A Kolmogorov Complexity–based Genetic Programming Tool for String Compression

I. De Falco, A. Iazzetta and E. Tarantino
IRSIP–CNR
Via P. Castellino, 111 80131 Naples, Italy
email: defalco.i@irsip.na.cnr.it
+39 081 5904222

A. Della Cioppa
Dep. of Electric and Inf. Eng.
Università di Salerno
Fisciano, SA, Italy
email: dean@unina.it
+39 089 964274

G. Trautteur
Department of Physics
University of Naples "Federico II"
Mostra d'Oltremare, Naples, Italy
email: Giuseppe.Trautteur@na.infn.it
+39 081 7253413

Abstract

By following the guidelines set in one of our previous papers, in this paper we face the problem of Kolmogorov complexity estimate for binary strings by making use of a Genetic Programming approach. This consists in evolving a population of Lisp programs looking for the "optimal" program that generates a given string. By taking into account several target binary strings belonging to different formal languages, we show the effectiveness of our approach in obtaining an approximation from the above of the Kolmogorov complexity function. Moreover, the adequate choice of "similar" target strings allows our system to show very interesting computational strategies. Experimental results indicate that our tool achieves promising compression rates for binary strings belonging to formal languages. Furthermore, even for more complicated strings our method can work, provided that some degree of loss is accepted. These results constitute a first step in using Kolmogorov complexity for string compression.

1 Introduction

The complexity of a computation is a measurement of computational requirements needed to execute it. One of the most important approaches to static complexity (related to program structure and size) is due to Kolmogorov. He [Kol65] and Chaitin [Cha66] developed the notion of algorithmic complexity independently of one another and of Solomonoff's notion [Sol64] of algorithmic probability. In his paper Kolmogorov described an algorithmic approach to information theory and he defined the complexity of a finite object. Though Kolmogorov complexity (KC) seems a heavily theoretical subject, it may indeed have practical applications in information representation, compression and transmission, both lossless and lossy. As an example, it seems to fit well the MPEG4 environment. Unfortunately, despite of its promising appeal, until now KC has not been used to achieve practical results in this field. The main problem is the impossibility in evaluating KC for a given string, since this is not computable.

This paper deals with a Genetic Programming approach to KC estimate for binary strings, by means of an approximation from the above of the Kolmogorov complexity function. We were the first to use this approach ([Con97]). The present paper constitutes a more formal and rigorous generalisation of the contents of that paper. New strings are taken into account, and new problems are presented. Moreover, compression is explicitly faced. This paper is organized as follows: Section 2 provides the reader with the historical framework KC was originated in, and resumes its basic features (for their more detailed description the reader may see our above referenced paper or [Gam99] or [Wal99]). Section 3 gives details about our GP approach. In Section 4 our experiments and the related results are reported. Finally, our conclusions and foreseen improvements follow.

2 Kolmogorov Complexity

In 1965 Kolmogorov published a paper ([Kol65]) on the algorithmic approach to information theory where he defined the complexity of a finite object and also showed how to measure the amount of mutual information of one (finite) object about another. Intuitively, it was clear to everyone that there are 'simple' objects and 'complex' ones. The problem was the diversity of the ways of describing objects: an object can have a 'simple' description (i.e. short) in one language but not in another. The discovery made independently by him and by Solomonoff was that with the help of the theory of algorithms it is possible to restrict this arbitrariness and define complexity as an invariant concept.

Since then the study and applications of the complexity of a body of information have followed at least three different streams, springing from three independent views of the concept. The first stream was that initiated by Kolmogorov with important later developments by Martin-Löf [Lof66] and Chaitin [Cha66]. The second stream (chronologically the first) springs from the work of Solomonoff, while the third was introduced by Wallace and Boulton [Wal68], with a similar but independent development by Rissanen [Ris78]. The motivations behind their work were completely different: for example Solomonoff, on the one hand, was interested in inductive inference and artificial intelligence, with reference to the problem of sequential prediction and to unordered data prediction. Kolmogorov, on the other hand, was interested in the foundations of probability theory and, also, of information theory.

We will make reference, in the following of this paper, to the first stream, i.e. that by Kolmogorov. In it the body of information is usually assumed to be a finite string of binary digits x. We will write $l(x)$ for the length of x. The theory of Kolmogorov complexity is based on the introduction, by Alan Turing in 1936, of the universal Turing machine (U). Turing found that there exists one Turing machine which can simulate any other Turing machine.

Definition 1. The *Kolmogorov complexity* or *algorithmic complexity* of x, i.e. $K(x)$, with respect to some specified universal Turing machine U may be defined as the length l of the shortest binary string I (representing a program), which, when supplied to U, causes U to output x and stop.

This program I is said to be *canonical* for x.

The universality of U ensures that for all strings x the difference between the complexities of x with respect to two different U's V and Z is bounded above by a constant independent of x, namely the length of the longer of the programs required to make V imitate Z and to make Z imitate V. Thus $K(x)$ is absolute in the sense of being independent of the programming language .

It is evident that there are strings which can be described by programs much shorter than themselves, but the majority of strings can hardly be compressed at all. For every l there are 2^l binary strings of length l, but only $2^l - 1$ possible shorter descriptions. Therefore, there is at least one binary string x of length l such that $K(x) \geq l$. We call such strings 'incompressible'. For every constant c, a string x is c–incompressible if $K(x) \geq l(x) - c$. Strings that are incompressible (say, c–incompressible with small c) are patternless, since a pattern could be used to reduce the description length. Intuitively, we think of such patternless sequences as being random, so that 'random sequence' is used synonymously with 'incompressible sequence'. For every constant $c \geq 1$ the majority of all strings of length l (with $l > c$) is c–incompressible.

An important theorem states that $K(x)$ is not partial recursive, i.e. given a string x it is not possible to algorithmically compute its canonical program. So the problem of computing the K function is unsolvable. Nevertheless, another theorem proves that there exists a total recursive function $\phi(t, x)$, monotonically non-increasing in t, and such that

$$\lim_{t \to \infty} \phi(t, x) = K(x) \ .$$

This function $\phi(t, x)$ provides us with an approximation from above to the non–recursive function K.

An attempt at approximating the function K may consist in evaluating the function $\phi(t, x)$. To do so one needs: a universal computer in order to run the programs in turn; a program generator providing syntactically correct programs for the chosen universal computer; a supervising routine [Con97].

The universal computer chosen is the TOY Lisp written by Chaitin [Cha94]. This implementation of Lisp has been chosen because it works on very short strings, allows an upper limitation on the number of evaluation steps and, finally, is extremely fast. The primitive functions of this Lisp are listed in Table 1 [Cha94]. As can be seen, the functions, whose names appear in the first column, are represented by single literals. Their values are displayed in the last column. It should be noted that the last two functions, unusual in standard Lisp implementations, have been defined by Chaitin and are particularly valuable for our kind of application.

The program generator is based on a grammar written for M–expressions, i.e. the programs for the TOY Lisp. Besides generating syntactically correct programs this module is endowed with some heuristics in order to generate "sensible" programs. In this grammar we have as terminals the atoms $0, 1, ()$ and j, k, s, t, v, w, y, z for the newly defined atoms. The symbol set of the functions is composed by the symbols of the primitive functions of the TOY Lisp and by the following symbols $a(g), b(h), c(i), d(lm), e(no), f(pqr)$ for the newly defined functions, where the symbols in the brackets represent the variables.

Since in [Con97] we ran into intractable space and time difficulties while performing an initial almost brute–force attempt, and since the fundamental problem was one of searching the program space we, very naturally, turned to evolutionary methods. Precisely, instead of approximating the function $K(x)$ for a generic x, we launched a search for pairs of strings x and their generating programs u, while minimizing the length of the latter. This might be a first step toward fulfilling the compression goal alluded to in Section 2 of [Nor96].

Table 1. The TOY Lisp atoms and primitive functions. (From [Cha94].)

Atom	**Symbol**		
Empty	()		
False	0		
True	1		
Function	**Symbol**	**No. of args**	**Value**
Quote	'	1	$'(xyz) \to (xyz)$
Atom	.	1	$.x \to 1 \quad .'(x) \to 0$
Equal	=	2	$=xx \to 1 \quad =xy \to 0$
Car	+	1	$+'(xyz) \to x \quad +x \to x$
Cdr	−	1	$-'(xyz) \to (yz) \quad -x \to x$
Cons	∗	2	$*x'(yz) \to (xyz) \quad *xy \to x$
If–then–else	/	3	$/1xy \to x \quad /0xy \to y$
Eval	!	1	$!x \to$ evaluate x
Append	^	2	$\hat{}'(xy)'(zk) \to (xyzk)$
Define	&	2	$\&xy \to x$ is y $\quad \&(yx)z \to y$ is $\&(x)z$
Try	?	3	$?yxz \to$ evaluate x times y with bits z
Let	:	3	$:xy\ z \to ('\&(x)z\ y) \quad :(yx)z\ k \to ('\&(y)k'\&(x)z)$

3 The GP Approach

GP [Cra85, Koz92, Koz94] is well suited for our application since our aim is to search for pairs (u, x) such that a program u in TOY Lisp generates a binary string x. The number of programs u able to produce a given string x is – even obviously limiting oneselves to programs of length not much larger than the length of the string – enormous and the program landscape has a very large number of suboptima. So, our approach consists in fixing a given target string x and in using a GP algorithm in order to search the program space looking for optimal generating programs.

Following [Koz92], the genotypes are TOY Lisp programs encoded as tree structures with no fixed bound on the size. The nodes of the trees are either atoms or functions of our grammar. The phenotypes are the strings x, obtained by the evaluation of the genotypes u in the TOY Lisp environment.

The fitness function takes into account several needs: firstly, the length of the string x evaluated by the program u (the closer to the target length the better the fitness), then a "Hamming–like" distance from the target (the lower the distance the better the fitness), and the genotype (i.e. the program) length (the lower its length the better the fitness). The search for the shortest program is thus fully reduced to a minimization problem, through the introduction of a fitness function. Furthermore, we have tried to favor programs which use new (i.e. "user"–defined) functions, because only by using these functions is it possible to achieve during the evolution shorter and shorter programs which take advantage of the pattern underlying the target string. This has been effected through the use of a fitness function which penalizes the programs that do not use new functions. A further penalty is used to take care of programs which do not stop within the limit on the number of evaluation steps allowed by the TOY Lisp. These penalties are obtained by assigning a high fitness value to the relative phenotype. Hence, except for the above penalties, we have written our fitness function as the sum of terms each expressing one of the above needs, and we have tried to balance all of these different and sometimes contrasting needs by means of weight coefficients. Then, we have:

$$\mathcal{F}(u) = a \sum_{i=1}^{k} d(x_i, \tau_i) + b|l(x) - l(\tau)| + c(l(u) - l(x))$$

where a, b and c are constants, u is the program that computes in TOY Lisp $x = x_1 \dots x_n$, $\tau = \tau_1 \dots \tau_m$ is the target string, $k = \min(l(x), l(\tau))$, and

$$d(x_i, \tau_i) = \begin{cases} 0 & \text{if } x_i = \tau_i \\ 1 & \text{if } x_i \neq \tau_i,\ x_i \in \{0,1\} \\ 2 & \text{if } x_i \neq \tau_i,\ x_i \notin \{0,1\} \end{cases}$$

represents a "Hamming–like" distance for the strings on the alphabet of the terminal symbols.

The program generator provides the starting randomly generated population by using a grammar that ensures the syntactic correctness of the programs. The new elements in the population are generated combining pairs of programs by means of a tree–crossover operator that ensures the new programs to be syntactically correct. We have assumed that this operator can take place in function nodes only. A node is randomly selected in each tree representing the parents, then crossover swaps the nodes and their relative subtrees from one parent to the other.

Mutation can be applied to any point in the string, so it can operate on either a function node or an atom node. The following kinds of mutation are allowed:

- *point mutation*: a node in the tree is randomly selected; if it is a function, either primitive or defined, it is replaced by a new function. Depending on both the functions, in some cases, the mutation determines also the replacement of the relative subtree with a new one randomly generated. If the chosen node is an atom it is simply replaced by a new atom;

- *insertion*: a new random node is inserted in a random point, along with the relative subtree if it is necessary;

- *deletion*: a node in the tree is selected and deleted in a way that ensures the syntactic correctness.

To grant the effectiveness of the mutation when applied to functions, we have decided to give to each function the maximum number of arguments occurring in the primitive functions constituting the TOY Lisp, i.e. three, in order to avoid syntactic problems when the tree–crossover and the mutation operators try to replace a function with another one having a different number of arguments. Any function will actually use the number of arguments it needs. This choice is the same made by W. Banzhaf in [Ban93] for the Pedestrian Genetic Programming. Finally, the classical proportional and the truncation selection mechanisms have been considered; the latter has been chosen because it accelerates the convergence time to a suboptimum. To this end, moreover, we have also used a 1–elitism strategy with both mechanisms.

4 Experimental Findings

4.1 Languages Considered

The application of Kolmogorov Complexity to formal languages is not new. In the present paper, differently from other approaches in literature [Li93], target strings are represented by instances of formal languages. This choice allows us to avoid taking pseudo–random sequences into account and is due to to our wish to show the goodness of the tool at finding generating programs, when these are *a priori* known.

The classes of languages considered for our experiments are summarized in Table 2. It is organized following the hierarchy introduced by Noam Chomsky, who defined these classes as models of natural languages, and comprises four types of languages and their associated grammars. It should be noted that it is possible to prove that, except for the matter of the empty string, these languages form a strict hierarchy, i.e. the type–i language properly includes the type–$(i+1)$ language for $i = 0, 1, 2$.

4.2 The GP Implementation

As concerns the GP parameters, we have made several trials with different values for the population size, the truncation rate, the crossover probability, the mutation probability, so as to determine a good parameter set for the problem at hand. As a consequence of these preliminary runs the following parameters have been employed: population size equal to 300 individuals, truncation rate in the range $15 \div 40\%$, crossover probability in the range $0.5 \div 0.8$ and mutation performed systematically on one random point in the program string. As regards the weight coefficients of the fitness function, we have chosen $a = 20.0$, $b = 30.0$ and $c = 2.0$. Finally, the termination criterion for the GP algorithm is only related to the maximum number of generations allowed.

As regards the experimentation, several tests (in the range $20 \div 30$) for each problem have been performed and the best results of these executions are discussed in the following. It must be remarked here that the tool has proved to be robust: in fact, the different trials for a same target string have yielded generating programs which are very similar. They differ mainly for the presence of introns, so that some easy post–execution manipulations allow to obtain the same program in almost all runs. Moreover, the number of generations needed to achieve such results is not greater than 100.

4.3 Results

Based on the above, the system has been initially tested against strings that are instances of the *regular languages*, as L_{r_1} and L_{r_2} in Table 2. Such strings are characterized by a nucleus which is repeated a given number of times to create the target string.

The target strings considered have been those with $j = 19$, $j = 30$, $j = 47$, $j = 64$ and $j = 77$. We have obtained the following pairs for these instances of L_{r_1} and L_{r_2}:

$$\begin{array}{ll}
u =: (ag)\hat{}gg{*}1a{*}1aaa\text{'}(1) & x = 1^{19} \\
u =: (ag)\hat{}gga{-}aaa\text{'}(11) & x = 1^{30} \\
u =: (ag)\hat{}gg{-}aaaa\text{'}(111) & x = 1^{47} \\
u =: (ag)\hat{}ggaaaaaa\text{'}(1) & x = 1^{64} \\
u =: (ag)\hat{}gg{*}1aa{*}1a{*}1aaa\text{'}(1) & x = 1^{77}
\end{array}$$

$$\begin{array}{ll}
u =: (ag)\hat{}gg\hat{}a\hat{}aaa\text{'}(10)\text{'}(10)\text{'}(10) & x = (10)^{19} \\
u =: (ag)\hat{}gga{-}{-}aaaa\text{'}(10) & x = (10)^{30} \\
u =: (ag)\hat{}gg{-}{-}aaaa{-}{-}aa\text{'}(10) & x = (10)^{47} \\
u =: (ag)\hat{}ggaaaaaa\text{'}(10) & x = (10)^{64} \\
u =: (ag)\hat{}gg{-}{-}\hat{}aaaaaa\text{'}(10)a{-}{-}aaa\text{'}(10) & x = (10)^{77}
\end{array}$$

In all the cases the GP algorithm has defined, using *Let*, a new function that concatenates two lists of atoms and uses such a function several times for the construction of the string. Moreover, depending on the

Table 2. The Chomsky hierarchy and the languages employed.

Grammars	Languages	Examples
Type–0	Recursively enumerable	Any computable function
Type–1	Context–sensitive	$L_{cs_1} = \{1^j 0^j 1^j \mid j \in \mathbb{N}, j \geq 1\}$
		$L_{cs_2} = \{1^{2^j} \mid j \in \mathbb{N}, j \geq 1\}$
		$L_{cs_3} = \{1^{2^j+1} \mid j \in \mathbb{N}, j \geq 1\}$
		$L_{cs_4} = \{1^{2^j-1} \mid j \in \mathbb{N}, j \geq 1\}$
		$L_{cs_5} = \{1^{2^j} 0^{2^j} \mid j \in \mathbb{N}, j \geq 1\}$
		$L_{cs_6} = \{1^{2^j+1} 0^{2^j+1} \mid j \in \mathbb{N}, j \geq 1\}$
		$L_{cs_7} = \{1^{2^j-1} 0^{2^j-1} \mid j \in \mathbb{N}, j \geq 1\}$
		$L_{cs_8} = \{1^{2^j} 0 1^{2^j} \mid j \in \mathbb{N}, j \geq 1\}$
		$L_{cs_9} = \{1^j \mid j \text{ is a prime}\}$
Type–2	Context–free	$L_{cf_1} = \{1^j 0^j \mid j \in \mathbb{N}, j \geq 1\}$
		$L_{cf_2} = \{1^j 0 1^j \mid j \in \mathbb{N}, j \geq 1\}$
Type–3	Regular	$L_{r_1} = \{1^j \mid j \in \mathbb{N}, j \geq 1\}$
		$L_{r_2} = \{(10)^j \mid j \in \mathbb{N}, j \geq 1\}$

length of the target string, the algorithm employs the function *Cons* for the insertion of an atom or performs a *Cdr* for the complete generation of the target string.

Afterwards, the program has been successfully tested against more complex strings as those that are instances of the *context–free languages* L_{cf_1} and L_{cf_2}. Setting j equal to 19, 30, 47, 64 and 77 we have obtained the following pairs for L_{cf_1} and L_{cf_2}:

$$u =: (ag)\hat{}gg\hat{}\ast 1a\ast 1aaa'(1)\ast 0a\ast 0aaa'(0) \quad x = 1^{19}0^{19}$$
$$u =: (ag)\hat{}gg\hat{}a-aaa'(11)a-aaa'(00) \quad x = 1^{30}0^{30}$$
$$u =: (ag)\hat{}gg\hat{}-aaaa'(111)-aaaa'(000) \quad x = 1^{47}0^{47}$$
$$u =: (ag)\hat{}gg\hat{}aaaaaa'(1)aaaaaa'(0) \quad x = 1^{64}0^{64}$$
$$u =: (ag)\hat{}gg\hat{}\ast 1aa\ast 1a\ast 1aaa'(1)$$
$$\ast 0aa\ast 0a\ast 0aaa'(0) \quad x = 1^{77}0^{77}$$

$$u =: (ag)\hat{}gg-a\ast 0\ast 1a\ast 1aaa'(1) \quad x = 1^{19}01^{19}$$
$$u =: (ag)\hat{}gg-a\ast 0a-aaa'(11) \quad x = 1^{30}01^{30}$$
$$u =: (ag)\hat{}gg-a\ast 0-aaaa'(111) \quad x = 1^{47}01^{47}$$
$$u =: (ag)\hat{}gg-a\ast 0aaaaaa'(1) \quad x = 1^{64}01^{64}$$
$$u =: (ag)\hat{}gg-a\ast 0\ast 1aa\ast 1a\ast 1aaa'(1) \quad x = 1^{77}01^{77}$$

It can be pointed out that in the second case the GP algorithm has defined, using *Let*, a new function that concatenates two lists and uses such a function several times, along with the *Cons* and *Cdr* functions for the construction of the string 01^j. Successively, the algorithm employs the function *Cons* for the insertion of the atom 0 and finally performs a *Cdr* for the complete generation of the target string.

Following the Chomsky hierarchy, we have tested the system on target strings belonging to *context–sensitive languages*, i.e. strings are instances of the languages L_{cs_i} $\forall i \in \{1, \ldots, 9\}$.

Setting j equal to 19, 30, 47, 64 and 77 for L_{cs_1}, the system has obtained the following pairs:

$$u =: (ag)\hat{}gg\hat{}\hat{}\ast 1a\ast 1aaa'(1)\ast 0a\ast 0aaa'(0)\ast 1a\ast 1aaa'(1)$$
$$x = 1^{19}0^{19}1^{19}$$
$$u =: (ag)\hat{}gg\hat{}\hat{}a-aaa'(11)a-aaa'(00)a-aaa'(11)$$
$$x = 1^{30}0^{30}1^{30}$$
$$u =: (ag)\hat{}gg\hat{}\hat{}-aaaa'(111)-aaaa'(000)-aaaa'(111)$$
$$x = 1^{47}0^{47}1^{47}$$
$$u =: (ag)\hat{}gg\hat{}\hat{}aaaaaa'(1)aaaaaa'(0)aaaaaa'(1)$$
$$x = 1^{64}0^{64}1^{64}$$
$$u =: (ag)\hat{}gg\hat{}\hat{}\ast 1aa\ast 1a\ast 1aaa'(1)\ast 0aa\ast 0a\ast 0aaa'(0)$$
$$\ast 1aa\ast 1a\ast 1aaa'(1)$$
$$x = 1^{77}0^{77}1^{77}$$

It should be noted that in this case the GP algorithm performs an action similar to that effected for the construction of the strings belonging to the *context–free* languages considered.

Setting j equal to $3, 4, 5, 6$ and 7, we have obtained the following pairs for L_{cs_2}, L_{cs_3} and L_{cs_4} respectively:

$$\begin{array}{ll} u =: (ag)\hat{}ggaaa'(1) & x = 1^{8} \\ u =: (ag)\hat{}ggaaaa'(1) & x = 1^{16} \\ u =: (ag)\hat{}ggaaaaa'(1) & x = 1^{32} \\ u =: (ag)\hat{}ggaaaaaa'(1) & x = 1^{64} \\ u =: (ag)\hat{}ggaaaaaaa'(1) & x = 1^{128} \end{array}$$

$$\begin{array}{ll} u =: (ag)\hat{}gg{*}1aaa'(1) & x = 1^{9} \\ u =: (ag)\hat{}gg{*}1aaaa'(1) & x = 1^{17} \\ u =: (ag)\hat{}gg{*}1aaaaa'(1) & x = 1^{33} \\ u =: (ag)\hat{}gg{*}1aaaaaa'(1) & x = 1^{65} \\ u =: (ag)\hat{}gg{*}1aaaaaaa'(1) & x = 1^{129} \end{array}$$

$$\begin{array}{ll} u =: (ag)\hat{}gg{-}aaa'(1) & x = 1^{7} \\ u =: (ag)\hat{}gg{-}aaaa'(1) & x = 1^{15} \\ u =: (ag)\hat{}gg{-}aaaaa'(1) & x = 1^{31} \\ u =: (ag)\hat{}gg{-}aaaaaa'(1) & x = 1^{63} \\ u =: (ag)\hat{}gg{-}aaaaaaa'(1) & x = 1^{127} \end{array}$$

For the last three languages considered, it is possible to note the emergence of a well defined structure. In fact, the programs that generate strings belonging to such languages assume the following form:

$$\begin{array}{ll} u =: (ag)\hat{}gga^{j}\text{'}(1) & x = 1^{2^j} \\ u =: (ag)\hat{}gg{*}1a^{j}\text{'}(1) & x = 1^{2^j+1} \\ u =: (ag)\hat{}gg{-}a^{j}\text{'}(1) & x = 1^{2^j-1} \end{array}$$

for any $j \in \mathbb{N}$ and $j \geq 1$. With the same values for j, we have obtained the following results for L_{cs_5}, L_{cs_6}, L_{cs_7} and L_{cs_8}:

$$\begin{array}{ll} u =: (ag)\hat{}gg\hat{}aaa'(1)aaa'(0) & x = 1^{8}0^{8} \\ u =: (ag)\hat{}gg\hat{}aaaa'(1)aaaa'(0) & x = 1^{16}0^{16} \\ u =: (ag)\hat{}gg\hat{}aaaaa'(1)aaaaa'(0) & x = 1^{32}0^{32} \\ u =: (ag)\hat{}gg\hat{}aaaaaa'(1)aaaaaa'(0) & x = 1^{64}0^{64} \\ u =: (ag)\hat{}gg\hat{}aaaaaaa'(1)aaaaaaa'(0) & x = 1^{128}0^{128} \end{array}$$

$$\begin{array}{ll} u =: (ag)\hat{}gg\hat{}{*}1aaa'(1){*}0aaa'(0) & x = 1^{9}0^{9} \\ u =: (ag)\hat{}gg\hat{}{*}1aaaa'(1){*}0aaaa'(0) & x = 1^{17}0^{17} \\ u =: (ag)\hat{}gg\hat{}{*}1aaaaa'(1){*}0aaaaa'(0) & x = 1^{33}0^{33} \\ u =: (ag)\hat{}gg\hat{}{*}1aaaaaa'(1){*}0aaaaaa'(0) & x = 1^{65}0^{65} \\ u =: (ag)\hat{}gg\hat{}{*}1aaaaaaa'(1){*}0aaaaaaa'(0) & x = 1^{129}0^{129} \end{array}$$

$$\begin{array}{ll} u =: (ag)\hat{}gg\hat{}{-}aaa'(1){-}aaa'(0) & x = 1^{7}0^{7} \\ u =: (ag)\hat{}gg\hat{}{-}aaaa'(1){-}aaaa'(0) & x = 1^{15}0^{15} \\ u =: (ag)\hat{}gg\hat{}{-}aaaaa'(1){-}aaaaa'(0) & x = 1^{31}0^{31} \\ u =: (ag)\hat{}gg\hat{}{-}aaaaaa'(1){-}aaaaaa'(0) & x = 1^{63}0^{63} \\ u =: (ag)\hat{}gg\hat{}{-}aaaaaaa'(1){-}aaaaaaa'(0) & x = 1^{127}0^{127} \end{array}$$

$$\begin{array}{ll} u =: (ag)\hat{}gg{-}a{*}0aaa'(1) & x = 1^{8}01^{8} \\ u =: (ag)\hat{}gg{-}a{*}0aaaa'(1) & x = 1^{16}01^{16} \\ u =: (ag)\hat{}gg{-}a{*}0aaaaa'(1) & x = 1^{32}01^{32} \\ u =: (ag)\hat{}gg{-}a{*}0aaaaaa'(1) & x = 1^{64}01^{64} \\ u =: (ag)\hat{}gg{-}a{*}0aaaaaaa'(1) & x = 1^{128}01^{128} \end{array}$$

Also for the last four languages considered, the emergence of a well defined structure can be noted. In fact, the programs that generate strings belonging to such languages assume the following form:

$$\begin{array}{ll} u =: (ag)\hat{}gg\hat{}a^{j}\text{'}(1)a^{j}\text{'}(0) & x = 1^{2^j}0^{2^j} \\ u =: (ag)\hat{}gg\hat{}{*}1a^{j}\text{'}(1){*}0a^{j}\text{'}(0) & x = 1^{2^j+1}0^{2^j+1} \\ u =: (ag)\hat{}gg\hat{}{-}a^{j}\text{'}(1){-}a^{j}\text{'}(0) & x = 1^{2^j-1}0^{2^j-1} \\ u =: (ag)\hat{}gg{-}a{*}0a^{j}\text{'}(1) & x = 1^{2^j}01^{2^j} \end{array}$$

for any $j \in \mathbb{N}$ and $j \geq 1$. As regards L_{cs_9}, setting j equal to 7, 11, 13, and 19, we obtained the following pairs:

$$\begin{array}{ll} u =: (ag)\hat{}gg{-}aaa'(1) & x = 1^{7} \\ u =: (ag)\hat{}gg{-}aa'(111) & x = 1^{11} \\ u =: (ag)\hat{}gg{*}1aa'(111) & x = 1^{13} \\ u =: (ag)\hat{}gg{*}1a{*}1aa'(11) & x = 1^{19} \end{array}$$

Finally, the system has been tested on target strings as the following:

$$\alpha : 1^2 0^2 1^4 0^4 \ldots 1^{2^j} 0^{2^j} \text{ with } j \in \mathbb{N}$$

By setting $j = 2$ and $j = 3$, we have obtained the following pairs:

$$\begin{array}{l} u =: (ag)\hat{}gg\hat{}\hat{}\hat{}a'(1)a'(0)aa'(1)aa'(0) \\ x = 110011110000 \\ u =: (ag)\hat{}gg\hat{}\hat{}\hat{}\hat{}\hat{}a'(1)a'(0)aa'(1)aa'(0)aaa'(1)aaa'(0) \\ x = 1100111100001111111100000000 \end{array}$$

Similar pairs (too lengthy to be reproduced here) have been obtained for $j = 4$ and $j = 5$. This results in a general structure as:

$$\begin{array}{l} u =: (ag)\hat{}gg\hat{}^{((2\cdot j)-1)}a'(1)a'(0)a^{2}\text{'}(1)a^{2}\text{'}(0)\ldots a^{j}\text{'}(1)a^{j}\text{'}(0) \\ x = 1^2 0^2 1^4 0^4 \ldots 1^{2^j} 0^{2^j} \end{array}$$

4.4 Compression Considerations

For all the previous strings the program has turned out to be able to write short M–expressions which obtain the target string by concatenating several times the repeating nucleus, thus demonstrating the string simplicity. The compression ratio $r = l(x)/l(u)$ can be simply evaluated by counting the number of symbols in the program u and in the relative output string x.

It can be simply noted that for L_{cs_2}, L_{cs_3} and L_{cs_4}, while we increase j by 1, (thus about doubling the string length) the length of the generating programs simply increases by 1. Similarly, for L_{cs_5}, L_{cs_6} and L_{cs_7}, while we increase j by 1 (thus about doubling $l(x)$) $l(u)$ increases by two. For L_{cs_8}, while $l(x)$ almost doubles, $u(x)$ increases by 1.

This seems a very intriguing feature. It appears that the Genetic Programming has discovered the

j	3	4	5	6	7	8	9
$l(x)$	18	34	66	130	258	514	1026
$l(u)$	27	29	31	33	35	37	39
r	0.666	1.172	2.129	3.939	7.371	13.891	26.307

Table 3. Values of r as a function of j for L_{cs_6}.

logarithm. To investigate more explicitly this fact, we have taken into account two strings belonging to the above languages, namely L_{cs_6} and α. For the former language we have performed further experiments as a function of j by taking into account for j values of 8 and 9. For it we have obtained the values of r as a function of j reported in Table 3.

This shows that, since the system has "discovered" the regularity underlying the structure of the strings, the compression ratio increases as j does. In fact, when passing from a value of j to the next, $l(u)$ increases by two symbols, while $l(x)$ almost doubles. The same results are reported in Fig. 1(a) which shows a linear dependence of r on $l(x)$, and a logarithmic one on j.

For the string α we have performed more runs setting for j values from 6 to 8.

We have obtained the values of r as a function of j reported in Table 4:

Fig. 1(b) reports the dependence on j and on $l(x)$ for this string as well. In this case we obtain a quasi–linear curve as a function of $l(x)$, and a logarithmic one as a function of j.

These results are of extremely high interest from the point of view of string compression in information transmission. In fact, this means that the generating program for a short sequence and that for a very long one are of comparable length, provided that the two strings have a similar structure, which should be the case when they belong to the same language. Therefore, compression becomes more effective when the string becomes longer.

Unfortunately most strings are not Kolmogorov–compressible, so all of the above holds true for few strings only. However, if we accept some degree of loss in information [Sow97], we can hope that, given a string x_u not belonging to a simply structured formal language, yet close to one of them (say L_w), our tool gives as the generating program for x_u that for L_w; in fact this can produce a string close to x_u.

j	2	3	4	5	6	7	8
$l(x)$	12	28	60	124	252	508	1020
$l(u)$	33	49	67	87	109	133	159
r	0.363	0.571	0.895	1.425	2.311	3.819	6.415

Table 4. Values of r as a function of j for α.

To ascertain the capability of our system in this situation, we have performed a set of experiments by taking into account the string L_{cf_1} with $j = 77$. We have randomly swapped one bit, then two bits, then three bits, and so on. Of course, the higher the number of mutated bits, the more difficult is it that the generating program for L_{cf_1} can be useful for the modified string as well. Nonetheless the system has been able to obtain the same program. Namely, the original program has always been found for strings with up to nine modified bits, while other programs appear when ten or more bits are changed. Since, as shown above, the canonical program has a length $l(u) \leq 40$ we have obtained in all of these successful cases a compression ratio $r = 144/40 = 3.6$, with an error ranging from $1/144 = 0.69\%$ to $9/144 = 6.25\%$.

5 Conclusion

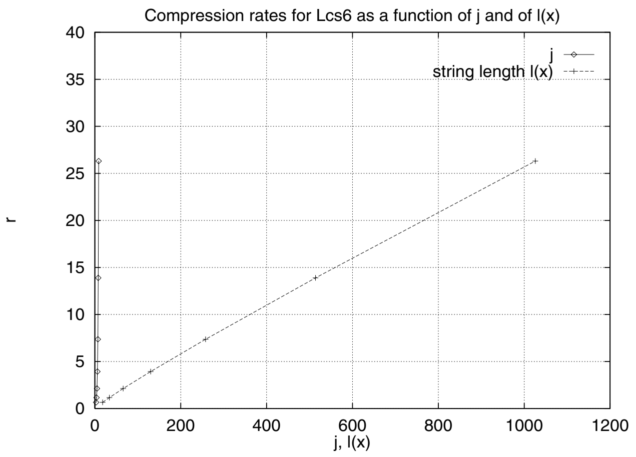
For all of the considered Chomsky classes of languages our GP–based system has proved capable of finding the LISP programs generating given instances of those languages. More interestingly, the generating programs for strings showing similar structures are similar as well, and often differ only by the number of times a function is applied.

Experiments have provided us with interesting compression rates for the classes of strings taken into account. Furthermore, for many such languages the compression ratio increases linearly or about linearly as the string length does, and depends logarithmically on j. This is of high interest when the strings become longer.

Moreover, we have shown that the same structure of generating program works both for the strings belonging to a given language and for strings showing a sufficiently high degree of similarity to the former ones. Thus, if we accept some degree of loss of information in compression, our method can be of high interest.

Since we have only performed a single characterization of the pairs (u, x) (i.e. we search only for the shortest program u generating x), our prospects of future work will concern the use of a GP approach to try to effect a characterization of whole classes of strings so as to make contacts with the formal languages theory. To this end, a GP methodology, able to generate a whole population of pairs (u, x) such that the target strings x fall into a given formal language, should be realized.

We are interested in making use of our approach to further investigate the idea of lossy approximations to given strings as well. Namely, we wish to determine the minimum similarity degree between a new string



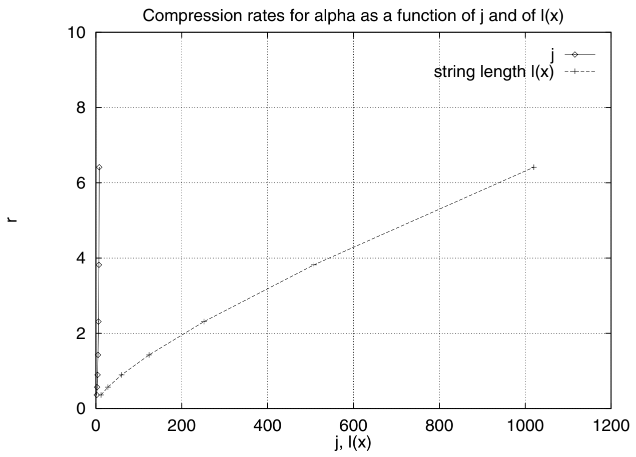


Fig. 1. The dependence of r on j and on $l(x)$ for L_{cs_6} (a) and for α (b).

and one for which the generating program is already known, so as to obtain a successful lossy compression for the former one.

Finally, we wish to implement a parallel version of our system on an MIMD machine in order to further improve the quality of the solutions.

Acknowledgements

Dr. A. Della Cioppa gratefully acknowledges the National Research Council of Italy – Institue for Research on Parallel Information Systems (Naples, Italy) for having supported his Ph.D. program.

References

[Ban93] W. Banzhaf (1993). Genetic Programming for pedestrians. *Tech. Report*: 93–03.

[Cha66] G.J. Chaitin (1966). On the length of programs for computing finite binary sequences. *Journal of the Association for Computing Machinery* **13**:547–569.

[Cha94] G.J. Chaitin (1994). The limits of mathematics. *Journal of Universal Computer Science* **2**(5): 270–305.

[Con97] M. Conte, I. De Falco, A. Della Cioppa, E. Tarantino, G. Trautteur (1997). Genetic Programming estimates of Kolmogorov Complexity. *Proc. of the Seventh International Conference on Genetic Algorithms and their Applications, 743–750.* San Francisco, CA: Morgan–Kaufmann.

[Cra85] N.L. Cramer (1985). A representation for the adaptive generation of simple sequential programs. *Proc. of the First International Conference on Genetic Algorithms and their Applications.* Pittsburgh: CMU.

[Gam99] A. Gammermann and V. Vovk (1999). Kolmogorov Complexity: sources, theory and applications. *The Computer Journal* **42**:252–255.

[Kol65] N. Kolmogorov (1965). Three approaches to the quantitative definition of information. *Prob. Inform. Transmission* **1**:4–7.

[Koz92] J.R. Koza (1992). *Genetic Programming: On Programming Computers by Means of Natural Selection and Genetics.* Cambridge, MA: The MIT Press.

[Koz94] J.R. Koza (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs.* Cambridge, MA: The MIT Press.

[Li93] M. Li, and P.M. Vitányi (1993). *An introduction to Kolmogorov Complexity and its Application.* Berlin: Springer–Verlag.

[Lof66] P. Martin–Löf (1966). On the definition of random sequences. *Information and Control* **9**:602–619.

[Nor96] P. Nordin, and W. Banzhaf (1996). Programmatic compression of images and sound. *Proc. of the First Annual Conference on Genetic Programming.* J. R. Koza, D. E. Goldberg, D. B. Fogel and R. L. Riolo eds., Stanford University, 354–350. CA, USA, Mit Press.

[Ris78] J.J. Rissanen (1978). Modelling by shortest data description. *Automatica* **14**:465–471.

[Sol64] R.J. Solomonoff (1964). A formal theory of inductive inference, I, II. *Information Control* **7**:1–22, 224–254.

[Sow97] D. Sow and A. Eleftheriadis (1997). Complexity Distortion Theory. *Proc. of ISIT 1997 International Conference, Ulm, Germany, June 29 – July 4.*

[Wal68] C.S. Wallace and D.M. Boulton (1968). An information measure for classification. *Computing Journal* **11**:185–194.

[Wal99] C.S. Wallace and D.L. Dowe (1999). Minimum message length and Kolmogorov complexity. *The Computer Journal* **42**:271–283.