection}}(v) \cdot |v| \ , \qquad \text{where}$$

$$p_{\text{selection}}(v) = p(v) - \sum_{u \in \mathcal{D} - \{v\},\, v \text{ is a prefix of } u} p_{\text{selection}}(u) \ .$$

[1] A *sufficiently long* string may indiscriminately be an infinite string or a finite string that is at least as long as the longest parseword.

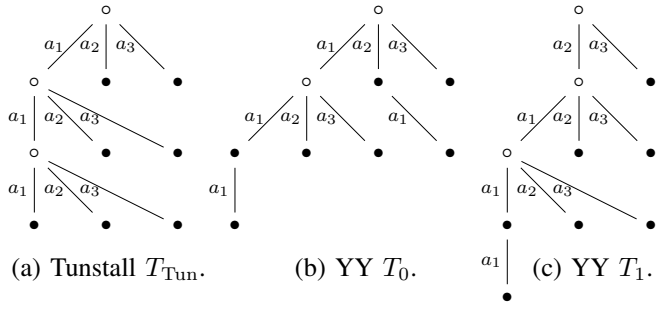


Fig. 1. Dictionary trees built using the Tunstall and the YY techniques.

Note that the equation uses $p_{\mathrm{selection}}$ instead of p as the probability of a parseword. Probability $p_{\mathrm{selection}}(v)$ is the probability that $v \in \mathcal{D}$ gets selected. In the case of an AIVF code, multiple parsewords of varying lengths may compete to match different prefixes of the source string. As a consequence, we generally have $p_{\mathrm{selection}}(v) \leq p(v)$. On the other hand, in the case of a PFVF code, we systematically have $p_{\mathrm{selection}}(v) = p(v)$.

For convenience, we allow ourselves to manipulate $\mathcal{D}$ under the form of a dictionary tree as well as under the form of a set of parsewords. The dictionary tree associated to $\mathcal{D}$ is depicted as follows. In the tree, there exists one and only one node per prefix of a parseword (or parsewords), even for improper prefixes[2]. We denote by n_v the node that corresponds to prefix v and, by abuse of language, we may say that n_v is v's node. Whenever prefixes v and vc both exist, for $c \in \mathcal{A}$, an arc labelled with c connects the parent n_v to the child n_{vc}. Some nodes correspond (exactly) to parsewords and they get marked accordingly. In the tree, such a node is depicted as a filled circle. We also say that such a node is *assigned* a codeword. On the other hand, a node for a prefix that is *not* a parseword is depicted as an empty circle. Figure 1 shows instances of trees that correspond to different dictionaries. In pseudo-code below, we write $t + W$ to denote a new tree like t that is augmented with all the new nodes whose paths are given in W. Also, we obtain the number of codewords that are assigned to the nodes of tree t using the notation $\#t$. In fact, when tree t is the graphical representation of dictionary $\mathcal{D}$, we have $\#t = |\mathcal{D}|$ (provided $\mathcal{D}$ contains no superfluous parsewords). Finally, if node n can be reached by following the path v from the root, then we may use the notation $\pi(n) = v$. Obviously, we have $\pi(n_v) = v$ for any parseword prefix v. Additional notation about trees is introduced in Section V but it is specialized for our DP technique.

Now, thanks to the correspondence with trees, we may easily define the AIVF codes: a VF code is an *AIVF code* if, in the tree that corresponds to its dictionary of parsewords, every incomplete node gets assigned a codeword; such nodes are the leaves and the incomplete internal nodes.

Let us describe the process of parsing using the example source string $w_{\mathrm{ex}} = a_2a_2a_1a_1a_1a_1a_1a_2a_3$. We first consider the Tunstall tree T_{Tun}. The parsing of w_{ex} would decompose it into the concatenation of the following parsewords: $a_2 \cdot a_2 \cdot a_1a_1a_1 \cdot a_1a_1a_2 \cdot a_3$. Each parseword would then be encoded into the corresponding codeword. Now, let us consider the YY tree T_0 in single-tree mode. A notable difference in T_0, compared to T_{Tun}, is that there exist incomplete internal nodes. Parsing w_{ex} would decompose it into: $a_2 \cdot a_2a_1 \cdot a_1a_1a_1 \cdot a_1a_2 \cdot a_3$. Note that the first parseword is limited to a single symbol since a_2 is not followed by a_1. On the other hand, the second parseword *has to be* two-symbols long, even if parseword a_2 exists, because matching must always be performed in a *greedy* fashion. Finally, let us consider both YY trees in multiple-tree mode. This mode is more complicated since matching may be performed using a different tree at each step. String w_{ex} would be parsed this way: $a_2 \cdot a_2a_1a_1a_1 \cdot a_1a_1 \cdot a_2a_3$. The first parseword would be identified by matching against T_0. Note that node n_{a_2} is an internal node with only one (1) outgoing arc, which is labelled a_1. This fact, plus the rule of greedy matching, implies that the next source symbol cannot be a_1. This partial information about the source string is to be used in the second step. As a consequence, the second parseword is identified by matching against T_1. The reached node $n_{a_2a_1a_1a_1}$ is a leaf (i.e. a node with no (0) outgoing arcs), which means that the third parseword should be matched against T_0. The third parseword leads to $n_{a_1a_1}$, which has one (1) outgoing arc, which means that the last parseword should be identified by matching against T_1.

We comment on the definition of our goal, which consists in designing $\mathcal{D}$ such that $\overline{\mathrm{len}}(\mathcal{D})$ is maximized. Unfortunately, this definition is an oversimplification. In single-tree mode, only tree T_0 has to be built, as if we were always in a context where we do not have any information about the next symbol of the source; i.e. as if $i = 0$. However, if T_0 contains incomplete internal nodes, then, after an encoding step has been performed, the source may be left in a state such that $i > 0$. So, in general, every matching step except the first one starts in a context that is a probabilistic mixture of different values of i. Taking the probabilistic mixture into account is not a difficult problem and it is addressed in the paper by Martinez et al [5]. In multiple-tree mode, each tree T_i is used exactly in the contexts where the partial information is i. However, having a family of trees where each tree individually has a maximal average parseword length does not guarantee that their joint use causes the best compression. The best compression also depends on the frequency of every value of i. We come back to this issue in Section VI.

III. The Tunstall and Yamamoto-Yokoo Techniques

We present YY for the single- and multiple-tree modes. Also, for the sake of completeness, we start by presenting the Tunstall code construction technique. Note that all algorithms are written using our notation. They are presented differently in their respective original papers. However, we make sure to preserve the operations they perform. Note that, since these techniques are prior work, our presentation is very succinct. In

[2] A prefix v of a string w is *improper* if v is the empty string or v is equal to w. Otherwise, v is a *proper* prefix of w.

Algorithm 1 COMPLETE(t): add a node's missing children

$\mathcal{V} \leftarrow \{\pi(n) \mid n$ is an incomplete node in $t\}$
$v_{\max} \leftarrow \arg\max_{v \in \mathcal{V}} p(v)$
$\mathcal{W} \leftarrow \{v_{\max}\} \cdot \mathcal{A}$ /* Paths to children of $v_{\max}$ */
return $t + \mathcal{W}$

Algorithm 2 TUNSTALL(M): build a PF t s.t. $\#t \leq M$

Require: $M \geq A$
$t_{\text{new}} \leftarrow$ ROOT $+ \{a_1, \ldots, a_A\}$
repeat
  $t_{\text{old}} \leftarrow t_{\text{new}}$
  $t_{\text{new}} \leftarrow$ COMPLETE(t_{old})
until $\#t_{\text{new}} > M$ /* Ultimate t_{new} gets discarded */
return t_{old} /* $M - (A-1) < \#t_{\text{old}} \leq M$ */

this section, we use a running example (taken from Yamamoto and Yokoo [3]) to illustrate the various VF codes that may be built for a source. Let $\mathcal{A}_{\text{ex}}$ be $\{a_1, a_2, a_3\}$, where $p(a_1) = 3/5$, $p(a_2) = 3/10$, and $p(a_3) = 1/10$. Let $M_{\text{ex}} = 7$ be the desired number of codewords in the VF codes.

A. Tunstall's Dictionary

Tunstall has proposed "the" classical technique to build VF codes [1]. The Tunstall technique is described by Algorithms 1 and 2. The Tunstall technique always builds optimal PFVF codes. It may fail to reach a dictionary size of exactly M parsewords, since each growth operation adds $A-1$ codewords to the tree. We chose to describe the COMPLETE procedure separately since it is also used by the YY technique.

Figure 1(a) shows the dictionary tree of the code that is built by TUNSTALL for $\mathcal{A}_{\text{ex}}$ and M_{ex}. We can see that T_{Tun} only contains nodes whose degrees are either 0 or 3. The tree features 7 nodes associated with parsewords (as requested). The 3 other nodes are not associated with parsewords.

B. Yamamoto-Yokoo Dictionary in Single-Tree Mode

In 2001, Yamamoto and Yokoo proposed to consider a class of VF codes that is more general than the PFVF codes: the AIVF codes [3]. Technique YY for both single- and multiple-tree modes is described in Algorithms 3 and 4. An M-codeword tree is created using YY$(0, M)$. Parameter 0 is used to indicate that we want the tree that assumes nothing about the very next symbol of the source string. YY works by iteratively growing a dictionary tree using two strategies that rival each other: Tunstall's COMPLETE and EXTEND. The EXTEND strategy is a purely greedy one: it builds the (or a) single new child that has the maximal probability. If we want to grow k children this way, we simply repeat this operation k times (EXTEND^k). Note that the initial tree consists only of the root and its children. Trees built by YY are exhaustive because the positioning of the codewords is made implicitly using the rule that AIVF codes must obey.

At first sight, one might think that the first strategy cannot be competitive with respect to the second strategy, since the second one only grows the best children, while the first one does not pay close attention to the individual children it grows.

Algorithm 3 EXTEND(t): add the best child

$\mathcal{V} \leftarrow \{\pi(n) \mid n$ is a node in $t\}$ /* Paths to all nodes */
$\mathcal{W} \leftarrow (\mathcal{V} \cdot \mathcal{A}) - \mathcal{V}$ /* Paths to potential children */
$w_{\max} \leftarrow \arg\max_{w \in \mathcal{W}} p(w)$
return $t + \{w_{\max}\}$

Algorithm 4 YY(i, M): build an AI t of type T_i s.t. $\#t = M$

Require: $0 \leq i \leq A-2$ **and** $M \geq A - i$
$t_{\text{new}} \leftarrow$ ROOT $+ \{a_{i+1}, \ldots, a_A\}$
repeat
  $t_{\text{old}} \leftarrow t_{\text{new}}$
  $t_{\text{I}} \leftarrow$ COMPLETE(t_{old}) /* Option I */
  $t_{\text{II}} \leftarrow \text{EXTEND}^{\#t_{\text{I}} - \#t_{\text{old}}}(t_{\text{old}})$ /* Option II */
  $t_{\text{new}} \leftarrow$ the best of t_{I} and t_{II}
until $\#t_{\text{new}} > M$
return $\text{EXTEND}^{M - \#t_{\text{old}}}(t_{\text{old}})$ /* Option II only */

However, the advantage of the first strategy comes from the fact that, by completing a node n, it relieves n from the obligation to be assigned a codeword. So the first strategy can grow k children at once by requiring only $k-1$ extra codewords (one codeword per new child, minus one codeword for the now complete parent).

Figure 1(b) shows the AIVF tree T_0 built by the YY technique for $\mathcal{A}_{\text{ex}}$ and M_{ex} in single-tree mode. Note that T_0 does not obey the PF property, as the property is violated by $n_{a_1 a_1}$ and $n_{a_1 a_1 a_1}$ (and also by n_{a_2} and $n_{a_2 a_1}$). The freedom to consider up to AIVF codes is beneficial because T_0 leads to a higher average parseword length than T_{Tun}: $499/250$ versus $49/25$, respectively.

C. Yamamoto-Yokoo Dictionaries in Multiple-Tree Mode

In multiple-tree mode, the whole family of AIVF codes is built using the YY procedure. Trees $T_0, T_1, \ldots, T_{A-2}$ are built by making the calls YY$(0, M)$, YY$(1, M)$, $\ldots$, YY(i, M), respectively. The root of each tree T_i is the parent of the appropriate children thanks to the initialization phase. Note that the trees T_{A-1} and T_A are not considered. Tree T_A, for sure, does not make sense since it would assume that the very next symbol of the source string is none from $\mathcal{A}$. Tree T_{A-1} is not considered either. Assuming that the very next symbol cannot be one of $a_1, \ldots, a_{A-1}$ would necessarily mean that the next symbol would have to be a_A. This means that the previous parsing step would have left symbol a_A on the source string, despite the fact that it would have known for sure that it was a_A. YY does not consider this possibility.

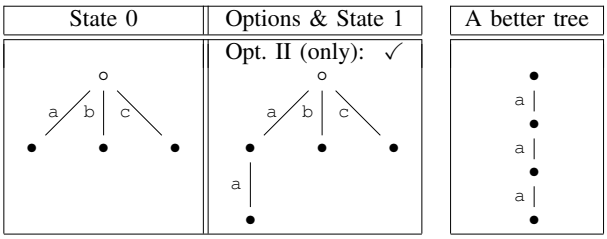


Fig. 2. Suboptimal construction due to the complete-root constraint.

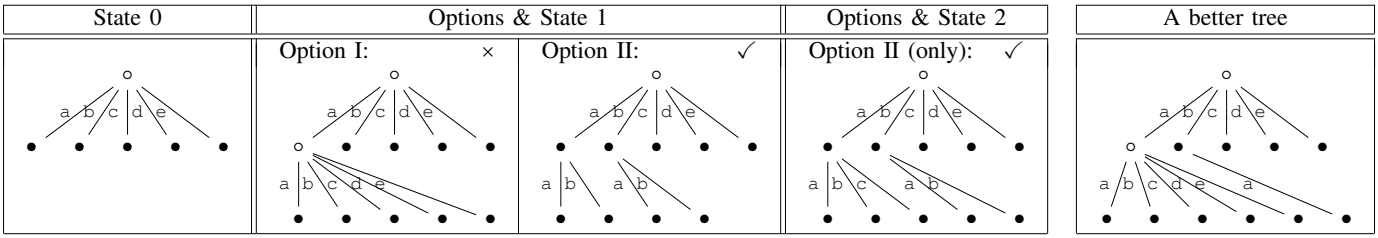


Fig. 3. Suboptimal construction due to the full execution of Option II's propositions.

Figure 1(c) shows T_1 built by the YY technique for $\mathcal{A}_{\text{ex}}$ and M_{ex} in multiple-tree mode. (Note that T_0 remains the same as in single-tree mode; i.e. the one shown in Figure 1(b).) Tree T_1 also represents a dictionary with 7 parsewords but note that its root has no outgoing arc labelled with a_1.

IV. Defects in the Yamamoto-Yokoo Technique

Despite the fact that YY is intended to build optimal AIVF codes, it suffers from subtle defects. In this section, we show the existence of two defects that we have found. Then, we present two modifications to YY such that the modified version would be more likely to build optimal AIVF codes.

A. Complete-Root Constraint in Multiple-Tree Mode

Note that YY initializes the dictionary tree with a complete root. While a complete root is a necessity in single-tree mode, to ensure progress during parsing, it is *not* in multiple-tree mode. In multiple-tree mode, it is not fatal for a dictionary to have an empty parseword. It would only mean that such a tree T_i would leave to T_{i+1} the care to handle certain symbols.[3]

In order to demonstrate the existence of this defect, we devised a small counter-example. Let us suppose we want to build T_0 in multiple-tree mode. Let $\mathcal{A}$ be $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, with $p(\mathtt{a}) = \alpha$, $p(\mathtt{b}) = \beta$, and $p(\mathtt{c}) = \gamma$, and let M be 4. Also, let us suppose that α, β, and γ are chosen this way: $\theta = \frac{\sqrt{31}}{2} + \frac{3\sqrt{3}}{2}$, $\omega = \sqrt[3]{\theta}$, $\psi = \omega - \frac{1}{\omega}$, $\phi = \psi/\sqrt{3} \approx 0.68233$, $\alpha > \phi$, and $\beta = \gamma = \frac{1-\alpha}{2}$. Figure 2 shows the trace of the construction of T_0^{YY} using YY and it shows another tree, T_0^{DH}, where DH are the authors' initials. We leave to the reader the verification that T_0^{DH} indeed has a higher average parseword length than T_0^{YY}.

B. Full Execution of Propositions by Option II

There is another defect in YY that is much more subtle. It is a consequence of the "full execution" of the growth propositions made by Option II. Indeed, even if the pseudo-code for YY manipulates propositions under the form of augmented trees, we can rather view the propositions under the form of a sequence of arc creations, from a "most desirable one" to a "least desirable one". Under this alternate point of view, a growth operation in YY consists in fully executing the sequence of arc creations.

In order to demonstrate the existence of this defect, we devised another small counter-example. Let us suppose that we want to build T_0 in single-tree mode. Let $\mathcal{A}$ be $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$, with $p(\mathtt{a}) = 1/3$, $p(\mathtt{b}) = 1/4$, $p(\mathtt{c}) = 1/6$, $p(\mathtt{d}) = 3/20$, and $p(\mathtt{e}) = 1/10$, and let M be 10. Figure 3 shows the trace of the construction of T_0^{YY} by YY and another tree T_0^{DH}. We leave to the reader the verification that T_0^{DH} is better than T_0^{YY}.

[3]The codeword associated to the root would be a kind of *escape* symbol.

C. Two Correctives to the Yamamoto-Yokoo Technique

We briefly describe two modifications on YY that would improve its efficiency. In order to correct the first defect, when in multiple-tree mode, the initialization phase should *not* force the completion of the root; i.e. the initial tree should be a root only. We mention that Iwata and Yamamoto have made an analogous discovery in the somehow related work on AIFV (i.e. fixed-to-variable) codes [6]. In order to correct the second defect, when the proposition of Option II turns out to be the best of the two, only the first arc of the proposed sequence of arc creations should be grown; i.e. $\text{EXTEND}(t_{\text{old}})$ instead of t_{II} should be kept. The "better trees" shown in Figures 2 and 3 can be obtained using the two modifications, respectively.

Note that we are careful to only mention that the correctives *improve* the efficiency of YY. We deliberately avoid claiming that the correctives make YY optimal. This is because there is a possibility that yet another subtle defect still exists, despite both correctives. Indeed, although the *operations* performed by YY are simple to understand, their *consequences*, in terms of optimality, are not. Even if this section does not end with an optimal variant of YY, we think the section's value is in contributing to the eventual elaboration of an optimal variant. YY remains interesting for its speed and devising an optimal variant is a worthwhile goal.

V. Construction Technique Based on Dynamic Programming

We propose a new technique to build optimal AIVF codes for both single- and multiple-tree modes. The technique is completely different from YY. Ours is based on DP. In order to present it, we choose to denote by T_i^N the various optimal dictionary trees *without constraint on the root* other that the partial information i on the source, where N indicates the number of codewords included in the tree. Our DP technique allows a larger range for i than YY: $0 \leq i \leq A - 1$. The number N has to be a strictly positive natural number. Similarly, we denote by S_i^N the optimal *complete-root* trees.

The idea behind our technique is based on a single observation. Any optimal tree T_i^N of non-trivial size (i.e. $N \geq 2$), with imperfect information about the next symbol (i.e. $i \leq A - 2$),

Algorithm 5 BUILD(i, N): build T_i^N using DP

Require: $0 \le i \le A-1$ **and** $N \ge 1$
1: **if** $i = A-1$ **then return** DEFAULT(T_0^N) /* Fig. 4(a) */
2: **else if** $N = 1$ **then return** ROOT /* Fig. 4(b) */
3: **else** /* Fig. 4(c) ↘ */
4: $\mathcal{T} \leftarrow \{T_0^L \oplus T_{i+1}^R \mid L+R = N,\ L \ge 1,\ R \ge 1\}$ /* ↙ */
5: **return** $\arg\max_{t \in \mathcal{T}} \overline{\text{len}}(t)$ /* $O(N)$-time case */
6: **end if**

Algorithm 6 FILL(M): build all T_i^N using DP s.t. $N \le M$

Require: $M \ge 1$ /* $O(A \cdot M^2)$-time algorithm */
1: **for** $N = 1$ to M **do**
2: **for** $i = 0$ to $A-1$ **do**
3: $T_i^N \leftarrow$ BUILD(i, N)
4: **end for**
5: **end for**

Algorithm 7 BUILD$^{\text{single}}(i, N)$: build a complete-root T_i^N

Require: $0 \le i \le A-1$ **and** $N \ge A-i$
1: **if** $i = A-1$ **then return** DEFAULT(T_0^N)
2: **else**
3: $\mathcal{S} \leftarrow \{T_0^L \oplus S_{i+1}^R \mid L+R = N,\ L \ge 1,\ R \ge 1,\ R \ge A-(i+1)\}$
4: **return** $\arg\max_{t \in \mathcal{S}} \overline{\text{len}}(t)$
5: **end if**

Algorithm 8 FILL$^{\text{single}}(M)$: build all comp.-root T_i^N s.t. $N \le M$

Require: $M \ge 1$
1: **for** $N = 1$ to M **do**
2: **for** $i = \max(A-N, 0)$ to $A-1$ **do** /* S_i^N undef. ... */
3: $S_i^N \leftarrow$ BUILD$^{\text{single}}(i, N)$ /* ... for $N < A-i$ */
4: **end for**
5: **end for**

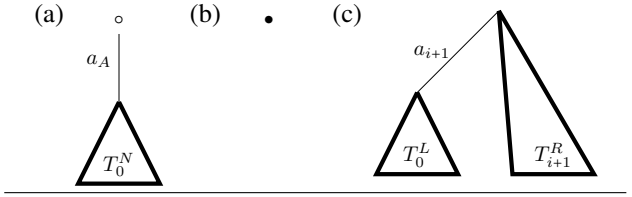
and free from the complete-root constraint should (or may as well) have the shape shown in Figure 4(c). In words, there should be at least one parseword in the dictionary that starts with a_{i+1}, the most probable of the symbols that may appear next in the source string. Moreover, the subtree t' rooted at $n_{a_{i+1}}$ must be optimal of the T_0 kind. Finally, the subtree t'' that is made with the remains of T_i^N if we remove t' and the incoming arc must be optimal of the T_{i+1} kind. Figure 4 also presents the shapes of the trees for the corner cases.

Given this idea, we may use Algorithms 5 and 6 to build optimal AIVF trees for the multiple-tree mode. Note that the multiple-tree mode is simpler than the single-tree mode, as the decision making is identical at the root of the trees and at deeper nodes. We may use Algorithms 7 and 8 to build optimal AIVF trees for the single-tree mode. Because our DP technique is so simple (and because of the 5-page limit), we trust that the given pseudo-code is self-explanatory.

DP is a standard algorithmic strategy and it is used in countless algorithms in computer science. In particular, DP has been used to build various codes in data compression; namely, AIFV codes [7] and `1`-ended codes [8].

VI. Future Work

First, we should verify whether applying our correctives on YY would make it optimal. Second, the AI property remains a constraint on the considered VF codes, even if it is looser than the PF property, and we should investigate on the opportunities offered by the removal or relaxation of this constraint; e.g., codes with a longer delay [7]. Third, optimizing every tree of a family individually may fail to reach the best compression for the whole family. It is sometimes more profitable to build a tree that is individually slightly less efficient but that is more likely to leave the source in a context with a higher i. Iwata and Yamamoto, in their study of AIFV codes, use a system of compensation to account for the impact of switching to one tree or another [7]. Our technique should integrate an analogous system. Fourth, when the alphabet is large and the multiple-tree mode is used, the full family of trees may occupy considerable space, as noted by Yoshida and Kida [9]. The latter proposed multiplexed trees, while we intend to represent the trees using the L-versus-R sizes calculated using DP.



(a) T_{A-1}^N = DEFAULT(T_0^N)
(b) T_i^1 = ROOT, where: $i \le A-2$
(c) $T_i^N = T_0^L \oplus T_{i+1}^R$, where: $i \le A-2$, $2 \le N = L+R$

Fig. 4. Fundamental shapes of the dictionary trees used in the DP technique.

Acknowledgment

We would like to thank Prof. Yamamoto and Prof. Iwata for having long and detailed discussions about this topic with the first author. We would also like to thank the anonymous reviewers for their comments.

References

[1] B. P. Tunstall, "Synthesis of noiseless compression codes," Ph.D. dissertation, Georgia Institute of Technology, 1967.
[2] S. A. Savari, "Variable-to-fixed length codes and plurally parsable dictionaries," in *Proc. of the Data Compression Conf.*, Mar. 1999, pp. 453–462.
[3] H. Yamamoto and H. Yokoo, "Average-sense optimality and competitive optimality for almost instantaneous VF codes," *IEEE Trans. on Information Theory*, vol. 47, no. 6, pp. 2174–2184, Sep. 2001.
[4] D. Dubé and F. Haddad, "Optimal single- and multiple-tree almost instantaneous variable-to-fixed codes," in *Proc. of the Data Compression Conf.*, Snowbird, Utah, USA, Mar. 2018, p. 405.
[5] M. Martinez, M. Haurilet, R. Stiefelhagen, and J. Serra-Sagristà, "Marlin: A high throughput variable-to-fixed codec using plurally parsable dictionaries," in *Proc. of the Data Compression Conf.*, Apr. 2017, pp. 161–170.
[6] K. Iwata and H. Yamamoto, "A dynamic programming algorithm to construct optimal code trees of AIFV codes," in *Proc. of the Int. Symp. on Information Theory and Applications*, Nov. 2016, pp. 641–645.
[7] ——, "An iterative algorithm to construct optimal binary AIFV-m codes," in *Proc. of the IEEE Information Theory Work.*, Nov. 2017, pp. 519–523.
[8] S.-L. Chen and M. J. Golin, "A dynamic programming algorithm for constructing optimal "1"-ended binary prefix-free codes," *IEEE Trans. on Information Theory*, vol. 46, no. 4, pp. 1637–1644, 2000.
[9] S. Yoshida and T. Kida, "An efficient algorithm for almost instantaneous VF code using multiplexed parse tree," in *Proc. of the Data Compression Conf.*, Mar. 2010, pp. 219–228.