HAL
open science

Teams of Genetic Predictors for Inverse Problem Solving

Michael Defoin Platel, Malik Chami, Manuel Clergue, Philippe Collard

To cite this version:

Michael Defoin Platel, Malik Chami, Manuel Clergue, Philippe Collard. Teams of Genetic Predictors for Inverse Problem Solving. EuroGP 2005, 2005, Lausanne, Switzerland. pp.341-350, 10.1007/b107383 . hal-00159604

HAL Id: hal-00159604

https://hal.science/hal-00159604v1

Submitted on 3 Jul 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Teams of Genetic Predictors for Inverse Problem Solving

Michael Defoin Platel[1], Malik Chami[2], Manuel Clergue[1], and Philippe Collard[1]

[1] Laboratoire I3S, UNSA-CNRS, Sophia Antipolis, France
[2] Laboratoire d'Oceanographie de Villefranche sur Mer, France

Abstract. Genetic Programming (GP) has been shown to be a good method of predicting functions that solve inverse problems. In this context, a solution given by GP generally consists of a sole predictor. In contrast, Stack-based GP systems manipulate structures containing several predictors, which can be considered as *teams of predictors*. Work in Machine Learning reports that combining predictors gives good results in terms of both quality and robustness. In this paper, we use Stack-based GP to study different cooperations between predictors. First, preliminary tests and parameter tuning are performed on two GP benchmarks. Then, the system is applied to a real-world inverse problem. A comparative study with standard methods has shown limits and advantages of teams prediction, leading to encourage the use of combinations taking into account the response quality of each team member.

1 Introduction

A direct problem describes a Cause-Effect relationship, while an inverse problem consists in trying to recover the causes from a measure of effects. Inverse problems are often far more difficult to solve than direct problems. Indeed, the amount and the quality of the measurements are generally insufficient to describe all the effects and so to retrieve the causes. Moreover, since different causes may produce the same effects, the solution of the problem may be not unique. In many scientific domains, solving an inverse problem is a major issue and a wide range of methods, either analytic or stochastic, are used. In particular, recent work [6][4] has demonstrated that Genetic Programming (GP) is a good candidate.

GP applies the Darwinian principle of survival of the fittest to the automatic discovery of programs. With few hypothesis on the instructions set used to build programs, GP is an universal approximator [16] that can learn an arbitrary function given a set of training examples. For GP, as for Evolutionary Computation in general, the way individuals are represented is crucial. The representation induces choices about operators and may strongly influence the performance of the algorithm. The emergence of GP in the scientific community arose with the use, *inter alia*, of a tree-based representation, in particular with the use of the Lisp language in the work of Koza [8]. However, there are GP systems manipulating linear structures, which have shown experimental performances equivalent to Tree GP (TGP) [1]. In contrast to TGP, Linear GP (LGP) programs are sequences of instructions of an imperative language (C, machine code, ...). The evaluation of a program can not be performed in a recursive way and so needs to use extra memory mechanisms to store partial computations. There are at least two kind of

M. Keijzer et al. (Eds.): EuroGP 2005, LNCS 3447, pp. 341–350, 2005.
© Springer-Verlag Berlin Heidelberg 2005

LGP implementation. In the first one [1], a finite number of registers are used to store the partial computations and a particular register is chosen to store the final result. In the other one, the stack-based implementation [12],[15] and [3], the intermediate computations are pushed into an operand stack and the top of stack gives the final result. It is important to note that in TGP, an individual corresponds to a sole program and its evaluation produces a unique output. In LGP, an individual may be composed of many independent sub-programs and its evaluation may produce several outputs and some of them may be ignored in the final result. In this paper, we propose to combine those sub-programs into a team of predictors.

The goal of Machine Learning (ML) is to find a predictor trained on set of examples that can approximate the function that generated the examples. Several ML methods are known to be universal approximators, that is they can approximate a function arbitrarily well. Nevertheless, the search of optimal predictors might be problematic due to the choice of inadequate architecture but also to over-fitting on training cases. Over-fitting occurs when a predictor reflects randomness in the data rather than underlying function properties, and so it often leads to poor generalization abilities of predictors. Several methods have been proposed to avoid over-fitting, such as model selection, to stop training or combining predictors, see [14] for a complete discussion. In this study, we mainly focus on combining predictor methods, also called ensemble, or committee methods. In a committee machine, a team of predictors is generated by means of a learning process and the overall predictions of the committee machine is the combination of the predictions of the individual team members. The idea here, is that a team may exhibit performances unobtainable by a single individual, because the errors of the members might cancel out when their outputs are combined. Several schemes for combining predictors exist, such as simple averaging, weighted combination, mixtures of experts or boosting. In practice, ANN ensembles have already been strongly investigated by several authors, see for instance [9]. In the GP field, there has been some work on the combination of predictors, see for examples [17][11][2].

In this paper, our goal is initially to improve the performance of stack-based GP systems by evolving teams of predictors. The originality of this study is that teams have a dynamic number of members that can be managed by the system. In Section 2, we describe different ways to combine predictors and how to implement them using a stack. In Section 3, evolutionary parameters are tuned and the performances of several combinations are tested on two GP benchmarks and on a real-world inverse problem. Finally, in Section 4, we investigate the relationship between uncontrolled growth of program size in GP and dynamical size of teams.

2 Teams of Genetic Predictors

2.1 Stack-Based GP

In stack-based GP, numerical calculations are performed in *Reverse Polish Notation*. According to the implementation proposed by Perkis[12], an additional type of closure

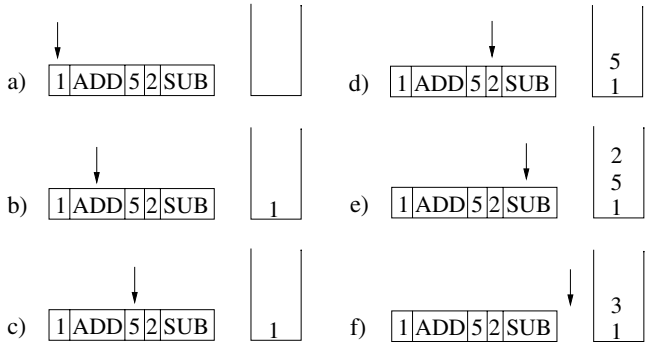

Fig. 1. Evaluation of a program in Stack-based GP

constraint is imposed on functions: they are defined to do nothing when arity is unsatisfied by the current state of the operand stack. In Figure 1, a basic example of program execution is presented. We can see the processing of the program "1,ADD,5,2,SUB". Initialization phase corresponds to step *a* where the stack is cleared. During steps *b*, *d* and *e*, numerical constants 1, 5 and 2, respectively, are pushed onto the operand stack. During step *f*, since the operand stack stores enough data (at least two), the "SUB" instruction is computed and the result ($3 = 5 - 2$) is pushed onto the stack. During step *c* the stack contains only one value, so computation of the "ADD" instruction is impossible: the instruction is simply skipped with no effect on the operand stack. In most cases, problems addressed using GP are defined with multiple fitness cases and a compilation phase can easily removed those unexpected instructions to speed up evaluation.

Let us note that after step *f*, the operand stack contains the values 1 and 3. This means that there are two independent sub-programs, "1" and "5,2,SUB", in the sequence "1,ADD,5,2,SUB". We propose to make the whole set of sub-programs involved in the fitness evaluation. The idea is to combine the results of each sub-programs, i.e. all the elements of the final stack, according to the nature of the problem addressed. For example, in the case of a *Symbolic Regression Problem*, any linear combination of the sub-programs outputs may be investigated. Using this kind of evaluation, the evolutionary process should be able to tune the effect of the genetic operators by changing the number and the nature of sub-programs and so modifying the contribution of each of them to the final fitness. Moreover the amount of useless code usually found in stack-based representation programs is significantly decreased leading to improved performance. This approach may be viewed as the evolution of teams of predictors corresponding to the combination of sub-programs.

2.2 Different Combinations

Let us consider a program with m instructions giving a team T having $k \in [1, m]$ predictors p_i. The team output $O(T)$ consists of a combination of the k predictors outputs $o(p_i)$. Several combinations have been tested :

Team combination	$O(T)$
`Sum` : Sum of each predictor output	$\sum_{i=1}^{k} o(p_i)$
`AMean` : Arithmetic mean of predictors outputs	$\frac{1}{k}\sum_{i=1}^{k} o(p_i)$
`Pi` : Product of each predictor output	$\prod_{i=1}^{k} o(p_i)$
`GMean` : Geometric mean of each predictor output	$\sqrt[k]{\prod_{i=1}^{k} o(p_i)}$
`EMean` : Mean of each predictor output weighted by their error	$\frac{1}{\sum_{i=1}^{k} w_i}\sum_{i=1}^{k} w_i o(p_i)$
`WTA` : Winner predictor output Takes All	$o(argmin_{p_i, i\in[1,k]}(E(p_i)))$
`Top` : Top of stack predictor output	$o(p_k)$

with $E(p_i)$ the training error of p_i and with $w_i = e^{-\beta E(p_i)}$, β a positive scaling factor.

3 Experimental Results

3.1 Symbolic Regression

In this section, the evolutionary parameters' tuning is extensively investigated on a Symbolic Regression Problem. We choose the Poly 10 problem [13], where the target function is the 10-variate cubic polynomial $x_1x_2 + x_3x_4 + x_5x_6 + x_1x_7x_9 + x_3x_6x_{10}$. In this study, the fitness is the classical Root Mean-Square Error. The dataset contains 50 test points and is generated by randomly assigning values to the variables x_i in the range $[-1, 1]$. We perform 50 independent runs with various mutation and crossover rates. Populations of 500 individuals are randomly created according to a maximum creation size of 50. The instructions set contains: the four arithmetic instructions ADD, SUB, MUL, DIV, the ten variables $\mathrm{X}_1 \ldots \mathrm{X}_{10}$ and one stack-based GP specific instruction DUP which duplicates the top of the operand stack. The evolution, with elitism, maximum program size of 500, 16-tournament selection, and steady-state replacement, takes place over 100 generations [1]. We use a statistical unpaired, two-tailed t-test with 95% confidence to determine if results are significantly different.

In Table 1, the best performances on the Poly 10 problem with different combinations of predictors are presented, using the best settings of evolutionary parameters found, crossover rate varying from 0 to 1.0 and mutation rate from 0 to 2.0. Let us notice that a mutation rate of 1.0 means that each program involved in reproduction will undergo, on average, one insertion, one deletion and one substitution. In the first row, results obtained using a Tree GP implementation[2] are reported in order to give an absolute reference. We see that the classical `Top` of stack and `WTA` methods work badly, which is to be expected, since in this case, teams' outputs correspond to single predictor outputs. The `Sum` and `AMean` methods report good results compared to the `Tree` method. In contrast, the `Pi` and `GMean` methods are not suitable for this problem, perhaps because the Poly 10 problem is easiest to decompose as a sum. Finally, `EMean` undoubtedly outperforms

[1] In a steady state system, the generation concept is somewhat artificial and is used only for comparison with generational systems. Here, a generation corresponds to a number of replacement equal to the number of individual in the population, i.e. 500.

[2] We note that an optimization of parameters has been also performed for Tree GP.

Table 1. Best Results on Poly 10

Team	Train			
	Mean	Std Dev	Best	Worst
Tree	0.250	0.072	0.085	0.407
Top	0.353	0.105	0.172	0.520
WTA	0.443	0.048	0.347	0.541
Sum	0.173	0.031	0.120	0.258
AMean	0.165	0.034	0.109	0.251
Pi	0.361	0.063	0.220	0.456
GMean	0.222	0.027	0.151	0.301
EMean	**0.066**	**0.017**	**0.029**	**0.141**

Table 2. Results on Mackey-Glass (10^{-2})

Team	Train		Valid		Test	
	Mean	Std Dev	Mean	Std Dev	Mean	Std Dev
Tree	**0.61**	**0.14**	1.06	0.43	1.21	0.71
Top	1.09	0.24	0.99	0.35	0.95	0.33
WTA	1.87	1.04	1.97	1.06	1.95	0.36
Sum	0.64	0.05	0.83	0.21	1.38	0.39
AMean	0.63	0.06	0.81	0.20	0.83	0.40
Pi	0.71	0.15	0.84	0.27	1.10	0.44
GMean	0.67	0.04	0.78	0.14	0.76	0.15
EMean	**0.59**	**0.03**	**0.74**	**0.07**	**0.71**	**0.06**

other methods. We note that results presented here correspond to a scaling factor β of 10. In Figure 2, the average training error is plotted as a function of β. The training error is clearly related to β and gives a minimum for $\beta = 10$.

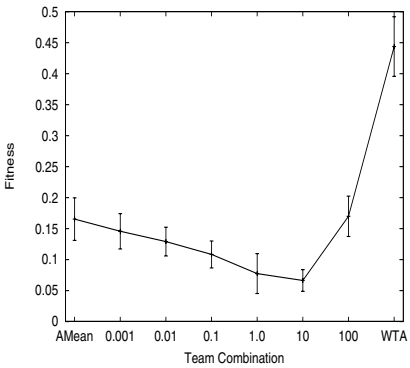

Fig. 2. Average train error for AMean, WTA and EMean with different β on Poly 10 problem

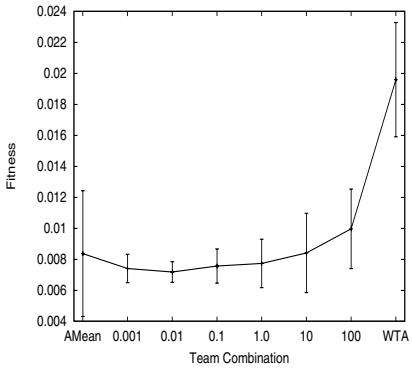

Fig. 3. Average test error for AMean, WTA and EMean with different β on Mackey-Glass problem

3.2 Chaotic Time Series

In this Section, we choose the IEEE benchmark Mackey-Glass chaotic time series (see http://neural.cs.nthu.edu.tw/jang/benchmark/, τ=17, 1201 data points, sampled every 0.1) to examine the performances of different teams in the context of over-fitting. This problem has already been tested in LGP, see [10] for example, and seems to have sufficient difficulty to allow appearance of over-fitting behaviors.

We have discarded the first 900 points of the dataset to remove the initial transients and we have decomposed the last 300 in three sets of same size, the Train, Validation and Test sets. The goal for GP is, given 8 historical values, to predict the value at time $t + 1$. Thus, each set contains 100 vectors with values at times $t - 128$, $t - 64$, $t - 32$,

$t-16$, $t-8$, $t-4$, $t-2$ and t. The evolutionary parameters are identical to those used to solve the Poly 10 problem. The instruction set is limited to handle only the 8 inputs and an Ephemeral Random Constant (cf. [8]) in the range [-1, 1] has been added onto the instructions set.

In Table 2, the performances on the Mackey-Glass problem with different combinations of predictors are presented, using the best settings of evolutionary parameters found on Poly 10 problem. The train errors reported here tend to confirm the previous results on Poly 10. Indeed, the `Top` and `WTA` combinations give the worst results while, with other combinations, we have obtained performances equivalent to Tree GP. We see important variations between Train and Test errors, in particular for `Tree` and `Sum` combination. However we have found only 2 or 3 programs with very bad test errors (among 50 per each combination) that are responsible of the main part of these variations. The `EMean` team has obtained the best results, for both train and test errors, and seems to over-fit less than the others. It is important to note that the programs discovered during our different experiments on the Mackey-Glass problem with Stack GP have approximatively the same number of instructions than programs obtained with the Tree GP implementation.

The results of `EMean` combination presented here correspond to a scaling factor β of 0.01. In Figure 3, the average test error is plotted as a function of β. The test error is clearly related to β and gives a minimum for β=0.01.

3.3 Inverse Problem

The real-world inverse problem addressed here deals with atmospheric aerosol characteristics. An accurate knowledge of these characteristics is central, for example, in satellite remote sensing validation. A large amount of data is necessary to train inverse models properly. The teaching phase should be performed with truthful data so that inverse models are learned with the influence of natural variabilities and measurement errors. However, according to the inverse problem to deal with, it may be difficult to gather enough measurements to cover the data space with sufficient density. It is common to use simulated data for the training step and to add some geophysical noise in the data set to account for measurement errors [7][4]. Therefore, in this paper, synthetic data obtained with a radiative transfer model were used to perform the learning step.

We want to inverse a radiative transfer model, called *Ordres Successifs Ocean Atmosphere* (OSOA) model, which is described in detail by Chami et al [5]. The direct model solves a radiative transfer equation (RTE) by the successive orders of scattering method for the ocean-atmosphere system. It takes into account the multiple scattering events and the polarization state of light for both atmospheric and oceanic media. The atmosphere is a mixture of molecules and aerosols. Aerosols are supposed to be homogeneous spheres. The aerosol model is defined (refractive index and size distribution) and its optical properties (phase function, single scattering albedo) are computed using Mie theory. The air-water interface is modeled as a planar mirror. Consequently, the reflection by a rough surface is not taken into account in the radiative transfer code. The OSOA outputs the angular distribution of the radiance field and its degree of polarization at any desired level (any depth, the surface or the top of atmosphere).

In this study, the solar zenith angle is fixed to 70 degrees. Each row of the dataset corresponds to sky radiances at 440 nm, 675 nm and 870 nm for 10 scattering angles

Table 3. Results on Inverse Problem

	Train		Valid		Test	
Team	Mean	Std Dev	Mean	Std Dev	Mean	Std Dev
`Tree`	0.018	0.004	0.018	0.004	0.021	0.006
`Top`	0.035	0.006	0.032	0.006	0.035	0.006
`WTA`	0.040	0.013	0.037	0.012	0.041	0.014
`Sum`	0.019	0.004	0.018	0.004	0.020	0.005
`AMean`	0.021	0.004	0.019	0.004	0.022	0.005
`Pi`	0.022	0.004	0.021	0.004	0.023	0.005
`GMean`	0.017	0.002	0.019	0.006	0.022	0.009
`EMean`	**0.013**	**0.002**	**0.012**	**0.001**	**0.014**	**0.003**

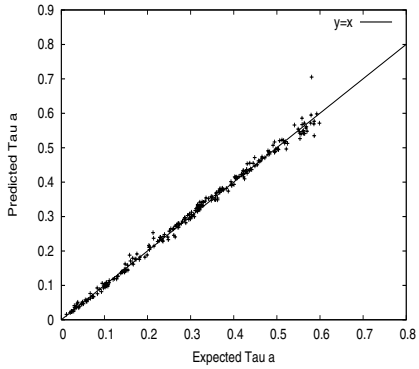

Fig. 4. Scatter plot of Expected vs Predicted τ_a on test data for the best team found.

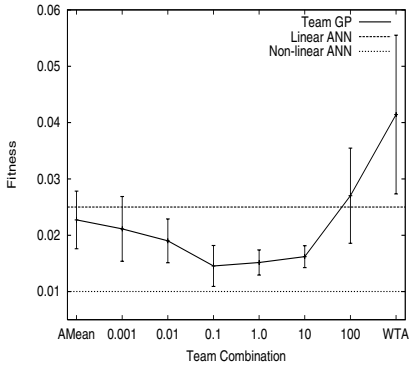

Fig. 5. Average test error for `AMean`, `WTA` and `EMean` with different β on inverse problem.

ranging from 3 to 150 degrees. From these 30 inputs, the inverse model has to retrieve the aerosol optical depth τ_a at 675 nm. To avoid over fitting, we have decomposed the dataset into three parts, 500 samples are devoted to the learning step, 250 to validation and test steps. The evolutionary parameters are identical to those used to solve the Poly 10 problem. The instruction set is extended to handle the 30 inputs. Finally, a LOG instruction ($ln|x|$) and an Ephemeral Random Constant (cf. [8]) in the range [-10, 10] have been also added onto the instructions set.

In Table 3, the performances on the inverse problem with different combinations of predictors are presented. As previously, for similar reasons, we see that the classical `Top` of stack and `WTA` methods works badly. Contrary to results reported for Poly 10 problem, `EMean` is the only method that significantly outperforms `Tree`. We note that results presented here correspond to a scaling factor β fixed to 0.1. Figure 4 shows performances on test data of the best team found with `EMean` and $\beta = 0.1$. We see that a good agreement is obtained between expected and predicted τ_a giving a nearly perfect inversion.

In Figure 5, the average test error is plotted as a function of β. The test error is clearly related to β and gives a minimum for $\beta = 0.1$. We have also trained two ANNs

on this inverse problem : one ANN with feed-forward topology and no hidden layer (to handle linear relationships only) and one ANN with feed-forward topology and one hidden layer (optimal number of nodes found experimentally). The corresponding test errors are also plotted in Figure 5. We see that the use of teams of predictors improves the performances of the GP system in such a way that it outperforms a linear ANN. Nevertheless, the accuracy of a non-linear ANN can not be obtained with the parameters used in this study. Let us notice that they are many sophisticated techniques, such as Automatic Define Functions, Demes, ..., well known in the GP community, that could significantly improve the results presented here. Our aim was not to compare ANN and GP, but rather to quantify the benefit of optimizing the β parameter.

4 Discussion

In this paper, a team of predictors corresponds to a combination of the sub-programs of a stack-based GP individual. Contrary to previous studies addressing teams of predictors in GP, here, the number of members of a team is not apriori fixed but can change during the evolutionary process. We had hoped that the dynamic of the algorithm would optimize the number of members. Unfortunately, the team size tends to increase quickly as of the early generations and is strongly correlated to the size of individuals. Moreover, it is well known that GP suffers from an uncontrolled growth of the size of individuals, a phenomenon called bloat. Thus, the team size can not be directly managed by the system. In Figure 6, the average number of predictors on the inverse problem is reported for `EMean` with $\beta = 0.1$. The limit, around 200 predictors, is probably due to the 'maximum allowed size of programs' parameter (500 instructions). We have also reported the number of predictors having a weight w_i (almost) equal to zero and the number of predictors whose weight is positive. We see that around a third of the predictors are all but eliminated from teams. So, the use of combinations of predictors that take into account the response quality of each team member gives a way to control team size.

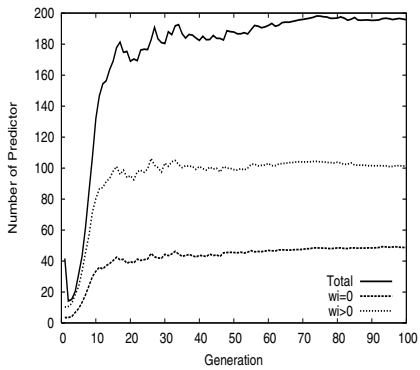

Fig. 6. Evolution of average number of predictors with $\beta = 0.1$ on inverse problem.

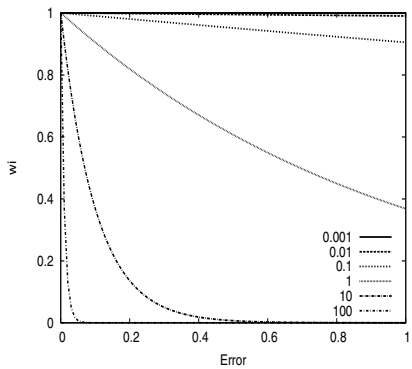

Fig. 7. Weights w_i as a function of the predictors error with $\beta = 0.001, 0.01, 0.1, 1, 10$ and 100.

However, we have seen that the best tuning of β depends on the problem addressed. Figure 7 shows the weights w_i as a function of the predictor errors for different β. We have limited the error to the range $[0, 1]$. We see that for small values of β, weights are almost equal to 1, as for the `AMean` combination, whereas when β is equal to 100, the majority of weights are null, as for `WTA` combination. From Figures 2, 3 and 5, we see that `EMean` is able to cover the entire performance spectrum between `AMean` and `WTA`.

5 Conclusion

In this paper, our aim is not to compare our work with other learning methods, such as boosting for example, but rather to improve the performance of stack-based GP systems. Thus, we start with one of their main drawbacks : they provide many outputs, that is sub-programs, instead of only one. A naive way to overcome this is to keep only one arbitrary output as the output of the program. This method exhibits very poor results, notwithstanding the waste of resources needed to evolve part of programs that are never used. Keeping only the best sub-program leads to even worse results, as the system undergoes premature convergence. Taking the arithmetic mean or the sum of sub-programs outputs gives our system the same performances as a tree-based GP system, which evolves individual predictors.

The best results we obtain are when the output of the program is a weighted sum of the sub-programs, where each sub-program receives a weight depending on its individual performance. This way programs are not penalized by bad sub-programs, since they do not contribute, or only lightly, to the final output. Moreover, this supplementary degree of freedom promotes the emergence of dynamically sized teams. Indeed, even if the total number of predictors in a team is strongly correlated to the maximum number of instructions in programs, only some of them contribute to the program output and so really participate in the team. The number of such sub-programs is free. Genetic operations may cause it to vary, with the introduction of a new "good" sub-program or the destruction of a former "good" one.

This paper represents the first step of our work and empirical results should be confirmed by experiments on other problems and theoretical studies. Moreover, some choices we have made are arbitrary, such as the use of a linear combination of sub-programs. Other types of combinations, e.g. logarithmic ones may improve performances. Also, we need to better understand how the value of β influences performance, and its effect on team composition.

References

1. M. Brameier and W. Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5(1):17–26, 2001.
2. M. Brameier and W. Banzhaf. Evolving teams of predictors with linear genetic programming. *Genetic Programming and Evolvable Machines*, 2(4):381–407, 2001.

3. W. S. Bruce. The lawnmower problem revisited: Stack-based genetic programming and automatically defined functions. In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 52–57, Stanford University, CA, USA, 13-16 1997. Morgan Kaufmann.
4. M. Chami and D. Robilliard. Inversion of oceanic constituents in case i and case ii waters with genetic programming algorithms. *Applied Optics*, 40(30):6260–6275, 2002.
5. M. Chami, R. Santer, and E. Dilligeard. Radiative transfer model for the computation of radiance and polarization in an ocean-atmosphere system: polarization properties of suspended matter for remote sensing. *Applied Optics*, 40(15):2398–2416, 2001.
6. P. Collet, E. Lutton, F. Raynal, and M. Schoenauer. Polar IFS+parisian genetic programming=efficient IFS inverse problem solving. *Genetic Programming and Evolvable Machines*, 1(4):339–361, 2000.
7. L. Gross, S. Thiria, R. Frouin, and B. G. Mitchell. Artificial neural networks for modeling the transfer function between marine reflectances and phytoplankton pigment concentration. *J. Geophys. Res.*, C2(105):3483–3495, 2000.
8. J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.
9. A. Krogh and J. Vedelsby. Neural network ensembles, cross validation, and active learning. *NIPS*, 7:231–238, 1995.
10. W. B. Langdon and W. Banzhaf. Repeated sequences in linear gp genomes. In *Late breaking paper at GECCO'2004*, Seattle, USA, June 2004.
11. G. Paris, D. Robilliard, and C. Fonlupt. Applying boosting techniques to genetic programming. In *Artificial Evolution 5th International Conference, Evolution Artificielle, EA 2001*, volume 2310 of *LNCS*, pages 267–278, Creusot, France, October 29-31 2001. Springer Verlag.
12. 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 1994. IEEE Press.
13. R. Poli. A simple but theoretically-motivated method to control bloat in genetic programming. In *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 200–210, Essex, 14-16 April 2003. Springer-Verlag.
14. W. Sarle. Stopped training and other remedies for overfitting. In *Proceedings of the 27th Symposium on Interface*, 1995.
15. K. Stoffel and L. Spector. High-performance, parallel, stack-based genetic programming. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 224–229, Stanford University, CA, USA, 28–31 1996. MIT Press.
16. Xin Yao. Universal approximation by genetic programming. In *Foundations of Genetic Programming*, Orlando, Florida, USA, 13 1999.
17. Byoung-Tak Zhang and Je-Gun Joung. Time series prediction using committee machines of evolutionary neural trees. In *Proceedings of the Congress of Evolutionary Computation*, volume 1, pages 281–286. IEEE Press, 1999.