Linear Genetic Programming using a compressed genotype representation

Johan Parent
Vrije Universiteit Brussel
IR, ETRO
Pleinlaan 2 1050 Brussel,
BELGIUM
johan@info.vub.ac.Be

Ann Nowé
Vrije Universiteit Brussel
Faculty of Science, COMO
Pleinlaan 2 1050 Brussel,
BELGIUM
asnowe@info.vub.ac.Be

Kris Steenhaut
Vrije Universiteit Brussel
IR, ETRO
Pleinlaan 2 1050 Brussel,
BELGIUM
kris@info.vub.ac.Be

Anne Defaweux
Vrije Universiteit Brussel
Faculty of Science, COMO
Pleinlaan 2 1050 Brussel,
BELGIUM
adefaweu@vub.ac.Be

Abstract- This paper presents a modularization strategy for linear genetic programming (GP) based on a substring compression/substitution scheme. The purpose of this substitution scheme is to protect building blocks and is in other words a form of learning linkage. The compression of the genotype provides both a protection mechanism and a form of genetic code reuse. This paper presents results for synthetic genetic algorithm (GA) reference problems like SEQ and OneMax as well as several standard GP problems. These include a real world application of GP to data compression. Results show that despite the fact that the compression substrings assumes a tight linkage between alleles, this approach improves the search process.

1 Introduction

Genetic algorithms (GA) and genetic programming (GP) search the space of possible solutions by manipulating the solution representation using genetic operators like crossover and mutation. Variable length representations allow the structure of the solution to be evolved but are still limited by the genetic primitives used to build the solutions. If GA and GP are to address more complex problems it becomes necessary to adapt the representation to the problem. Modularization adapts the representation by extending the set of genetic primitives with problem specific functionalities. In this text we explore a low level modularisation technique. The presented compressed GA (cGA) attempts to identify usefull combination of genes in the population and makes these available as new primitives. The search process can then benefit from these new functionalities to progress more quickly through the search space. The cGA achieves modularisation by applying a compression operator to the population of candidate solutions.

This paper is structured as follows Section 2 describes related work. In Section 3 the compression/substitution scheme is presented in detail. In Section 4 the different experiments are presented, each of them having different characteristics which impact on the performance of our algorithm. Results are presented in Section 5.

2 Related work

The benefits of modularization for the GP search process are well known [10][1][7]. Modularization fosters code reuse on one hand, and on the other hand allows the GP system to identify and use high level functionalities. Different modularization strategies for tree based GP have been explored. Automatically Defined Functions (ADF) is the most prominent approach. It consists of a main tree which evolves together with a predefined number of additional trees. Those additional trees can be called from the main tree and, as such, complement the set of primitives available to the GP system. An alternative method is *encapsulation*. It replaces a subtree by a new symbol and adds it to the primitive set of the system [6][10]. The symbol created in this way corresponds to a terminal/leaf node since it has no arguments. A third method, *module acquisition*, works in a similar fashion, but can create both function nodes (modules) and leaf nodes. If the depth of the selected subtree exceeds a certain threshold the subtrees below this level are removed. In this case a function node will be created by adding an argument for each removed subtree [2][3]. Although modularization has mainly been of interest to the GP community it is of course related to the search of building blocks in a GA. In [5] a *module acquisition algorithm* is presented which exploits modularity and hierarchy. Despite the use of a variable length representation, the problem of modularity is approached purely from a GA point of view. A *module* is defined as a set of gene values that correspond to a maximum fitness for the individual. As in the work of [5] our algorithm relies on a substitution scheme but with notable differences in the substitution strategy. Another difference is that the compressed genotype scheme of the cGA was developed with its use in a linear GP system in mind. The linear encoding a GP program exhibits a much tighter linkage than is the case for standard GA problems. This assumption is important as it justifies the use of the simple compression scheme which supposes a strong dependency between adjacent values.

3 A compressed GA

This section presents the compressed genetic algorithm (cGA) which consist of a GA extended with a substitution/compression based modularization mechanism. Figure 1 contains the pseudo code for cGA loop. The cGA adds a compression and a decompression step in the GA loop. Compression is applied to the individuals of the population prior to the creation of the next generation. As a result the genetic operators, 1-point crossover and 1-point mutation, are applied to a compressed genotype representations. After the creation of the next generation the individuals are decompressed so that their fitness can be evaluated.

Compression is applied to the genotype to protect

```
0.  Choose initial population
1.  Evaluate each individual's fitness
2.  Repeat
3.      Compress individuals
4.      Select best-ranking individuals to reproduce
5.      Mate pairs at random
6.      Apply crossover operator
7.      Apply mutation operator
8.      Decompress individuals
9.      Evaluate each individual's fitness
10. Until terminating condition
```

Figure 1: The standard GA pseudo code with 2 new steps. Step 3 adds the compression of the individual prior to the creation of the next generation. Step 8 decompresses the individuals so that their fitness can be evaluated.

substrings that represent building blocks (above average schemata). This protection would guarantee the preservation of good allele combinations and be beneficial to the search process. Compression can shorten the representation of the building blocks thus reduce the chance of disruption by crossover. Crossover which is modelled by the schema theory as mostly disruptive for building blocks. Since probability of disruption by crossover is proportional to the defining length of the building block, making the shorter protects them. This idea lead to step 2 of the cGA which will be described in the remainder of this section.

Compressing the genotype can protect building blocks. But it does not by itself provide modularization comparable to that of ADFs, encapsulation or module acquisition. Modularization implies that a form of code reuse should be present. Code reuse is achieved by the cGA by adhering to several conditions. First of all, the representation used by the cGA is adapted during the search. A new symbol is added to the alphabet of the genetic system for each potential building block identified by the cGA. Second, the genetic operator must be unrestricted. Which means that no distinction is made between compressed symbols and symbol of the original alphabet. Mutation for instance can replace a compressed symbol by an *original* symbol and vice versa. The third condition is that no distinction is made between individuals with and without compressed genotype. There are two possible reasons for having uncompressed individuals in the population. An individual can simply not be compressible by the cGA. Or, as will be explained in section 3.3, only a fraction of the population is subjected to compression. By not discriminating between original genes and *compressed genes*, the cGA can make full use of the genetic combinations that where identified as possible building blocks. It then becomes possible for the cGA to address problems where repetition is present. Crossover and mutation operators serve as natural mechanism to obtain this form of *translocation* of genetic information. Translocation occurs when a part of a chromosome is detached and reattached at a position different from the its original position in the chromosome. Before presenting the compression strategy of the cGA the basic compression algorithm will be described. In Subsection 2 the basics of the substitution coder used by the cGA to perform compression are presented. The compression of the individuals is a stochastic process in two stages: building the dictionary and selecting the individuals to which compression will be applied. Subsection 3.2 explains how the dictionary used for the compression is built. Subsection 3.3 describes how compression is applied to the population of the cGA.

```
0. D = build_dictionary();
1. current = in; /* in input string */
2. out = ""; /* Empty output */
3. For e in D /* Loop A */
4.    For pos = 1 to |in| /* Loop B */
5.       If match(pos, current, e.str) Then
6.          out[pos] = current[pos];
7.       Else
8.          out[pos] = e.ref;
9.          pos = pos + |e.str|;
10.      End;
11.   End
12.   current = out;
13.End
```

Figure 2: Pseudo code for the substitution step of the coder used in the cGA. Given are the dictionary $\mathcal{D}$ and the input string in. The coder searches for a match for each dictionary entry (loop A). If a match is found the reference is placed in the output, otherwise the original symbol. The inner loop (loop B) can be replaced by the more efficient string matching algorithms with a linear runtime complexity.

3.1 Substitution coder

A substitution coder is a lossless[1] compression algorithm. Compression is obtained by replacing substrings in the input by a shorter reference. A substitution coder maintains a *dictionary* which contains the substrings that need to be substituted and associates a placeholder symbol with each string. Compression involves two phases: 1) building a dictionary and 2) performing the substitutions. Figure 2 represents the pseudo code of a substitution coder. Since the content of the dictionary is critical for the compression performance, research in data compression studies algorithms to build the dictionary. Section 3.2 describes how the dictionary is built in the cGA. The substitution step is computationally expensive since it involves searching for a match for each dictionary entries in the input string. If a match for substring s is found its placeholder symbol is placed in the output instead of the original symbols. The decompression step involves the replacing the placeholder symbols by the original strings. Decompression is much faster since it does not involve any search.

Suppose a dictionary $\mathcal{D} = \{"101" \rightarrow \alpha, "00" \rightarrow \beta, "11" \rightarrow \gamma\}$ the substitution coder would compress the individual $"1010001011101"$ to $"\alpha\beta 0\alpha 1\alpha"$. One can observe

[1] Lossless compression means that the original data is restored after compression.

that the order in which the dictionary entries are ordered is important. If the opposite order had been used (11,00 and 101) the result would have been "$\alpha\beta010\gamma\alpha$".

3.2 Building the dictionary

The dictionary has to be built before the substitution(s) can be performed. This section explains the stochastic algorithm used in the cGA to build the dictionary as it differs from the algorithm(s) used for *pure* data compression applications. In this case we seek to build a dictionary which contains building blocks. Reducing memory needed to represent the individuals is not the goal of the compression. The problem of identifying building blocks is recurrent in all the modularization algorithms. In [5] the identification of a set of alleles as a module involves additional fitness evaluations. These are used to determine whether an alternative set of alleles with a better fitness exists. If no such set can be found the allele set will be used as a module. Another approach, used by [11], uses a separate *block fitness function* to evaluate the merits of a subpart of a genetic program. This function is presented as scaled down version of the fitness function which consider a smaller version of the original problem. The strategy adopted in this work is to see a GA as the building block discovery process. Since by definition building blocks are contributing in a positive way to the fitness of an individual, they are bound to be present in the better individuals of a population. We believe that this approach is more general as it 1) relies on the information already present in the population, 2) avoids additional computations and 3) does not require additional fitness functions to be engineered. Especially the last two are unacceptable when it comes to real world applications.

The cGA builds the dictionary in two stages. In the first stage, a pool of M individuals is selected stochastically from the population. The genotype of these individual that will be used to fill the dictionary in the second stage. The individuals in the pool are selected using fitness proportional selection. This pool will thus mainly consist of above average individuals, yet with a minimum of diversity. Once this pool has been created every substring of length l of each individual is added to the dictionary. In the current implementation of the cGA the dictionary does only contain strings of the same length[2]. For example, suppose a non-binary alphabet $\{a, b, c, d\}$ and the substring length l equal to 3. In this case the individual "$abccdabc$" would add the substring (and their respective placeholder) "abc" $\rightarrow \alpha$, "bcc" $\rightarrow \beta$, "ccd" $\rightarrow \gamma$, "cda" $\rightarrow \delta$ and "dab" $\rightarrow \sigma$ to the dictionary. In the current cGA implementation the dictionary contains next to the substrings and their placeholder symbol a counter of the occurrence of every substring in the selected pool. This counter is used to order entries of the dictionary. We have chosen to sort the dictionary entries based on the occurrence of each substring, the most frequently occurring substring(s) will be substituted first.

In the current version of the algorithm the dictionary is rebuilt for each generation. This allows to update the content of the dictionary with information that reflects the evolution of the population. This differs from [5] where the *module formation algorithm* is applied periodically. This also excludes the presence of dictionary entries containing references to other entries.

[2] In traditional substitution coders the dictionary can contain entries of different lengths

3.3 Compressing the population

Once the dictionary has been built the substitution coder (Subsection 2) can be used to compress the individuals in the population. The cGA applies compression stochastically to a fraction of the population. A fraction κ ($\in [0, 1]$) of the population will be compressed. This fraction κ of individuals are selected using fitness proportional selection. Since the compression setup is repeated anew for each generation this means that the cGA does not systematically compress the same individuals. This makes it possible to explore the benefit of the different modules (dictionary entries) in different contexts (ie. allele combinations).

3.4 Lossless and stochastic

The cGA applies a deterministic transformation, compression, to some of the individuals in the population. Since the compression is lossless the (genetic) information present in the population of the cGA or a GA is exactly the same. Yet, as described above, a stochastic component has been added to the overall population compression process. Both the creation of the dictionary and the choice of the individuals for compression are non-deterministic processes. This stochastic component is meant to counter balance the effect of the substitutions on the search process. The protection against crossover provided by the compression does have a negative impact on the population diversity. Several factors explain this phenomenon. First the search efficiency of the crossover operator is reduced by the compression of the genotype. Second, the compression of the individuals is detrimental for the sampling of schemata. A last reason is that the dictionary entries are not guaranteed to correspond to building blocks. This situation is especially exacerbated during the first generation as the population has yet to discover promising gene combinations. This was illustrated by experiments with κ set to 1. Those have invariably lead to premature convergence and consequently suboptimal results. The parameter κ is a way to limit the decrease in the population diversity. The other non-deterministic steps were introduced for the same reason as they maintain a minimum of diversity at the substring level.

3.5 Further changes

Although the cGA is meant to be a minimal extension to the standard GA a few extra modification were required. The biggest difference is an unavoidable transition to a variable length linear representation.

3.5.1 Variable length

Despite the use of a lossless compression algorithm and the fact that the cGA, like a standard GA, starts with a popula-

tion of individuals of identical size, individuals of different sizes quickly emerge. This phenomenon, due to the use of compression, occurs through several mechanisms:

1. not all the individual are compressed
2. different individuals compress to different sizes
3. the genetic operators can create individuals of very different sizes

Since the cGA does not discriminate between normal symbols and placeholder symbols recombination and mutation can add or remove symbols representing a substring and vice versa. This can result in an important change in length. Depending on the scenario the individual can exceed the initial individual size or be shorter. Consider the following example. Suppose the dictionary $\mathcal{D} = \{"xy" \rightarrow \alpha, "zz" \rightarrow \beta\}$ over the alphabet $\{x, y, z\}$. Mutating the second gene of the individual of length two from x to α creates "$x\alpha$". The decompressed genotype of this individual is now "xxy". Similar scenarios exists for crossover.

3.5.2 Recombination and mutation

As illustrated above the genetic operators can create individuals with very different sizes. As a result a maximum individual length has to be enforced to avoid bloat. Individuals created by crossover that exceed the maximum length are truncated. Yet in the case of both mutation and crossover one problem remains. As illustrated in the previous paragraph it is possible for an individual to exceed the maximum once it has been decompressed. In this case the cGA truncates the genotype after decompression. A minor modification to the mutation operator is needed since it has to be able to cope with a variable alphabet size.

4 Experiments

Although the cGA has been designed to be used in a linear GP setup, it has first been tested using *non GP* settings first. The cGA has been used to solve two categories of problems. The first category of problems was used to assess the compression based modularization mechanism of the cGA. More in particular its ability to identify the substring that correspond to building blocks and to reuse them. The second category is directly related to the use of the cGA in linear GP system. Here two standard GP problems and a real world data compression application served as a testbed. The different problems are introduced below.

4.1 OneMax and SEQ

The problems presented in this section are artificial in nature but provide a testbed to assess the modularization capabilities of the cGA. The OneMax problem [12] consists in maximizing the number of ones of a binary bitstring. The fitness function is then simply $F(\vec{x}) = \sum_{i=1}^{N} x_i$. This problem contains much repetition and is thus used to test the ability of the cGA to reuse the acquired genetic combinations. This problem has been extended by using a non-binary alphabet. A larger alphabet puts more strain on the dictionary building process.

The second problem, the sequence problem (SEQ) [5] does not feature repetition. It is designed to evaluate the module acquisition abilities of an algorithm. The SEQ problem is defined as the search for a string of size n over an alphabet of equal size n. The problem defines two global optima, the strings containing all the symbols either in ascending or in descending order. The fitness function of this problem combines two aspects. Individual genes contribute a value of 1 to the fitness if they have a correct value (ascending or descending). In addition, any pair of neighbouring genes positions $(2j, 2j+1)$ (with $j \in [0, \frac{n}{2}]$) with correct values adds another 2 points to the fitness. This fitness function thus favours correct genes values as well as correct combinations of genes.

4.2 Symbolic regression, Even-5-parity

The assumption of tight linkage between individual genes (cfr substrings) used by the cGA was made with the use of the cGA in a linear GP system in mind. A linear GP system uses a linear representation similar to that used by a GA [4][13]. Following the approach used by [13] the GP loop is consists of a standard (generational) GA on top of which a problem specific evaluator is placed to simulate the program execution. This cGA driven linear GP has been tested on three problems. First the standard symbolic regression and Even-5-parity problem described in this subsection were used. The next subsection details the use of cGA in a compression application. The nature of these three problems is quite different. They were chosen to illustrate the applicability of the cGA's modularization scheme. The problems are briefly described below together with a more in depth presentation evaluators used for each problem.

The symbolic regression task is that of discovering a function (in symbolic form) given a set of data points. This experiment has been done with two target functions. The first $f_1(x) = x^4 + x^3 + x^2 + x$ features much repetition, the second $f_2(x) = cos(x) + sin(x)$ not. Each time the data set consisted of 20 values for x, randomly chosen in the intervals $[-1, 1]$ and $[-2, 2]$ for f_1 and f_2 respectively, and the corresponding $y = f(x)$ values. The terminals was $T = \{x\}$. To offer a fair comparison different function sets were used $F_1 = \{+, -, \times, \%, sin, cos, rlog, exp\}$ for f_1 and $F_2 = \{+, -, \times, \%\}$ for f_2. The raw fitness (f_{raw}) is the sum of the absolute value of the error between the computed y value and the target value. The standard fitness: $f_{std} = \frac{1}{1+f_{raw}}$

4.2.1 Stack based evaluator

The individual for the symbolic regression problem are encoded using a postfix representation. This allow their evaluation using a stack based virtual machine. The virtual machine provides the standard[3] arithmetic operations. The operations take their operands from the stack and push their re-

[3] As is usual in GP the operations are *protected*. Division by zero returns zero, rlog(x) returns zero for any x smaller or equal to zero.

sult onto it. The execution of an operation is skipped whenever there are not enough values on the stack. As a terminal we use x which is implemented by pushing the value of the independent variable on the stack. Ephemeral random constants were not used. The result of the program is the value of the top of the stack after evaluation. The individual [sin, x, +, cos, x, +, exp] represents the equation $exp(x+cos(x))$.

The Even-n-Parity problem requires the correct classification of bit strings of length n having an even number of 1's. This classification is formulated as a boolean function returning the value *true* for an even number of 1's and *false* otherwise. The terminals and function set are $T = \{b_0, b_1, b_2, b_3, b_4\}$ and $F = \{AND, OR, NAND, NOR\}$. The raw fitness ($f_{raw}$) is the ratio of correct classifications over the entire data set. The standard fitness is equal to the raw fitness for this problem.

4.2.2 Boolean evaluator

The Even-5-parity individuals are postfix encoded as well and evaluated by a stack based virtual machine as described in [9]. Its instruction set provides the four boolean operators[4] and five terminals. The operations take their 2 operands from the stack and push their result onto it. The terminal instructions correspond to a push of the individual input bits on the stack. As for the numerical evaluator the result of the program is the value of the top of the stack after evaluation. The individual [b_0, NOR, NAND, b_0, b_4, OR, AND, b_2, OR, b_2, NOR, NAND] corresponds to the boolean expression (b_0 OR b_4) NAND (b_2 NOR b_2).

4.3 Compression preprocessors

The last problem used to evaluate the cGA is a lossless data compression application. In this setting genetic programming is used to find a transformation, denoted $T(d)$, which improves the compression ratio of data d [8]. The transformation is applied to the data prior to compression and needs to be undone after decompression. The purpose of this GP based application is to further reduce the size of medical images obtained from MRI, CT and PET scanners. The GP system generates image specific transformations. The critical nature of the information requires lossless compression. To meet the lossless requirements the GP generated transformation needs to be reversible. This condition can be meet with a correct design of the GP instruction set. The instruction set for this problem consists of basic transformations. Each of this transformation can remove certain forms of redundancy, thus lower the entropy of the data. The terminals set T contains the constants from 1 to n, $T = \{1, 2, 3, \cdots, n\}$. These will be used for the index and parameter of the different transformations (see 4.3.1). The function set contains basic transformations $F = \{dpcm, raw, min, inv\}$. The size of the data d after being transformed with $T(d)$ and compressed with an algorithm C should be smaller than without the transformation. This condition is capture in Equations 1. The reversibility of the transformation is formalized in Equation 2.

$$|C(T(d))| < |C(d)| \tag{1}$$

$$\forall T, \exists T' | T'(T(d)) = d \tag{2}$$

In order to avoid computation expense of Equation 1 the problem statement is reformulated as searching a function which reduces the information content of the data. The information content corresponds to the theoretical amount of information, expressed in bits, present in the message m. It can be computed by multiplying the entropy of a message ($H(m)$) by the length of the message($|m|$). The entropy of a message corresponds to the average bits of information per symbol in that message. Equation 3 gives the equation for the entropy of a message m. The marginal probability of a symbol in the message m is denoted by p_i. The raw fitness is the ratio of the information content of the original data and the data after transformation by T [5] : $f_{raw} = \frac{H(d)\times|d|}{H(T(d))\times|T(d)|}$. The standard fitness equals the raw fitness for this problem.

$$H(m) = -\sum_{\text{i=1}}^{\text{n}} p_i . \log_2(p_i) \tag{3}$$

4.3.1 Transformation Virtual Machine

This virtual machine reads the data from disk and splits it into chunks of equal size prior to the execution of a program. The number of chunks is currently fixed a priori as our implementation does (for simplicity's sake) not allow to re-segment the data during evaluation. The instructions transform one chunk at a time. The general form of an instruction is a 3-tuple: $< instr, index, param >$. The first *byte* is the instruction number, the second the index of the chunk to be processed and the last a parameter for the transformation[6]. This instruction set is both simple and flexible. Chunks can be processed several times in any order or skipped altogether. To assure the reversibility of the transformation extra information is required. In addition to the transformed data every instruction produces a header with *decoding information*. This increases the size of the chunk each time its data is processed. At the end of the evaluation the VM concatenates the content of all the chunks to produce the *output data*. The basic transformations are described shortly below.

- dpcm : stands for differential pulse code modulation. $\{x_0, x_1, x_2, x_3, \ldots, x_n\} \Rightarrow \{x_0, x_1 - x_0, x_2 - x_1, x_3 - x_2, \ldots, x_n - x_{n-1}\}$ This operation has a stride/step parameter.
- raw : no transformation is applied at all to the data. $\{x_0, x_1, x_2, x_3, \ldots, x_n\} \Rightarrow \{x_0, x_1, x_2, x_3, \ldots, x_n\}$

[4] Following the observations of [9] the lazy version of the operators was implemented.

[5] This fitness function does not preclude an increase of the data size after transformation. Some transformation in fact increase the size of the data, see Section 4.3.1.

[6] Not all the transformation require a parameter, yet we have chosen to use a fixed format for the instructions.

Alphabet	GA	cGA
2	0.581	0.692
8	0.181	0.273

Table 1: The raw fitness of the best of population (averaged over 100 runs) for the OneMax problem. Target string length is 1024.

- min : This operation uses the average of all the symbols (binary representation) in the current chunk. A symbol is replaced by the difference between the average and the symbol. This operation expands the data by 1 symbol, the symbol which represents the average[7].

  $\{x_0, x_1, x_2, x_3, \ldots, x_n\} \Rightarrow \{x_{avg}, x_0 - x_{avg}, x_1 - x_{avg}, x_2 - x_{avg}, x_3 - x_{avg}, \ldots, x_n - x_{avg}\}$ This operation has a stride/step parameter.

- inv : inversion. $\{x_0, x_1, x_2, x_3, \ldots, x_n\} \Rightarrow \{MAX(x) - x_0, MAX(x) - x_1, MAX(x) - x_2, MAX(x) - x_3, \ldots, MAX(x) - x_n\}$ Here $MAX(x)$ represents the maximum value that can be represent by the data type used to represent the symbols. This operation has a stride/step parameter.

5 Results

All the results presented here are the average of 100 independent runs, using randomly seeded initial populations. The genotype length is 32. Other settings were: crossover rate 80%, mutation rate 5% and the top 5% of the population was kept for every generation. Every experiment used 50 generations.

For the runs with the cGA the following values were used: 33% of the population was selected for compression (κ) and substrings of length 2 were used as dictionary entries. A pool of 10 individuals is used to build the dictionary.

5.1 OneMax

Two instances of the OneMax problem were used. A first instance using a binary alphabet and second with an alphabet size equal to eight. For both instances the string size was 1024. Table 1 compares the raw fitness of the best individual from a population of size 500. As one might expect, the repetition present in this problem is favorable to the cGA.

5.2 SEQ

Two instances of the SEQ problem were used, each time with a population of 500 individuals. The string lengths were 32 and 128 (the alphabet size equals the string length). The experiments show that the GA systems performs very poorly for this task, score of the best individual(s) remaining much lower than the theoretical maximum score. This may be due to inappropriate choice of parameters. The results indicate that the modularization used by the cGA degrades the performance compared to the GA. The raw fitness of the best individuals drops from 18.66 (GA) to 17.78 (cGA) for the size 128 problem instance and from 21.69 to 18.40 for the size 32 problem instance.

[7] min (stride=1) changes the series 2_5_3_2_5_6_7 to -2_1_-1_1_2_3 + 4 (the average)

Pop. size	GA (f_1)	cGA (f_1)	GA (f_2)	cGA (f_2)
50	0.138	0.182	0.094	0.095
100	0.175	0.290	0.100	0.110
500	0.341	0.527	0.128	0.161

Table 2: The raw fitness of the best of population (averaged over 100 runs) for function f_1. One can see that for this problem the cGA outperforms the GA for the different population sizes

5.3 Symbolic regression

This first GP problem has been run with different population sizes. Table 2 contains the value of the best individual of the population of both the GA and the cGA for both functions. No parameter tuning has been done, consequently we are aware of the fact that these results may not be competitive with state of the art EA based implementations.

As could be expected the population size is an important factor, as one can see the fitness consistently improve as the population grows. But more importantly one can see that the addition of compression in the genetic loop has a positive influence on the search process. The difference in performance is significant for all population sizes (see section 6).

One can argue that those differences are due to the variable length representation of the cGA. Although this may be a factor, several facts do not sustain this assertion. First, an analysis of the average genotype length revealed a decrease of the genotype length from 32 down to 27 chromosomes over 50 generations, a 15% decrease. It seems unlikely that such a minor difference in size alone could explain the difference in performance. To confirm this hypothesis 100 additional runs using the GA with an initial population seeded with individuals of sizes ranging 27 to 32 were done. The results, with an average best raw fitness of 0.362 shows little difference with the constant length GA. Which suggests that having a range of program sizes alone is insufficient.

5.4 Even 5-parity

The even 5-parity problem was run with 2 different population sizes. Again no parameter tuning has been performed. For this problem too, the use of substitution had a positive effect on the performance of the GP system. Figure 3 compares the fitness of the best of population for both algorithms. As for the symbolic regression problem the difference in fitness is significant.

In order to compare the performance of the two algorithms the cumulative probability of success has been computed using runs of 200 generations (population 500). 200 generations being more than sufficient for both algorithms to convergence. The linear GP driven by the GA achieves a

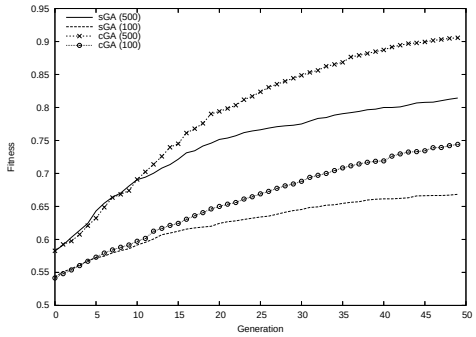

Figure 3: The even 5-parity problem for population size of 100 and 500. For this problem too the cGA outperforms the GA.

Table 3: Medical image set

Filename	Size (bytes)	Resolution	Slices
ADAM.bin	17039360	256x256	130
Bone.bin	2805264	1386x2024	1
Pacemaker.bin	6555120	1716x1910	1
MRINormal.bin	15990784	256x256	244

28% probability of success after 200 generations. The same setup but using the cGA obtains a probability of success of 74%.

5.5 Data compression

The medical images listed in Table 3 were used to test our approach. Being genuine medical images none of the files are less than 2 megabytes in size. This implies that our linear GP system will process huge amounts of data during a run. For this reason a population size of 50 was used.

Table 4 contains the relative reduction in entropy achieved by transforming the data. With the given data set, the linear GP has been able to reduce the randomness, hence improves the compression ratio, of the given files. Using the cGA to drive the linear GP one can systematically achieve higher gains compared to the GA. For the Pacemaker image there is a relative increase of 5% ($\frac{19.7\%-18.7\%}{18.7\%}$).

The size after compression with the GZIP compression algorithm can be found in table 5. It contains the compressed size both without and with image specific transformation (indicated by GP+GZIP). The current design of the virtual machine, with its generic transformation, do not consider the two and three dimensional nature of the medical images. This inherently limits the absolute compression gain that can be achieved and therefore deserves further work (see Section 8).

Filename	GA	cGA	gain
ADAM.bin	1.7%	2.5%	47%
Bone.bin	45.3%	45.8%	1%
Pacemaker.bin	18.7%	19.7%	5%
MRINormal.bin	5.2%	5.8%	11%

Table 4: Reduction in entropy after transformation for each image (average of 100 runs). One can for example remove roughly 20% of the randomness of the Pacemaker.bin file (before compression) and this without loss of information.

Filename	Size	GZIP	GP+GZIP
ADAM.bin	17039360	6604349	6563025
Bone.bin	2805264	1904731	1407500
Pacemaker.bin	6555120	5900586	5013531
MRINormal.bin	15990784	13836484	13792211

Table 5: Comparison between the compressed size of the original and the transformed data (GP+GZIP). The compression algorithm used is gzip.

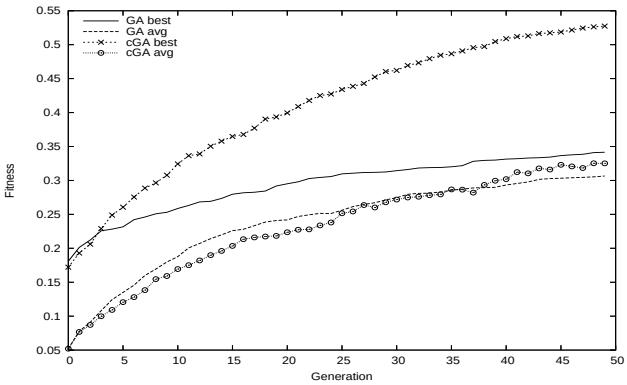

Figure 4: Comparison of the average fitness and best fitness for a population size of 500. Despite the difference in fitness of the best individual, the average fitness for the GA and cGa follow a very similar evolution.

6 Discussion

The cGA introduced in this paper provides a form of modularization at the lowest level of representation. Through the identification and substitution of substrings the cGA creates a series of very short pieces of genetic code which can be compared to small subroutines/macros in assembler. The unpaired Wilcoxon-Mann-Whitney test has been used to confirm that the difference in performance is indeed significant for all the presented problems.

The difference in performance between the runs with the GA and cGA appears to be the result of the protective effect of the substitutions. One could hypothesise that the substitution has an impact on the genetic operators. Indeed one can suppose that since the operators are unrestricted, the perturbations due to the operators are potentially much higher. For example mutating a symbol which replaces a substring is a much more significant change than in the *traditional* case. A similar scenario exist when crossover moves much more symbols by exchanging *compressed symbols*. In other words, since both algorithms use elitism, the cGA should then correspond to a GA with a higher mutation rate. Figure 4 depicts the average and best fitness for both algorithms. One can observe that the progress of the average fitness of the GA and cGA are very similar. If the previous hypothesis would be correct one would except a lower average fitness

for the cGA than for the GA.

7 Conclusions

In this paper we introduce a GA variant which uses compression as a modularisation mechanism. Assuming a tight linkage between elements of building blocks the cGA replaces substrings in the genotype with a shorter reference. This scheme was designed to be used in a linear GP system where the assumption is more likely to hold. We demonstrate the cGA on a wide range of test problems, going from synthetic problems to a real world data compression GP application, to illustrate the feasibility of our approach. Results show that although the cGA seemed to perform poorly on one of the synthetic problems it performs reliably on the GP problems.

8 Future work

The influence of the parameters of the cGA needs to be studied in greater detail. Currently the length of the substrings has empirically been set to 2. Strategies to let the size of the substrings evolve during a run should be considered. This would allow to build modules for higher order schemata. One possibility would be to merge adjacent substrings. Similarly parameters like the pool size used to build the dictionary and the percentage of the population being compressed need to be studied.

Bibliography

[1] M. Ahluwalia and L. Bull. Coevolving functions in genetic programming. *Journal of Systems Architecture*, 47(7):573–585, July 2001.

[2] P. J. Angeline and J. B. Pollack. The evolutionary induction of subroutines. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, Bloomington, Indiana, USA, 1992. Lawrence Erlbaum.

[3] P. J. Angeline and J. B. Pollack. Coevolving high-level representations. In C. G. Langton, editor, *Artificial Life III*, volume XVII of *SFI Studies in the Sciences of Complexity*, pages 55–71, Santa Fe, New Mexico, 15-19 June 1992 1994. Addison-Wesley.

[4] M. Brameier and W. Banzhaf. Effective linear genetic programming. Technical report, Department of Computer Science, University of Dortmund, 44221 Dortmund, Germany, 2001.

[5] E. D. de Jong and D. Thierens. Exploiting modularity, hierarchy, and repetition in variable-length problems. In K. Deb, R. Poli, W. Banzhaf, H.-G. Beyer, E. Burke, P. Darwen, D. Dasgupta, D. Floreano, J. Foster, M. Harman, O. Holland, P. L. Lanzi, L. Spector, A. Tettamanzi, D. Thierens, and A. Tyrrell, editors, *Genetic and Evolutionary Computation – GECCO-2004, Part I*, volume 3102 of *Lecture Notes in Computer Science*, pages 1030–1041, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[6] D. Howard. Modularization by multi-run frequency driven subtree encapsulation. In R. L. Riolo and B. Worzel, editors, *Genetic Programming Theory and Practise*, chapter 10, pages 155–172. Kluwer, 2003.

[7] J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors. *Genetic Programming 1996: Proceedings of the First Annual Conference*, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[8] J. Parent and A. Nowe. Evolving compression preprocessors with genetic programming. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 861–867, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[9] T. Perkis. Stack-based genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 148–153, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[10] S. C. Roberts, D. Howard, and J. R. Koza. Evolving modules in genetic programming by subtree encapsulation. In J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 160–175, Lake Como, Italy, 18-20 Apr. 2001. Springer-Verlag.

[11] J. P. Rosca and D. H. Ballard. Hierarchical self-organization in genetic programming. In *Proceedings of the Eleventh International Conference on Machine Learning*. Morgan Kaufmann, 1994.

[12] J. Schaffer and L. Eshelman. On Crossover as an Evolutionary Viable Strategy. In R. Belew and L. Booker, editors, *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 61–68. Morgan Kaufmann, 1991.

[13] K. Stoffel and L. Spector. High-performance, parallel, stack-based genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 224–229, Stanford University, CA, USA, 28–31 July 1996. MIT Press.