On Procedures for Multiple-string Match with Respect to Two Sets[1]

Weiler A. Finamore, Rafael D. de Azevedo
& Marcelo da Silva Pinho

Center for Telecommunications Studies (CETUC)
Catholic University of Rio de Janeiro
Marqus de S. Vicente, 225
22453-900, RIO DE JANEIRO, RJ
Brazil

e-mail: weiler@cetuc.puc-rio.br

Abstract. String match procedures with respect to two sets are investigated. The procedures traditionally used for data compression are based on single-string match with respect to a single set [LZ78, W84]. Some recent work broadened this view by presenting procedures for multiple-string match with respect to a single set [FPC98, PFP99] with improved performance as compared to the single-match versions. In this work an algorithm based on double-match with respect to two sets is stated. We do conjecture that multiple-string match procedures with respect to two sets can achieve even better performance. A preliminary analysis corroborating this conjecture with some evidence is reported in this work.

Key words: Multiple-string match, Lempel-Ziv algorithm, Data compression.

1 Introduction

The procedure proposed by Lempel and Ziv in 1978 [LZ78] for lossless data compression is a rather simple and elegant string-match based algorithm. Its low complexity and implementation simplicity has turned it into a very popular algorithm which is used for instance in the *compress* program of UNIX operational system.

By selecting diferent combinations of the basic parameters of this algorithm many variations can be established. In the result published in [FPC98] a version that searches for double-string matches instead of the usual single-match is stated — an improved performance was obtained. Extension to multiple string-match was proposed in [PFP99]. Similar results were reported by Hartman and Rodeh in [HR85].

In this work the two most popular Lempel-Ziv variations, LZ78 and LZW [LZ78, W84], has been cast in the framework of string-match with respect to two sets. We also propose two new variations (designated lg-LZ and dt-LZ), which are inspired and discussed in this new framework. Although the ultimate goal of finding new

[1]This work was supported by grant **CNPq**-502235/91-8(NV) and **AEB/PR**-004/97.

algorithms with improved is a motivation behind the algorithms proposed, the imediate objective is to expand the ways of looking at the string matches algorithms and hopefully to find better procedures.

This work is organized as follows: in Section 3, we present the idea of string match with respect to two sets and establish a motivation by discussing two well-known algorithms in the framework of matching with respect to two sets. A new algorithm (lg-LZ) which is a simple variation of the Lempel-Ziv algorithm is also proposed in this section. In Section 4 a version of double-match/double-tree algorithm is introduced. Results obtained by computer simulation are presented in Section 5. Our conclusion is then summarized in Section 5.

2 Notations

We establish the following notation for use in this work.

1. $x_i^j = x_i x_{i+1} \ldots x_j$, $j > i$ denotes a finite sequence of symbols x_k, $i \leq k \leq j$, that take their values in a given set $\mathcal{A} = \{a_0, a_1, \cdot, a_{|\mathcal{A}|-1}\}$ of cardinality $|\mathcal{A}|$. If $j = i$, this is the single symbol string x_i and if $i > j$ we will assume that x_i^j is the empty string.

2. $|\alpha|$ denotes the length, if α is a sequence, or the cardinality, if α is a set.

3. λ denotes the null-length string, i.e. $|\lambda| = 0$.

4. $\mathbf{s}_i \circ \mathbf{s}_j$ denotes the concatenation of the strings $\mathbf{s}_i$ and $\mathbf{s}_j$. (the result of the concatenation will also be indicated by $\mathbf{s}_i\mathbf{s}_j$ or $\mathbf{s}_i$,$\mathbf{s}_j$)

5. When $\mathbf{s}_1, \mathbf{s}_2, \cdots, \mathbf{s}_k \in \mathcal{A}^*$ are strings of symbols of lengths $|\mathbf{s}_1|, |\mathbf{s}_2|, \cdots, |\mathbf{s}_k|$ respectively, the notation $\mathbf{s}_1^k$ represents the string of length $|\mathbf{s}_1| + |\mathbf{s}_2| + \cdots + |\mathbf{s}_k|$ formed by the concatenation of strings $\mathbf{s}_1 \circ \mathbf{s}_2 \circ \cdots \circ \mathbf{s}_k$.

6. The concatenation of the string $\ell \in \mathcal{L} = \{\ell_0, \cdots, \ell_{|\mathcal{L}|-1}\}$ and the set $\mathcal{M} = \{\mathbf{m}_0, \cdots, \mathbf{m}_{|\mathcal{M}|-1}\}$ is the set

$$\ell \circ \mathcal{M} = \bigcup_{i=0}^{|\mathcal{M}|-1} \{\ell \circ \mathbf{m}_i\}$$

7. Let $\mathcal{L} = \{\ell_0, \cdots, \ell_{|\mathcal{L}|-1}\}$ and $\mathcal{M} = \{\mathbf{m}_0, \cdots, \mathbf{m}_{|\mathcal{M}|-1}\}$. We define the concatenation of these two sets by

$$\mathcal{L} \circ \mathcal{M} = \bigcup_{i=0}^{|\mathcal{L}|-1} \{\ell_i \circ \mathcal{M}\}$$

8. $\lceil x \rceil$ denotes the smallest integer greater than or equal to number x.

9. $\Pi[\mathbf{z}|\mathcal{L}]$, for $|\mathcal{L}| > 0$, is the longest string $\ell_i \in \mathcal{L} = \{\ell_0, \cdots, \ell_{|\mathcal{L}|-1}\}$ which is a prefix of $\mathbf{z}$.

10. $\mathcal{X}[\mathbf{s}|\mathcal{L}]$ is the unique integer index i that identify the member $\ell_i \in \mathcal{L}$ such that $\ell_i = \mathbf{s}$.

11. $\mathbf{z} - \mathbf{y}$, when $\mathbf{z} = x_i x_{i+1} \cdots x_j$ and $\mathbf{y} = x_i x_{i+1} \cdots x_k$ is a prefix of $\mathbf{z}$, represents the string $x_{k+1} \cdots x_j$.

12. $\mathcal{F}[\mathbf{z}]$ is the length 1 prefix of $\mathbf{z}$, if $|\mathbf{z}| > 0$ else it is the empty string.

13. $\mathcal{S}[\mathbf{z}]$ is the length $|\mathbf{z}| - 1$ prefix of $\mathbf{z}$.

14. $\phi_k[J]$, $k \geq \log J$ (base 2 logarithm) is the trivial k-bit binary representation of the integer J.

3 The Idea of String Match Algorithm with Respect to Two Sets

To establish the framework and the rationale behind our discussion, the well-known string-match procedure proposed by Ziv and Lempel [LZ78] for data compression will be presented, in the context of string match with respect to two sets. We will undistinguishably refer to this as a double-tree string match context since the sets we will be dealing with are tree-structured.

3.1 Lempel-Ziv Algorithm (LZ78)

Let us consider that $\mathbf{z}_0 = x_0^{N-1}$ is the sequence of N symbols generated by the information source which is to be encoded (each source symbol x_i belongs to the source alphabet $\mathcal{A}$, of dyadic cardinallity for simplicity). Generally speaking the Lempel-Ziv algorithm (LZ78) [LZ78] can be envisioned as divided in three tasks: The first task, (parsing), which yields the unique parsing

$$x_0^{N-1} = (\ell_0 \circ \mathbf{m}_0), (\ell_1 \circ \mathbf{m}_1), \cdots, (\ell_t \circ \mathbf{m}_t)$$

of the source sequence in $t+1$ phrases. The next task, (map to integers), assign each phrase $\mathbf{s_i} = (\ell_i \circ \mathbf{m}_i)$ to a unique pair of integers (J_i, K_i) which are then, in the task that follows (integer code), replaced (or encoded) by a binary representation according to some rule to encode integer numbers into binary.

Specifically, the algorithm LZ78 [LZ78] can be stated using the double-tree framework by initially setting $\mathcal{L}_0 = \{\lambda, x_0\}$, $\mathcal{M}_0 = \mathcal{A}$ and $\mathbf{s}_0 = (\ell_0 \circ \mathbf{m}_0) = (\lambda \circ x_0) = x_0$. At a general step i, the sets $\mathcal{L}_{i-1}$ and $\mathcal{M}_{i-1}$ are known, the source string has been parsed in i phrases $\mathbf{s}_0, \cdots, \mathbf{s}_{i-1}$ and there is a remaining unparsed string which will be denoted by $\mathbf{z}_i$. The algorithm is described next.

Algorithm LZ78

$i = 0$

$\mathbf{z}_0 = x_0^{N-1}$
$\mathcal{L}_0 = \{\lambda\}$, $\mathcal{M}_0 = \mathcal{A}$
$\mathbf{s}_0 = \ell_0 \circ \mathbf{m}_0$ with $\ell_0 = \lambda$ and $\mathbf{m}_0 = x_0$.

$1 \leq i \leq t$

1. Update unparsed string:
   $\mathbf{z}_i = \mathbf{z}_{i-1} - (\ell_{i-1} \circ \mathbf{m}_{i-1})$
2. Find longest match $\mathbf{s}_i$ with respect to $\mathcal{D}_i = \mathcal{L}_{i-1} \circ \mathcal{M}_{i-1}$:
   $\mathbf{s}_i = \Pi[\mathbf{z}_i|\mathcal{D}_i] = \ell_i \circ \mathbf{m}_i$,
   with $\ell_i = \Pi[\mathbf{z}_i|\mathcal{L}_{i-1}]$, and $\mathbf{m}_i = \Pi[(\mathbf{z}_i - \ell_i)|\mathcal{M}_{i-1}]$.
3. $(J_i, K_i) = (\mathcal{X}[\ell_i|\mathcal{L}_{i-1}],\ \mathcal{X}[\mathbf{m}_i|\mathcal{M}_{i-1}])$
4. Update $\mathcal{L}$-tree:
   $\mathcal{L}_i = \mathcal{L}_{i-1} \cup \{\ell_i \circ \mathcal{F}[\mathbf{m}_i]\}$
   $\mathcal{M}_i = \mathcal{A}$
5. $(B_i, C_i) = (\phi_{\lceil \log |\mathcal{L}_{i-1}| \rceil}[J_i],\ \phi_{\lceil \log |\mathcal{M}_{i-1}| \rceil}[K_i])$

The efficiency of a string match algorithm is closely related to the number $t+1$ of phrases parsed off from the source string and to the rate of growth of the sets $\mathcal{L}$ and $\mathcal{M}$. In the present case, LZ78, $t+1$ phrases are generated and the N source symbols will be represented by L binary symbols,

$$L = \sum_{i=0}^{t} (|B_i| + |C_i|) = (t+1) \log_2 |\mathcal{A}| + \sum_{i=0}^{t} |B_i|,$$

rendering a $\rho = L/N$ compression rate. If the source symbols are drawn from an stationary source, the compression rate provedly [LZ78] converges to the entropy of the source. The interplay between these two parameters is quite envolved [S97] and is not our main concern. It is worth mentioning that Integer Codes more efficient than the one used to produce the binary block (B_i, C_i) could be used. An improvement in the above code, for instance, can be introduced simply by noticing that the phrase $\mathbf{s}_i$ which is parsed off at the i-th step, actually belongs to a set $\mathcal{D}_i$ (called dictionary or codebook)

$$\mathcal{D}_i = \mathcal{L}_{i-1} \circ \mathcal{M}_{i-1}$$

with some elements (or codewords) on it, which are not able to be selected as a match to $\mathbf{s}_i$ — the enumeration reserved for these are therefore a waste of bits. This is of little concern to us at this point and the Integer Code as it is will be used with the other algorithm versions discussed in the entire work.

The important point to be stressed in relation to the LZ78 is that no matter the value of i, the associated tree $\mathcal{M}_i$ is kept fixed, equal to $\mathcal{A}$. Whether there are procedures which performs more efficiently, by allowing $\mathcal{M}_i$, the second dictionary tree, to grow rather than be fixed, is a conjecture naturally raised. This issue is examined on the next section. A variation of the LZ78 which constructs the dictionary $\mathcal{D}_i$ in a sligthly different manner and which, for this reason, has a slightly better performance will be presented. Example I ilustrates the workings of LZ78.

Example I

Let the sample string to be compressed be

$$\texttt{Sample0} = x_0^{33} = \texttt{aacabadababaacadabacabadadababaaaba}$$

The quaternary source alphabet is $\mathcal{A} = \{\texttt{a},\texttt{b},\texttt{c},\texttt{d}\}$. The sequence $\{\mathcal{L}_i : i = 0, 14\}$ of sets obtained with the LZ78 procedure, the corresponding phrases and binary codewords obtained are next presented.

Step $i = 0$
$\mathbf{z}_0 = \texttt{aacabadababaacadabacabadadababaaaba}$
$\mathcal{L}_0 = \{\lambda\}$, $\mathcal{M}_0 = \mathcal{A}$
$\ell_0 = \lambda$, $\mathbf{m}_0 = \texttt{a}$
$\mathbf{s}_0 = \ell_0 \circ \mathbf{m}_0 = \texttt{a}$, $W_0 = 00$
Step $i = 1$
$\mathbf{s}_0, \mathbf{z}_1 = \texttt{a,acabadababaacadabacabadadababaaaba}$
$\mathcal{L}_1 = \{\texttt{a}\}$
$\ell_0 = \texttt{a}$, $\mathbf{m}_0 = \texttt{c}$
$\mathbf{s}_1 = \ell_1 \circ \mathbf{m}_1 = \texttt{a} \circ \texttt{c}$, $W_1 = 1\ 10$
Keep going like this will take us to
$\mathbf{s}_0^{13}\mathbf{z}_{14} = \texttt{a,ac,ab,ad,aba,b,aa,c,ada,ba,ca,bad,adab,abaa,aba}$
$\mathcal{L}_{14} = \{\ \texttt{a, ac, ab, ad, aba, b, aa, c, ada, ba, ca, bad, adab, abaa}\ \}$
$\mathbf{s}_{14} = \texttt{aba}\ \lambda,\ \ W_{14} = 0101\ \lambda$

3.2 A Less Greedy LZ78

We observe, in the plain LZ78 discussed on Section 3.1, that the set $\mathcal{L}_i$ is increasead by one element at each step i, i.e., $|\mathcal{L}_i| = |\mathcal{L}_{i-1}| + 1$. The dictionary $\mathcal{D}_i$ is built by transforming the tree corresponding to $\mathcal{L}_{i-1}$ into a complete tree having only terminal nodes and nodes with exactly $|\mathcal{A}|$ branches stemming from them. This greedy expansion of the set $\mathcal{L}_{i-1}$ seems to be one reason for the degraded performance of the LZ78 algorithm, as compared to other variations, such as LZW for instance. The variation introduced in this section (lg-LZ, in short), allows for a less-greedy expansion in order to get the dictionary $\mathcal{D}_i$. The longest string match is not found this time (lg-LZ), with respect to the dictionary $\mathcal{D}_i = \mathcal{L}_{i-1} \circ \mathcal{M}_{i-1}$ but, instead, with respect to the dictionary

$$\mathcal{D}_i = \mathcal{L}_{i-1} \cup \{\mathbf{s}_i \circ \mathcal{A}\}.$$

The dictionary $\mathcal{D}_i$ is now built by expanding the $\mathcal{L}_{i-1}$ tree by appending to the node corresponding to the path just selected as a longest match, the tree corresponding to the alphabet $\mathcal{A}$. The algorithm is stated next.

Algorithm lg-LZ

$i = 0$

$\mathbf{z}_0 = x_0^{N-1}$

$\mathcal{L}_0 = \mathcal{A}$, $\mathcal{M}_0 = \mathcal{A}$

$\mathbf{s}_0 = x_0$

$J_0 = \mathcal{X}[\mathbf{s}_0|\mathcal{A}]$, $B_0 = \phi_{\lceil \log |\mathcal{A}| \rceil}[J_0]$

$1 \leq i \leq t$

1. Update unparsed string
   $\mathbf{z}_i = \mathbf{z}_{i-1} - \mathbf{s}_{i-1}$
2. Find longest match $\mathbf{s}_i$ with respect to $\mathcal{D}_i = \mathcal{L}_{i-1} \cup \{\mathbf{s}_{i-1} \circ \mathcal{M}_{i-1}\}$
   $\mathbf{s}_i = \Pi[\mathbf{z}_i|\mathcal{D}_i]$,
3. $J_i = \mathcal{X}[\mathbf{s}_i|\mathcal{D}_i])$
4. $B_i = \phi_{\lceil \log |\mathcal{D}_i| \rceil}[J_i]$
5. Updating tree
   $\ell_{new} = \mathbf{s}_{i-1} \circ \mathcal{F}[\mathbf{s}_i]$
   if $|\ell_{new}| = |\mathbf{s}_i|$ and $\mathbf{s}_i \notin \mathcal{L}_{i-1}$ then $\ell_{new} = \mathbf{s}_i$
   $\mathcal{L}_i = \mathcal{L}_{i-1} \cup \{\ell_{new}\}$
   $\mathcal{M}_i = \mathcal{A}$

Also here we have $\mathbf{s}_i = \ell_i \circ \mathbf{m}_i$ with, possibly, $\mathbf{m}_i = \lambda$. The performances displayed on Table 2, obtained by computer simulation show instances where the lg-LZ performs better when compared to its counterpart LZW. The example presented next ilustrate the workings of the lg-LZ.

Example II

Let $x_0^{33} =$ `aacabadababaacadabacabadadababaaaba`. $\mathcal{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. The parsing that the procedure lg-LZ yields is

`a, ac, a, b, a, d, ab, aba, aca, da, ba, c, aba, da, dab, abaa, aba`

The compressed representation of x_0^{33} is a binary string with 72 bits — compression rate of 0.257

3.3 Lempel-Ziv-Welch Algorithm

The Lempel-Ziv-Welch procedure, popularly called LZW, is known to have a performance on the average 10% better then the plain LZ78 version. One aspect that makes the LZW different from LZ78 is that it works with a rule that build the dictionary $\mathcal{D}_i$ by appending only one node to the corresponding tree $\mathcal{L}_{i-1}$.

The following would be the description of the LZW algorithm.

Algorithm LZW

$i = 0$

$\mathbf{z}_0 = x_0^{N-1}$ $\mathcal{L}_0 = \mathcal{A}$,
$\mathcal{M}_0 = \{\lambda\}$ and
$\mathbf{s}_0 = x_0$. $\ell_0 = x_0$

$1 \leq i \leq t$

1. $\mathbf{z}_i = \mathbf{z}_{i-1} - \mathbf{s}_{i-1}$
2. Find longest match with respect to $\mathcal{D}_i = \mathcal{L}_{i-1} \cup \ell_{i-1} \circ \mathcal{M}_{i-1}$
   $\ell_i = \Pi[\mathbf{z}_i | \mathcal{L}_{i-1}]$,
   $\mathbf{s}_i = \Pi[\mathbf{z}_i | \mathcal{D}_i]$,
3. $J_i = \mathcal{X}[\mathbf{s}_i | \mathcal{D}_i]$
4. $\mathcal{L}_i = \mathcal{D}_i$
   $\mathcal{M}_i = \{\mathcal{F}[\mathbf{z}_i - \mathbf{s}_i]\}$
5. $B_i = \phi_{\lceil \log |\mathcal{D}_i| \rceil}[J_i]$

Example III

Consider again $\texttt{Sample0} = x_0^{33} = \texttt{aacabadababaacadabacabadadababaaaba}$ with $\mathcal{A} = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$. This sequence is parsed into 20 phrases as follows

```
a, a, c, a, b, a, d, ab, aba, ac, ad, aba, ca, ba, da, da,
ba, ba, aa, ba
```

and its compressed representation is a binary string with 81 bits — a compression rate of 0.289

4 Description of Double-tree Algorithms

In the previous section two known agorithms (LZ78 and LZW) and a simple variation of the former (lg-LZ) were stated within the framework of a double-tree string match. Each one of the algorithms produce a sequence of trees $\{\mathcal{L}_i\}_{i=0,t}$ and corresponding sequence of dictionaries $\{\mathcal{D}_i\}_{i=0,t}$ with a string match done with respect to each dictionary. The basic difference among the three algorithms relies in the manner in which the tree $\mathcal{L}_{i-1}$ is concatenated with the corresponding $\mathcal{M}_{i-1}$, to build the dictionary $\mathcal{D}_i$. Table 1 summarizes this aspect.

$$
\begin{array}{lrcl}
\text{LZ78:} & |\mathcal{D}_i| & = & |\mathcal{L}_{i-1} \circ \mathcal{M}_{i-1}| \\
 & & \leq & |\mathcal{L}_{i-1}||\mathcal{M}_{i-1}| \\
\text{lg-LZ:} & |\mathcal{D}_i| & = & |\mathcal{L}_{i-1} \cup \{\ell_i \circ \mathcal{A}\}| \\
 & & = & |\mathcal{L}_{i-1}| + |\mathcal{A}| \\
\text{LZW:} & |\mathcal{D}_i| & = & |\mathcal{L}_{i-1} \cup \ell_{i-1} \circ \mathcal{M}_{i-1}| \\
 & & = & |\mathcal{L}_{i-1}| + 1
\end{array}
$$

Table 1: Length of the dictionaries

A point which is common to the three algorithms so far discussed is that they all concatenate the set $\mathcal{L}_{i-1}$ with a depth one tree in order to build their dictionaries. It is quite natural at this point to ask whether there are procedures which performs more efficiently when the second dictionary tree is allowed to have depth greater than one. A double-tree string match algorithm, with a second tree having a more general structure is stated in this section. Allowing a more general structure for the second tree $\mathcal{M}_{i-1}$, enlarge the number of algorithm variations that can be stated. The search for string matches are now searches for double-matches — this imply that more general ways to search are possible and that the longest-match is not necessarily a concatenation of a string ℓ_i (which is the longest match with respect to the tree $\mathcal{L}_{i-1}$) with the string $\mathbf{m}_i$ (which is the longest match with respect to the tree $\mathcal{M}_{i-1}$). Now, in order to optimize the number $t+1$ of parses, the best strategy is to search for a concatenation $(\ell_i \circ \mathbf{m}_i)$ which among all double-matches, have the largest size $|\ell_i|+|\mathbf{m}_i|$. We have implemented one version of a double-match/double-tree procedure and analysed their performance by computer simulations. The algorithm, which will be, abreviatedly, referred to as dt-LZ, is presented next.

Algorithm dt-LZ

$i = 0$ (Initialization step)

- $\mathbf{z}_0 = x_0^{N-1}$
- $\mathcal{L}_0 = \mathcal{M}_0 = \mathcal{A}$
- $\mathbf{m}_0 = \Pi[\mathbf{z}_0|\mathcal{M}_0]$,
- $K_0 = \mathcal{X}[\mathbf{m}_0|\mathcal{M}_0]$;
- $C_0 = \phi_{\lceil \log |\mathcal{M}_0| \rceil}[K_0]$
- $\mathbf{z}_1 = \mathbf{z}_0 - \mathbf{m}_0$;
- $\mathcal{M}_0 = \mathcal{M}_0 \cup \{\mathbf{m}_0 \circ \mathcal{F}[\mathbf{z}_1]\}$

$1 \le i \le t$ (Generic step)

1. Segmentation:
   (a) $\ell_i = \Pi[\mathbf{z}_i|\mathcal{L}_{i-1}]$,
   $\mathbf{z}_{temp} = \mathbf{z}_i - \ell_i$,
   $\mathbf{m}_i = \Pi[\mathbf{z}_{temp}|\mathcal{M}_{i-1}]$,
   $\tau = |\ell_i| + |\mathbf{m}_i|$,
   $\mathbf{u} = \ell_i$.
   (b) i. $\mathbf{u} = \mathcal{S}[\mathbf{u}]$
   $\mathbf{z}_{temp} = \mathbf{z}_i - \mathbf{u}$
   $\mathbf{v} = \Pi[\mathbf{z}_{temp}|\mathcal{M}_{i-1}]$.
   ii. If $(|\mathbf{u}| + |\mathbf{v}| \ge \tau)$: $(\ell_i, \mathbf{m}_i) = (\mathbf{u}, \mathbf{v})$, $\tau = |\ell_i| + |\mathbf{m}_i|$.
   iii. If $|\mathbf{u}| > 0$ return to step (i).
   (c) $\mathbf{z}_i = (\mathbf{z}_i - \ell_i) - \mathbf{m}_i$
2. Update Dictionaries:
   $\mathcal{L}_i = \mathcal{L}_{i-1} \cup \{\ell_i \circ \mathcal{F}[\mathbf{m}_i]\}$
   $\mathcal{M}_i = \mathcal{M}_{i-1} \cup \{\mathbf{m}_i \circ \mathcal{F}[\mathbf{z}_i]\}$
3. Map to Integer
   $(J_i, K_i) = (\mathcal{X}[\ell_i|\mathcal{L}_{i-1}],\ \mathcal{X}[\mathbf{m}_i|\mathcal{M}_{i-1}])$

4. Integer Code:
$$(B_i, C_i) = (\phi_{\lceil \log |\mathcal{L}_{i-1}| \rceil}[J_i],\ \phi_{\lceil \log |\mathcal{M}_{i-1}| \rceil}[K_i])$$

Example IV

Let x_0^{33} = `aacabadababaacadabacabadadababaaaba`. $\mathcal{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. The parsing for the procedure dt-LZ yields is
`(-,a), (a,c), (a,b), (a,d), (a,ba), (b,aa), (c,a), (d,a), (ba,ca), (ba,da), (da,bab), (a,aa),(b,a).`
where we show the double-matches displayed in parenthesis.

5 Some Computer Simulation Results

The algorithms discussed have been implemented as computer programs which were used to compress some sample sequences. Although the performance of all these algorithms are optimum in the sense that their compression rate asymptotically converges to the entropy of the information source or to the Lempel-Ziv complexity of the individual sequence, they perform quite differently when finite sequences and the rate of convergence to the asymptotic optimum are considered. Table 2 displays some of the simulation results exhibiting the performance of the algorithms. We have not

Sequence (size)	LZW (size)	lg-LZ (size)	dt-LZ (size)
`Sample0` (280)	.289 (81)	.257 (72)	.311 (87)
`Sample1` (576)	.089 (51)	.099 (57)	.097 (56)
`Sample2` (544)	.077 (42)	.086 (47)	.103 (56)
`Sample3` (672)	.357 (240)	.371 (249)	.335 (225)
`Sample4` (256)	.258 (66)	.113 (29)	.320 (82)

Table 2: Compression rate of algorithms LZW, lg-LZ and dt-LZ (all sequence sizes, in parenthesis, are in bits)

presented results for the LZ78 algorithm. As the other versions this algorithm is asymptotically optimum but has an inferior perfomance as compared to the LZW. As it can be noticed from the results presented in Table 2 the behavior of the algorithms are sequence dependent. For some sequences the LZW can achieve a better result than the lg-LZ — this gain is basically due to the penalty paid by the lg-LZ for expanding the first tree with $\mathcal{A}$ nodes to build the dictionary, instead of the one node expansion done by the LZW. This gain in performance tend to disappear as the sequence length grows larger. Examining the line on Table 2 corresponding to `Sample4` one can see that the performance of lg-LZ can converge considerably fast

to the optimum, as compared to LZW, for certain types of sequences. These are sequences constructed to benefit the performance of lg-LZ (no such construction can be done, we conjecture, to benefit LZW).

Conclusion

We have proposed algorithms which are based on the idea of string matches with respect to two sets or, equivalently, string match with respect to two trees. Many implementations variations of these algorithms are possible — a double-string match with respect to two trees version (called dt-LZ) was implemented.

In our preliminary investigation we exam the behavior of these algorithms and analyse its performance by computer simulation. Also we stated the well known LZ78 algorithm [LZ78] in the framework of string match with respect to two trees, as well as the LZW [W84]. A simple modification of the LZ78 was also proposed (this was called lg-LZ).

It is our expectation that higher compression can be achieved with double-string match with respect to two trees procedures. This is based on the argument that the use of two trees allows the construction of concatenated trees with more general structures, leaving more room for optimizing the search. It is also based on results we have obtained with multiple-string matches algorithms [PFP99] — which achieve a better compression than single-matches ones. These multiple-string match algorithms are based on the double-tree idea yet the two trees involved in the process are kept equal.

The results presented in this work do not single out a definite better double-match/double-tree algorithm — if one can be found — but bring to our attention that there are many variations. Our investigations will be further pursued by examining other double-match/double-tree implementations. An extension of the multiple-match described in [PFP99] will also be sought.

References

[LZ78] Ziv, J., Lempel, A., "Compression of individual sequences via variable-rate coding," IEEE Trans. Inform. Theory, vol. IT-24, pp.530-536, Sep. 1978.

[W84] Welch, T. A., "A technique for high-performance data compression," Computer, vol. 17, pp.8-19, Jun. 1984.

[FPC98] Finamore, W. A., Pinho, M. S., Craizer, M., "A multi-string match algorithm for lossless data compression," Abstracts of Invited Lectures and Short Communications, 7^{th} International Colloquium on Numerical Analysis and Computer Sciences with Applications, p.39, Plovdiv, Bulgaria, Aug. 1998.

[PFP99] Pinho, M. S., Finamore, W. A., Pearlman, W. A., "Fast multi-match Lempel-Ziv," Proc. of IEEE Data Compression Conference, Snowbird, UT, April 1999.

[HR85] Hartman, A., Rodeh, M., "Optimal Parsing of Strings," Combinatorial Algorithms on Words, Springer-Verlag, A. Apostolico & Z. Galil, editors, pp. 155-167, 1985.

[S97] Savari, S. A., "Redundancy of Lempel-Ziv incremental parsing rule," IEEE Trans. Inform. Theory, vol. IT-43, pp.9-21, Jan. 1997.