Heterogeneous Cooperative Coevolution: Strategies of Integration between GP and GA

Leonardo Vanneschi, Giancarlo Mauri,
Andrea Valsecchi
Dipartimento di Informatica,
Sistemistica e Comunicazione
University of Milano-Bicocca
Milan, Italy
{vanneschi,mauri}@disco.unimib.it,
andrea.valsecchi@studenti.unimib.it

Stefano Cagnoni
Dipartimento di Ingegneria dell'Informazione
University of Parma
Parma, Italy
cagnoni@ce.unipr.it

ABSTRACT

Cooperative coevolution has proven to be a promising technique for solving complex combinatorial optimization problems. In this paper, we present four different strategies which involve cooperative coevolution of a genetic program and of a population of constants evolved by a genetic algorithm. The genetic program evolves expressions that solve a problem, while the genetic algorithm provides "good" values for the numeric terminal symbols used by those expressions. Experiments have been performed on three symbolic regression problems and on a "real-world" biomedical application. Results are encouraging and confirm that our coevolutionary algorithms can be used effectively in different domains.

Categories and Subject Descriptors

I.2 [**Artificial Intelligence**]: Automatic Programming

General Terms

Algorithms

Keywords

Coevolution, Genetic Algorithms, Genetic Programming

1. INTRODUCTION

This paper focuses on the study of coevolutionary algorithms which optimize the values of the numeric terminals of a genetic program being evolved. The simple algorithms presented here can be considered as a very first step towards the much longer-term goal of designing complex systems based on the (self-organized) coevolution of different heterogeneous populations, which exchange information that may not only affect the dynamics of their evolution, but even their primary goals.

Previous work in which constants of a genetic program have been effectively optimized has dealt with memetic algorithms, in which classical local optimization techniques (e.g., gradient descent [23], linear scaling [10] or other methods based on diversity measures [22]) were used find good values for the constant set of some simple symbolic regression problems. Symbolic regression problems consist of finding an algebraic expression which correctly maps a set of input data into a set of corresponding outputs. These problems may be solved by means of Genetic Programming (GP)[12], where expressions are usually represented as tree structures, which are built using a set of functions $\mathcal{F}$ and a set of variables and constants $\mathcal{T} = \mathcal{V} \cup \mathcal{K}$. The elements of the sets $\mathcal{F}$ and $\mathcal{T}$ have to be chosen before executing GP and this choice may considerably affect the success of GP in finding satisfactory solutions. While the choice of the number of variables is often naturally induced by the problem specifications, the choice of functions and, even more, of constants is usually much harder and less obvious. In many contributions (see for instance [4, 12, 25]) symbolic regression problems have been solved using a set of simple arithmetic operators and a set of randomly generated constants, usually referred to as ephemeral random constants (ERCs). One of the main weaknesses of those systems resides in the fact that also the search for the appropriate values of the numeric terminals may involve the navigation of very complex fitness landscapes. In those cases, local optimization techniques may get stuck in local optima, failing to find "good" constants. This may be avoided by using a more sophisticated technique to search for numeric terminals than a local optimization algorithm.

The present work describes a set of algorithms in which the values of the numeric terminals used by GP are determined by a coevolving GA. While hybrid systems integrating Genetic Algorithms (GAs) and Artificial Neural Networks (ANNs) or in which ANNs were co-evolved as ensembles of neurons have been described [2, 27, 15, 14, 8], coevolution of GP and GAs has never been investigated before, except for a very first attempt in [3], of which this paper can be considered the natural extension. The idea underlying coevolution is far from new [11, 18, 17], nevertheless it has only recently become a very hot research topic in Evolutionary Computation (EC) for both theory [26] and applications [19, 20]. Differently from the most common cooperative coevolutionary systems, in which homogeneous populations (i.e., with similar genotypic representations) coevolve, we use two heterogeneous populations, one composed by tree expressions and the other one composed by fixed length strings.

The paper is structured as follows: section 2 provides the details of the symbolic regression problem. The subsequent sections present the different coevolutionary strategies we have studied and compare their performances with standard GP and with GP using gradient descent (GP+GD) to optimize constants locally; in particular, section 3 presents the simplest of the algorithms we have considered, called Basic-GPGA, section 4 presents a first variant of Basic-GPGA called *DELAY*, while sections 5 and 6 present two other variants called *CINI* and *AUTO*, respectively. In section 7, experimental results obtained by our coevolutionary algorithms in solving a "real-world" biomedical problem are presented. Finally, section 8 offers our conclusions and ideas for future activities.

2. EXPERIMENTAL SETTING

Each coevolutionary algorithm presented in this paper is based on the alternate evolution of two populations. The first one is composed by GP expressions built from the sets $\mathcal{F} = \{+, -, *, //\}$ (where // indicates the protected division i.e. $x//y$ returns x/y if $y \neq 0$ and 1 otherwise) and $\mathcal{T} = \mathcal{V} \cup \mathcal{K} = \{x_0, x_1, ..., x_{m-1}\} \cup \{c_0, c_1, ..., c_{n-1}\}$, where $\mathcal{V}$ is the set of variables involved in the objective function and $\mathcal{K}$ is a set of references to 'external' numeric terminals. During fitness calculation, references point to the numeric values represented by the best individual of a coevolving population of fixed-length strings, driven by a GA, which will be called *GA population*. Fitness of GP individuals will be calculated as the mean square error (MSE), defined as follows:

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (d_i - f(\mathbf{x}_i, \mathbf{c}))^2$$

where N is the number of fitness cases, i.e. of input-output pairs $(d_i, \mathbf{x}_i)$, $\mathbf{x}$ is a vector of input values, and $\mathbf{c}$ is a vector of numeric constants.

In all experiments, evolution parameters for the GP have been set as follows: population of size 200, grow initialization, standard subtree-swapping crossover [12] with rate equal to 0.9, standard subtree mutation [12] with rate equal to 0.1, tournament selection with tournament size equal to 10, maximum depth for the individuals for both initialization and crossover equal to 8, elitism (i.e., the best individual is copied unchanged into the next population). The following parameters were used for the GA: population size equal to 100, 13 bits per constant, standard single-point crossover [7, 9] with rate equal to 0.9, standard point mutation [7, 9] with rate equal to 0.01, tournament selection with tournament size equal to 8, elitism (i.e., the best individual is copied unchanged into the next population). Furthermore, in the experiments presented here, the number of numeric terminals in $\mathcal{K}$ has been set to 6 (and thus the string length of the GA individuals was $6 \times 13 = 78$ bits). The fitness of each individual i in the GA population is the best fitness which can be obtained by GP using the set of constants represented by i. Experiments using a different size for $\mathcal{K}$ and evaluating fitness of GA individuals averaging fitness over the a number of GP individuals larger than one have been performed and presented in [24]. Based on such results, changing those parameters does not seem to significantly affect performances.

The variants of the coevolutionary algorithm were tested on the following functions:

$$F(x, y, z) = (x + y + z)^2 + 1$$
$$G(x, y, z) = \tfrac{1}{2}x + \tfrac{1}{3}y + \tfrac{2}{3}z$$
$$K(x, y, z, w) = \tfrac{1}{2}x + \tfrac{1}{4}y + \tfrac{1}{6}z + \tfrac{1}{8}w$$

Fitness has been evaluated using 30 fitness cases randomly generated in the range $[-100, 100]$ as in [23].

2.1 Computational Effort

In evaluating GP results, basing performance comparisons among different GPs on the best fitness value obtained in corresponding generations or, as it is usually done with GAs, after performing the same number of fitness evaluations is inadequate. While this is often acceptable in EAs with fixed length representations, it can be misleading in GP, where individuals change their size dynamically. Therefore, we analyzed results, as in [6], against the *computational effort* (CE), defined as the total number of nodes that have been evaluated up to that moment. This quantity is strictly increasing with the number of generations. Strictly speaking, one should take into account the different computation times required from different operators in $\mathcal{F}$, but this simplification is still a better first approximation than the ones mentioned above. Clearly, this measure is also problem-specific, but it is useful to compare different solutions to the same problem.

In order to compare our coevolutionary algorithms with standard GP, we have evaluated the CE spent by both GP and GA. Let N_{GP} be the population size in the GP. To evaluate the fitness of a GA individual, it is necessary to calculate the fitness of all GP individuals which use numeric terminals. Thus, the following formula expresses the computational effort spent by the GA population at each generation:

$$CE_{GA} \leq N_{GA} * \sum_{j=1}^{N_{GP}} l_j \qquad (1)$$

where l_j is the number of nodes of the j^{th} GP individual and N_{GA} is the size of the GA population. This CE must be added to the CE of the GP, to estimate the global CE spent by the whole coevolutionary system.

2.2 Performance comparison

Comparing performances of EAs is a notoriously difficult issue. For problems in which the solution is not known, such as hard real-life optimization problems, a useful figure of merit is the *mean best fitness* (MBF). This measure is defined as "the average over all runs of the best fitness values at termination" [5]. The very concept of termination itself is rather fuzzy. Indeed, in the above situation, one does not know in advance whether the global optimum has been reached. Consequently, one common attitude is to measure performance after a specified amount of CE has been spent. For problems with known solutions, such as those that are studied here, MBF-based measures are not entirely adequate because a sizable part of the runs are unsuccessful within the prescribed effort limits; using larger effort values would help in some cases, but could become prohibitively expensive. This prevents one from knowing whether increasing the length of the runs would have been useful and in which cases. Thus, MBF only partially describes the problem-solving capabilities of the methods under evaluation. Instead, when the solution is known, the *success rate* (SR), defined as the ratio of successful runs with respect to the total number of runs, which have been terminated after reaching a specified CE, is a good indicator of algorithmic effectiveness [5, 1]. Some criticisms have been recently raised against SR as a reliable measure [13], especially in the case of unknown search spaces. These criticisms are well founded, but, in our opinion, they do not fully apply to the present case, because the test problems have known solutions.

In our previous paper [3], we have compared a basic coevolutionary algorithm, similar to Basic GPGA presented here, comparing it with plain GP by evaluating the MBF against number of generations (therefore neglecting, with respect to using CE, any computation overhead introduced by the GA stage) and number of fitness evaluations (neglecting only tree size). Results have shown that, as expected, adding a constant optimization stage dramatically speeds up convergence in terms of number of generations. However, the number of fitness evaluations after which the coevolutionary strategy produces better results on an absolute scale strongly depends on the choice of the parameters that regulate it, which seem to be problem-dependent.

In this study, a run will be considered successful if an individual with an error value lower than 30 has been evolved within the allotted CE; SR *and* MBF are both reported and evaluated against CE. In fact, both measures seem to be useful, since they present results from different points of view: SR allows one to understand the ability of each algorithm to find optimal solutions, while the mean fitness curves provide a visual feeling of the performance of the algorithms over time. Overall, the concurrent evaluation of these two measures should give a faithful and rather complete picture of the evolutionary process. The number of runs performed per experiment described in this paper is unusually high (100); published experimental work on EAs mostly evaluate performances by performing 20 to 50 runs per experiment. Furthermore, a statistical study of data (inspired by the one presented in [6]) is also presented. From a statistical point of view, GP runs may be considered as a series of independent trials of the same experiment, having only two possible outcomes: success (if a solution accurate enough is found before a prefixed amount of CE has been spent) or failure (otherwise). In this case, the number of successes (or of failures) is binomially distributed [21]. The maximum likelihood estimator $\hat{p}$ for the mean of a series of independent equi-probable Bernouilli trials, and hence for the probability of success, is simply the number of successes divided by the sample size (the number of runs n). With this information at hand, one can calculate the *sample standard deviation* $\sigma = \sqrt{n\ \hat{p}\ (1-\hat{p})}$. This measure will also be included in the tables reporting the number of successful runs.

In this paper, we compare our coevolutionary algorithms with conventional (or standard) GP and with GP optimized by gradient descent (GP+GD) as described in [23]. Our coevolutionary algorithms will be presented in sections 3, 4, 5 and 6, while GP+GD is briefly described in subsection 2.3.

2.3 GP with Gradient Descent

The GP+GD algorithm attempts to minimize the MSE of the best individual in the GP population running a few iterations of a simple gradient descent. At each generation, the vector of numeric terminals $\mathbf{c}$ is updated n times using the rule:

$$\mathbf{c} \leftarrow \mathbf{c} - \alpha \nabla MSE(\mathbf{c})$$

where α is the learing rate. To calculate derivatives efficiently, we have applied the same technique as in [23]. As in that paper, we have assumed that GP expressions do not contain any non-differentiable node, which is true for our function set $\mathcal{F}$. Furthermore, as in [23], we have set $\alpha = 0.5$ and $n = 3$ (see [23] for a motivation of these choices). Each component of $\nabla MSE(\mathbf{c})$ is a tree which has to be evaluated for each iteration. Therefore, the additional computational effort required by one iteration of gradient descent is equal to the number of nodes of all the trees in $\nabla MSE(\mathbf{c})$.

3. THE BASIC GPGA ALGORITHM

Let a *turn* be the isolated and uninterrupted evolution of one population for a fixed number of generations. The Basic GPGA (B_{GPGA} from now on) algorithm consists of the iterated execution of a turn of NG_{GP} generations of the GP, followed by a turn of NG_{GA} generations of the GA population. In the experiment reported in this paper we set $NG_{GP} = 20$ and $NG_{GA} = 50$[1]. Once both turns are completed, the numeric terminals used by all the individuals in the GP population are assigned the values represented by the best GA individual.

In the basic algorithm described in [3] the GA was used to optimize the values of the constants used by the best individuals in the GP population from scratch. In the algorithms proposed in this paper, after the first turn, when a new turn of the GA starts, the GA population is not re-initialized but is still the same as in the last generation of the previous turn. Random initializations of the populations take place only once at the beginning of the run. Even if the algorithms presented here still have a strong sequential nature, the term coevolution suits this approach much better than the previous one, besides making search much less local.

3.1 Experimental Results

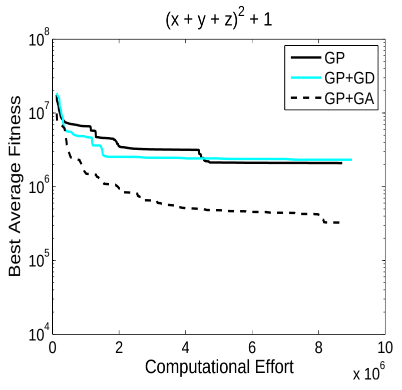
In Figure 1 the mean best population fitness is plotted against computational effort for B_{GPGA}, standard GP and GP+GD over the three test functions.

As the figure clearly shows, B_{GPGA} outperforms standard GP and GP+GD on function F, while it is outperformed by both standard GP and GP+GD on function G. Finally, on function K, B_{GPGA}, standard GP and GP+GD have found equivalent solutions, in terms of quality, for all values of computational effort which have been considered.

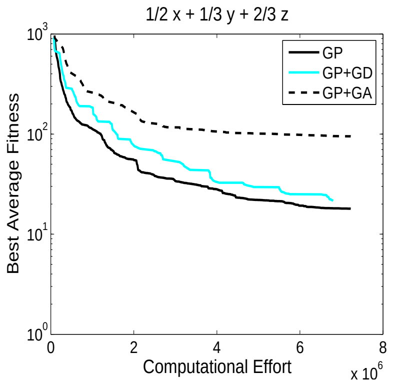
Table 1 shows results in terms of SR, along with the sample standard deviations, for the same 100 independent runs of Figure 1 for B_{GPGA}, standard GP and GP+GD.

B_{GPGA} and standard GP have exactly the same SR on function F, while the SR of GP+GD is lower. Standard GP has a higher SR number than B_{GPGA} and approximatively the same SR as GP+GD on function G. B_{GPGA} has the

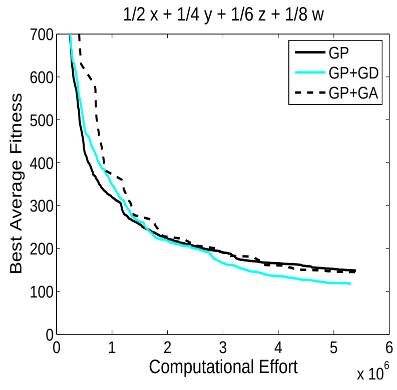
[1]Experiments with different number of generations have also been performed. They have not been shown here for lack of space. Results can be found in [24].



(a)



(b)



(c)

Figure 1: Best average fitness against computational effort over 100 independent runs for standard GP, GP+GD, and B_{GPGA}. In all algorithms the size of $\mathcal{K}$ has been set to 6. (a): results on function F; (b): results on function G; (c): results on function K.

Table 1: Number of successes, with corresponding standard deviations, calculated over 100 independent runs for standard GP, GP+GD and B_{GPGA}. Each row refers to a different test function, whose name is reported in the first column.

	std GP	GP + GD	B_{GPGA}
F	55 ($\sigma = 4.975$)	41 ($\sigma = 4.918$)	55 ($\sigma = 4.975$)
G	88 ($\sigma = 3.249$)	90 ($\sigma = 3.0$)	81 ($\sigma = 3.923$)
K	72 ($\sigma = 4.489$)	66 ($\sigma = 4.737$)	71 ($\sigma = 4.537$)

same SR as standard GP on function K, while GP+GD has a lower SR.

In conclusion, the B_{GPGA} algorithm has been effective in searching function F, in terms of MBF. It has found the same number of optimal solutions as standard GP, but it has been able to produce better solutions than standard GP in the runs which have not been successful. On the other hand, B_{GPGA} has shown worse (or sometimes equivalent) performance than standard GP for the other test functions studied. These preliminary results led us to developing some variants of the B_{GPGA} algorithm, to improve its performances. Such variants and the experimental results which have been obtained are described in the following sections.

4. THE DELAY ALGORITHM

This algorithm works like B_{GPGA} with the only difference that, before starting the alternated execution of GP and GA turns, the GP alone is executed for a certain number of generations. The idea behind this algorithm is that GP must develop “good” tree structures before “good” numeric constants can be generated: the computational effort spent by GA in attempting to generate constants for individuals which do not have the right structure yet may be wasted. The delay D, in terms of generations, with which the evolution of the GA population begins saves the computational effort that would be spent to generate the numeric costants for tree structures which are still far from an optimal structure.

At the beginning of a run, GP alone is run for D generations. Some experiments have been performed with different values of D. Results of these experiments (not shown here for lack of space, but presented in [24]) have confirmed that 100 may be a reasonable value for D (it corresponds to 5 turns of GP).

4.1 Experimental Results

Figure 2 plots the values of the mean best population fitness against computational effort for $DELAY$, standard GP and GP+GD in searching the three test functions.

The $DELAY$ algorithm clearly outperforms standard GP and GP+GD on function F (it also outperforms the B_{GPGA} algorithm, see Figure 1(a)). Furthermore, it is interesting to remark that the $DELAY$ curve in Figure 2(a) has a sudden improvement, corresponding to a value of the computational effort approximatively equal to 3. This probably corresponds to the end of the first turn of the GA. Our interpretation of such a result is that automatically generating numeric terminals for ’well-evolved’ tree structures has been very useful. On the other hand, $DELAY$ is outperformed by standard GP and GP+GD on function G, although much less than B_{GPGA} (see Figure 1(b), in particular for values of the computational effort larger than 5). Finally, $DELAY$ outperforms standard GP on function K.

Table 2 reports the number of successful runs with their standard deviations for the same 100 independent runs of Figure 2 for $DELAY$, standard GP and GP+GD.

Table 2: Number of successes with their standard deviations calculated over 100 independent runs for standard GP, GP+GD and $DELAY$. Each row refers to a different test function, whose name is reported in the first column.

	std GP	GP + GD	$DELAY$
F	55 ($\sigma = 4.975$)	41 ($\sigma = 4.918$)	50 ($\sigma = 5.0$)
G	88 ($\sigma = 3.249$)	90 ($\sigma = 3.0$)	95 ($\sigma = 2.179$)
K	72 ($\sigma = 4.489$)	66 ($\sigma = 4.737$)	73 ($\sigma = 4.439$)

These results can be summarized as follows: standard GP has a slightly higher SR than $DELAY$ and a much higher SR than GP+GD on function F. The differences between $DELAY$ and standard GP on the F function seem to be

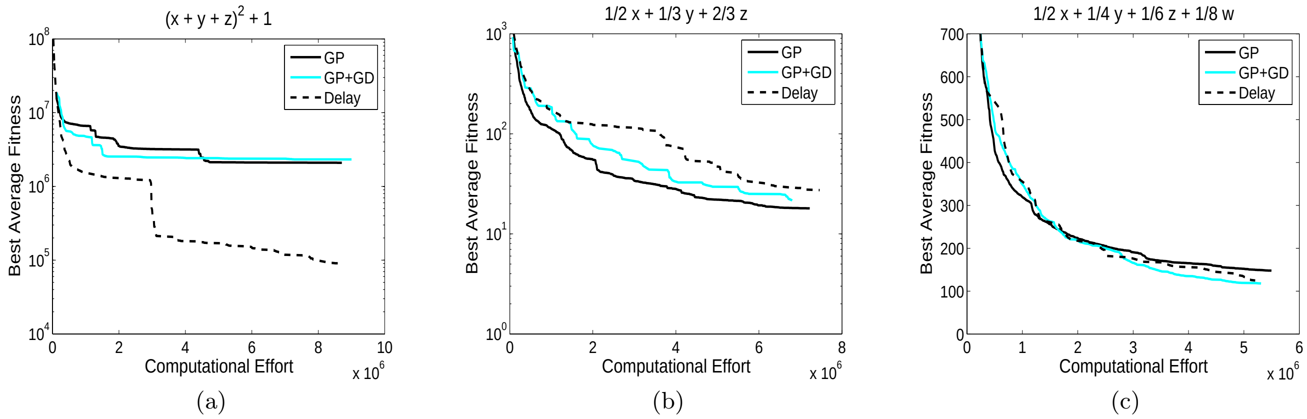


Figure 2: Best average fitness against computational effort over 100 independent runs for standard GP, GP+GD, and *DELAY*. In all algorithms the size of $\mathcal{K}$ has been set to 6. (a): results on function F; (b): results on function G; (c): results on function K.

marginal. Similarly, *DELAY* has a slightly higher SR than standard GP and GP+GD on function K, but, once again, standard deviations show that differences are not relevant. Finally, *DELAY* has a higher SR than standard GP and GP+GD on the G function.

In conclusion, the *DELAY* algorithm represents an improvement compared to B_{GPGA} since it often outperforms standard GP both from the point of view of the mean best fitness and of the success rate. In cases where *DELAY* is outperformed by standard GP, differences between the two algorithms appear to be marginal.

5. THE CINI ALGORITHM

The idea behind this variant of the algorithm derives from the observation that the B_{GPGA} algorithm unconditionally activates GA optimization also when GP by itself is being able to improve fitness significantly. The strategy adopted by this algorithm is to execute a turn of GA only when the execution of GP does not yield significant fitness improvements for a certain number of turns. For this reason, we have called this algorithm *CINI* (*Coevolve If No Improvement*). The pseudo-code describing the *CINI* algorithm is presented in Figure 3.

Strictly speaking, the *CINI* algorithm coincides with standard GP when GP alone is able to improve fitness quality in all turns. In practice, this event never occurred in any of the experiments.

5.1 Experimental Results

Figure 4 plots the values of the mean best population fitness against computational effort for *CINI*, standard GP and GP+GD for the three test functions.

As this figure clearly shows, *CINI* outperforms standard GP and GP+GD in searching all three test functions. The differences between the performances of these three algorithms are remarkable for functions F and G, while they are marginal for function K.

Table 3 shows the number of successful runs with their standard deviations for the same 100 independent runs of Figure 4 for *CINI*, standard GP and GP+GD.

CINI has a higher SR than standard GP and GP+GD for functions F and G and a slightly lower SR than standard GP for function K. A further attempt to improve the B_{GPGA} algorithm performance is presented in the next section.

```
Randomly initialize both GP and GA
populations;
repeat:
    1.  repeat:
            Execute one turn of GP
        until the best fitness of
              GP has not improved
              or termination condition
    2.  if not termination condition
        then
            Execute one turn of GA;
            Assign the constants represented
            by the best GA individual to the
            numeric terminals used by the GP
            individuals;
until termination condition.
```

Figure 3: Pseudo-code of the *CINI* algorithm.

Table 3: Number of successes with their standard deviations calculated over 100 indipendent runs for standard GP, GP+GD and *CINI*. Each row refers to a different test function, whose name is reported in the first column.

	std GP	GP + GD	*CINI*
F	55 ($\sigma = 4.975$)	41 ($\sigma = 4.918$)	56 ($\sigma = 4.964$)
G	88 ($\sigma = 3.249$)	90 ($\sigma = 3.0$)	91 ($\sigma = 2.862$)
K	72 ($\sigma = 4.489$)	66 ($\sigma = 4.737$)	68 ($\sigma = 4.665$)

6. THE AUTO ALGORITHM

The idea underlying this algorithm is that, at the end of each turn, one should proceed by running the "most promising" EA. Let Δf be the difference between the best fitness at

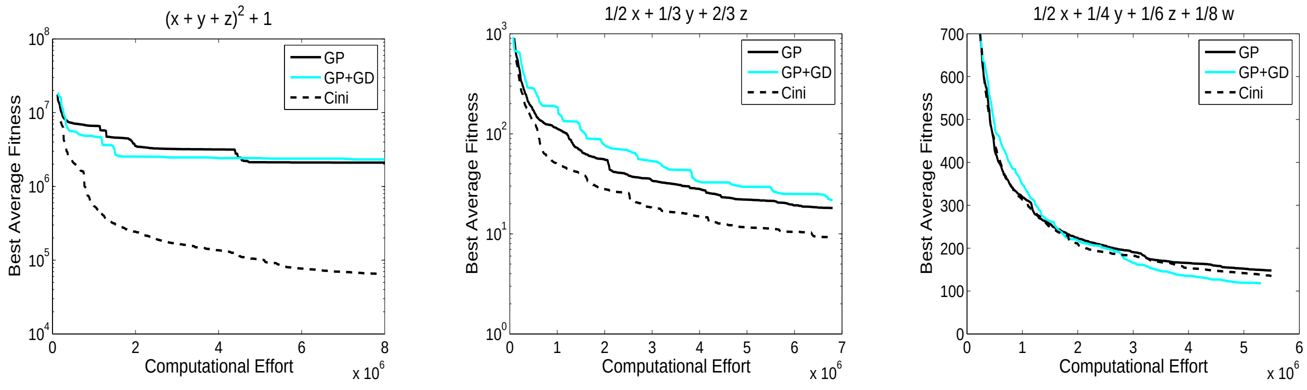


Figure 4: Best average fitness against computational effort over 100 independent runs for standard GP, GP+GD, and *CINI*. In all algorithms the size of $\mathcal{K}$ has been set to 6. (a): results on function F; (b): results on function G; (c): results on function K.

the beginning of a turn and the best fitness at the end of it. The EA that has had the highest Δf value in its last turn will be executed next. In order to make GA and GP performances comparable, GA turns and GP turns are assigned the same amount of computational effort. Thus, the number of generations composing a GA turn, called N_{genGA} can be dynamically computed as follows:

$$N_{genGA} = \frac{N_{genGP} \sum_{j=0}^{N_{GP}} l_j}{CE_{GA}}$$

where l_j is the number of nodes of the j^{th} individual in the GP population, CE_{GA} is the computational effort of a GA turn (see equation 1 in section 2.1) and N_{genGP} is the number of generations in a turn of GP. The pseudo-code describing the $AUTO$ algorithm is presented in Figure 5.

6.1 Experimental Results

Figure 6 shows the values of the mean best population fitness against computational effort for $AUTO$, standard GP and GP+GD over the three chosen test functions. $AUTO$ clearly outperforms both standard GP and GP+GD on function F, while the three algorithms have more or less the same performance on functions G and K. The number of successful runs with their standard deviations for the same 100 independent runs of Figure 6 for $AUTO$, standard GP and GP+GD are shown in Table 4 The three algorithms have more or less the same SR on all the three test functions studied. In fact, differences are not remarkable, with the only difference that GP+GD has a much lower SR on function F.

Table 4: Number of successes with their standard deviations calculated over for 100 indipendent runs for standard GP, GP+GD and $AUTO$. Each row refers to a different test function, whose name is reported in the first column.

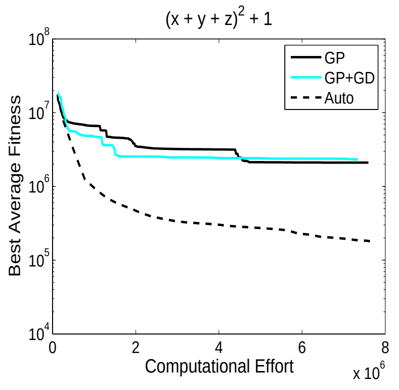
	std GP	GP + GD	$AUTO$
F	55 ($\sigma = 4.975$)	41 ($\sigma = 4.918$)	50 ($\sigma = 5.0$)
G	88 ($\sigma = 3.249$)	90 ($\sigma = 3.0$)	90 ($\sigma = 3.0$)
K	72 ($\sigma = 4.489$)	66 ($\sigma = 4.737$)	69 ($\sigma = 4.625$)

```
Randomly initialize both GP and GA
populations;
Evolve the GP population for 20 generations;
Calculate $\Delta f_{GP}$;
Calculate $N_{genGA} = \frac{N_{genGP} \sum_{j=0}^{N_{GP}} l_j}{CE_{GA}}$;
Evolve the GA population for $N_{genGA}$
generations;
Assign the constants represented by the best
GA individual to the numeric terminals used by
the GP individuals;
Calculate $\Delta f_{GA}$;

repeat:
    if ($\Delta f_{GP} > \Delta f_{GA}$)
    then
            Run GP for 20 generations;
            Calculate $\Delta f_{GP}$;
            Calculate $N_{genGA}$;
    else
            Run GA for $N_{genGA}$ generations;
            Assign the constants represented
            by the best GA individual to the
            numeric terminals used by the GP
            individuals;
            Calculate $\Delta f_{GA}$;

until termination condition.
```

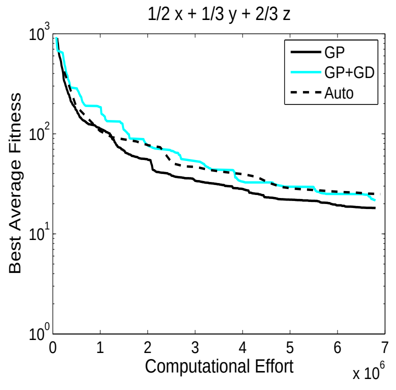
Figure 5: Pseudo-code describing the $AUTO$ algorithm.

7. A “REAL-WORLD” BIOMEDICAL APPLICATION

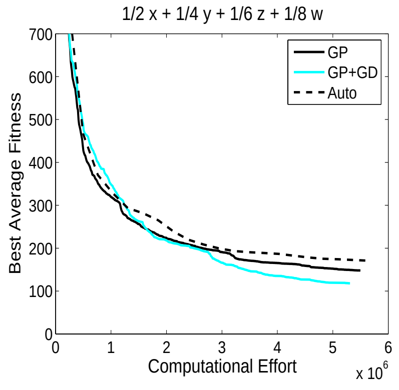
In this section, we present the results obtained by our co-evolutionary algorithms and standard GP on a “real-world” problem. This application consists of finding an algebraic expression matching the data contained in a matrix dataset



(a)



(b)



(c)

Figure 6: Best average fitness against computational effort over 100 independent runs for standard GP, GP+GD, and $AUTO$. In all algorithms the size of $\mathcal{K}$ has been set to 6. (a): results on function F; (b): results on function G; (c): results on function K.

D composed by 4107 rows and 346 columns with the values contained in a one-dimensional target vector V of size 4107. Each row i of the dataset D describes a different chemical compound by means of the values $\{D_{(i,1)}, D_{(i,2)}, ..., D_{(i,346)}\}$ of 346 different molecular descriptors. The corresponding value V_i is the docking energy of that compound. The goal is to find a function F such that $F(D_{(i,1)}, D_{(i,2)}, ..., D_{(i,346)}) = V_i$, $\forall i$ with $1 \leq i \leq 4107$. In other words, this problem aims at finding an algebraic expression to quantify the docking energy of a compound as a function of its molecular descriptors. The development of such an algebraic expression would represent a relevant contribution to the field of biotechnologies. In fact, the docking energy of a large number of drugs is strictly correlated with their ability to heal. Thus, being able to measure the docking energy would be extremely helpful in synthesizing drugs. For a more detailed description of the importance of measuring the docking energy as a function of the molecular descriptors of a chemical compound, see for instance [16].

7.1 Experimental Settings and Results

Given the particular shape of the training set, the unknown target function is defined over 346 variables. The fitness of the GP individuals has to be calculated over 4107 fitness cases, each one corresponding to a different row of the training set. All other GP and GA parameters were set to the same values as in the previous sections.

Table 5 shows the values of the mean best and average population fitness obtained by all the algorithms considered in this paper after each of them has spent a computational effort equal to 10^7.

These results are averages over 100 independent runs. They clearly show that *CINI* has been able to find the best solutions, since it is the algorithm with the best values of mean best fitness. *DELAY* has also yielded good values of mean best fitness. $AUTO$ has the best values of mean average population fitness. For all the algorithms we have studied, the values of the mean best fitness are better than the ones of standard GP.

8. CONCLUSIONS AND FUTURE WORK

This paper has introduced four variants of a cooperative coevolutionary algorithm. All these algorithms are characterized by the alternate evolution of two populations: the former is driven by a GP that evolves expressions that may solve the problem at hand, while the latter is driven by a GA which evolves numeric terminals for the GP. The characterizing feature of each algorithm is represented by the strategy it uses in alternating between the execution of the two evolutionary algorithms. The basic algorithm, called Basic-GPGA, statically alternates a fixed number of generations of GP with a (possibly different) number of generations of a GA. This process is iterated until a satisfactory solution to the problem at hand has been found or a pre-set value of the computational effort has been reached. The static nature of the Basic-GPGA algorithm is its main drawback, since the strategy behind alternation is not affected at all by the results obtained by either population in their most recent turns. To overcome this drawback, some variants of the Basic-GPGA algorithm have been developed. The second algorithm presented in this paper (called *DELAY*) starts working as the Basic-GPGA only after a large number of generations of GP have been initially executed. The third algorithm (called *CINI*) executes a turn of the GA population only when GP has not been able to improve fitness. Finally, the $AUTO$ algorithm dynamically chooses which population to evolve in the next turn by calculating

Table 5: Averages of the average and best population fitnesses reached at a fixed level of the computational effort by standard GP, B_{GPGA} and $AUTO$ for the Docking Energy Mapping problem described in the text (each calculated over 100 independent runs).

Algorithm	Avg. Fitness	Best Fitness
std GP	1122	1055
$GP + GD$	1043	951
B_{GPGA}	1258	1051
DELAY	1040	929
CINI	1211	909
$AUTO$	1016	966

which one has produced a higher fitness improvement in its last turn. Experiments have been performed on three simple test functions and one "real-world" biomedical data set. On all these benchmarks, our algorithms have shown interesting performances, improving standard GP performances in many cases. In particular, the *CINI* algorithm seems to be the most effective both on simple test functions and on the "real-world" application, where it has been able to yield promising results.

Future work includes deepening the analysis of the results, to justify the rather different outcomes of our algorithms in the tests we have reported, as well as extending their application to other domains, such as image processing. A longer-term goal is to evolve our coevolutionary framework up to defining a cooperative coevolutionary model which can run on GRID architectures to perform (possibly many different) optimizations concurrently over a large set of geographically distributed machines, with a high degree of self-organization.

9. REFERENCES

[1] T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, Oxford, 1996.

[2] R. K. Belew, J. McInerney, and N. N. Schraudolph. Evolving networks: using the genetic algorithm with connectionist learning. In *Proc. of the 2nd Artificial Life Conference*, pages 511–547, 1991.

[3] S. Cagnoni, D. Rivero, and L. Vanneschi. A purely evolutionary memetic algorithm as a first step towards symbiotic coevolution. In *Proc. of the IEEE Congress on Evolutionary Computation (CEC 2005)*, pages 1156 – 1163, 2005.

[4] J. M. Daida, R. Bertram, S. Stanhope, J. Khoo, S. Chaudhary, and O. Chaudhary. What makes a problem GP-hard? Analysis of a tunably difficult problem in genetic programming. *Genetic Programming and Evolvable Machines*, 2:165–191, 2001.

[5] A. Eiben and M. Jelasity. A critical note on experimental research methodology in EC. In *Proc. of the Congress on Evolutionary Computation (CEC 2002)*, pages 582–587, 2002.

[6] F. Fernández, M. Tomassini, and L. Vanneschi. An empirical study of multipopulation genetic programming. *Genetic Programming and Evolvable Machines*, 4(1):21–52, 2003.

[7] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[8] F. Gomez, D. Burger, and R. Miikkulainen. A neuro-evolution method for dynamic resource allocation on a chip multiprocessor. In *Proc. of the International Joint Conference on Neural Networks (IJCNN 2001)*, volume 4, pages 2355–2360, 2001.

[9] J. H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, 1975.

[10] M. Keijzer. Improving symbolic regression with interval arithmetic and linear scaling. In *Genetic Programming, Proceedings of the 6th European Conference, EuroGP 2003*, volume 2610 of *LNCS*, pages 71–83. Springer, 2003.

[11] J. R. Koza. Genetic evolution and co-evolution of computer programs. In C. Taylor, C. Langton, J. D. Farmer, and S. Rasmussen, editors, *Artificial Life II*, volume X, pages 603–629. Addison-Wesley, 1991.

[12] J. R. Koza. *Genetic Programming*. The MIT Press, Cambridge, Massachusetts, 1992.

[13] S. Luke and L. Panait. Is the perfect the enemy of the good? In *Proc. of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 820–828, 2002.

[14] D. Moriarty and R. Miikkulainen. Hierarchical evolution of neural networks. In *Proc. of the IEEE World Congress on Computational Intelligence (WCCI 1998)*, pages 428–433, 1998.

[15] S. Nolfi, D. Parisi, and J. L. Elman. Learning and evolution in neural networks. *Adaptive Behavior*, 3(1):5–28, 1994.

[16] E. Papaleo, P. Fantucci, M. Vai, and L. De Gioia. Three-dimensional structure of the catalytic domain of the yeast β-(1,3)-glucan transferase gas: a molecular modeling investigation. *Journal of Molecular Modeling*, 12(2):237–248, 2005.

[17] J. Paredis. Coevolutionary computation. *Artificial Life*, 2(4):355–375, 1995.

[18] M. Potter and K. De Jong. A cooperative coevolutionary approach to function optimization. In *Proc. 3rd Conf. on Parallel Problem Solving from Nature (PPSN III)*, volume 866 of *LNCS*, pages 249–257. Springer, 1994.

[19] M. Roberts and E. Claridge. Cooperative coevolution of image feature construction and object detection. In *Proc. 8th Conf. on Parallel Problem Solving from Nature (PPSN VIII)*, volume 3242 of *LNCS*, pages 902–911. Springer, 2004.

[20] M. Roberts and E. Claridge. A multistage approach to cooperatively coevolving feature construction and object detection. In *Applications of Evolutionary Computing, EvoWorkshops 2005*, volume 3449 of *LNCS*, pages 396–406. Springer, 2005.

[21] S. M. Ross. *Introduction to Probability and Statistics for Engineers and scientists*. Academic Press, 2000.

[22] C. Ryan and M. Keijzer. An analysis of diversity of constants of genetic programming. In *Genetic Programming, 6th European Conference, EuroGP2003*, volume 2610 of *LNCS*, pages 409–418. Springer, 2003.

[23] A. Topchy and W. F. Punch. Faster genetic programming based on local gradient search of numeric leaf values. In *Proc. of the Genetic and Evolutionary Computation Conference (GECCO 2001)*, pages 155–162, 2001.

[24] A. Valsecchi. Algoritmi coevolutivi per problemi di regressione. Master's thesis, Università di Milano-Bicocca, Milano, 2005.

[25] L. Vanneschi. *Theory and Practice for Efficient Genetic Programming*. Ph.D. thesis, Faculty of Science, University of Lausanne, Switzerland, 2004.

[26] R. P. Wiegand. *An Analysis of Cooperative Coevolutionary Algorithms*. Ph.D. thesis, George Mason University, Fairfax, VA, USA, 2004.

[27] B. T. Zhang and H. Mühlenbein. Evolving optimal neural networks using genetic algorithms with Occam's razor. *Complex Systems*, 7(3):199–220, 1993.