IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 27, 2020, accepted February 10, 2020, date of publication February 21, 2020, date of current version March 3, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2975753

Time and Individual Duration in Genetic Programming

FRANCISCO FERNÁNDEZ DE VEGA[1], (Senior Member, IEEE), GUSTAVO OLAGUE[2], (Senior Member, IEEE), DANIEL LANZA[3], FRANCISCO CHÁVEZ DE LA O[1], WOLFGANG BANZHAF[4], ERIK GOODMAN[4], JOSE MENENDEZ-CLAVIJO[2], AND AXEL MARTINEZ[2]

[1]Grupo de Evolución Artificial, Universidad de Extremadura, 06800 Mérida, Spain
[2]EvoVisión Laboratory, CICESE Research Center, Ensenada 22860, Mexico
[3]Information Technology Department, CERN, 1211 Geneva, Switzerland
[4]BEACON Center, Michigan State University, East Lansing, MI 48824, USA

Corresponding author: Gustavo Olague (olague@cicese.mx)

This work was supported in part by the Spanish Ministry of Economy and Competitiveness (Deep-Bio-Uex) under Project TIN2017-85727-C4-4-P, in part by the Regional Government of Extremadura, Department of Commerce and Economy, co-funded by the European Regional Development Fund (a Way to Build Europe) under Grant G15068IB16035, and in part by the Center for Scientific Research and Higher Education at Ensenada (CICESE) through the Programación Cerebral Aplicada al Estudio del Pensamiento y la Visión under Project 634-128.

ABSTRACT This paper presents a new way of measuring complexity in variable-size-chromosome-based evolutionary algorithms. Dealing with complexity is particularly useful when considering bloat in Genetic Programming. Instead of analyzing size growth, we focus on the time required for individuals' fitness evaluations, which correlates with size. This way, we consider time and space as two sides of a single coin when devising a more natural method for fighting bloat. We thus view the problem from a perspective that departs from traditional methods applied in Genetic Programming. We have analyzed first the behavior of individuals across generations, taking into account their fitness evaluation times, thus providing clues about the general practice of the evolutionary process when modern parallel and distributed computers are used to run the algorithm. This new perspective allows us to understand that new methods for bloat control can be derived. Moreover, we develop from this framework a first proposal to show the usefulness of the idea: to group individuals in classes according to computing time required for evaluation, automatically accomplished by parallel and distributed systems without any change in the underlying algorithm, when they are only allowed to breed within their classes. Experimental data confirms the strength of the approach: using computing time as a measure of individuals' complexity allows control of the natural size growth of genetic programming individuals while preserving the quality of solutions in both the parallel and sequential versions of the algorithm.

INDEX TERMS Bloat, computing time, genetic programming.

I. INTRODUCTION

Genetic Programming (GP) became popular from the mid-nineties onwards, after Koza published his first book on the topic [1]. This machine learning technique has widely demonstrated capabilities for addressing hard, real-world problems, and in the case of tree-based GP, a challenge remains: the bloat phenomenon, see [2]–[4].

The associate editor coordinating the review of this manuscript and approving it for publication was Nuno Garcia.

A. PROBLEM STATEMENT

According to [5], fitness improvement correlates with the inherent bloating behavior present in any evolutionary approach using variable-size chromosomes. Although many authors have addressed this problem from different perspectives, no perfect solution exists, and there is still room for improvement.

This paper aims to provide new perspectives for fighting bloat, which may, in the future, allow the development of a series of new bloat control methods based on function (chromosome) evaluation time. Although there is a correlation

between an individual's size and its evaluation time, the latter also depends on the underlying hardware infrastructure that is used to carry out the evaluation. In this paper, we explore the connection with parallel and distributed model architectures. Although island models have been analyzed before in this context ([6], [7]), these previous approaches relied more on the spatial structure of the models, while in this work we are more interested in the standard GP algorithm, whether implemented sequentially or in parallel.

By analyzing the intrinsic behavior of the algorithm, we show the space-time connection (correlation between individual size and evaluation time) in the bloat phenomenon [8]. We also show how this new relationship could benefit EAs with chromosomes of variable size, enabling the development of new kinds of bloat-control mechanisms that can naturally benefit from the underlying parallel processors that are present in modern computer systems, including handheld devices, desktop systems, and large computer systems.

Based on this space-time connection, we have devised new ways of naturally controlling the bloat problem: we analyze the bloat phenomenon using *time* (execution time) instead of *space* (memory consumption), although both are measures of a single entity: the time-space behavior of individuals. We resort to the space-time continuum as a metaphor providing a new source of inspiration and thus try to provide a new way of analyzing and fighting bloat. To the best of our knowledge, this is the first conceptualization of such an approach (description and application), and the results we show demonstrate the usefulness of the idea, both in sequential and embarrassingly parallel versions of the algorithm, which opens up new possibilities in other parallel and distributed versions of GP.

B. RESEARCH CONTRIBUTIONS

The main contributions of this work are (1) the idea of properly establishing a relationship between an individual's size and evaluation time, showing the advantages of using evaluation time when analyzing individuals' complexity. Also, (2) we show that the approach leads to a new kind of straightforward bloat control mechanism, and (3) we present the first such implementation based on fitness evaluation time, tested in both sequential and parallel executions of the algorithm. Although many possibilities arise from the new perspective, this first method shows the usefulness of the approach, paving the way towards a new class of bloat control mechanisms.

We organize the paper as follows: In Section II we describe some GP-associated considerations that gave rise to the ideas developed later. Section III reviews previous approaches for fighting bloat, and the main features of parallel models that may be taken into account when analyzing this problem. Then, Section IV describes how load-balancing techniques could be of interest in this context. Section V presents our proposal, while Section VI shows the experiments performed and the results obtained. Finally, we draw our conclusions in Section VIII.

II. THE COMPUTATIONAL COMPLEXITY OF GP IN TIME AND SPACE

Before describing the idea that motivates our work, we need to define some ideas in a proper context so we can better convey the key concepts. GP is said to be a machine learning model that explores the search space of possible programs that solve a given problem, [1]. Usually, in computer science, problems are classified according to a complexity class, which is associated with an algorithm that attempts to solve the problem. Programmers implement the proposed algorithm through a language that is the formal realization of a problem. When we want to reason theoretically about a problem, we often examine the corresponding language. In particular, the technique of GP builds computer programs, algorithms, automatically written in a domain-specific language suitable to the studied problem. Therefore, the system automatically applies an encoding w of an input y to the problem X, and the answer is the solution (program) to (attempt to) solve that problem. An algorithm is a step-by-step way to solve a problem. GP creates many different candidate algorithms for solving a particular problem through a generate-and-test strategy that builds a formal analog of an algorithm–a machine working with a given language.

Computational complexity refers to the theory that formalizes the difficulty of achieving a solution to a problem. Whatever the algorithm used, the programmer describes the solution in terms of how many resources the best algorithm requires to address the problem. Indeed, the running time may, in general, depend on the instance. In particular, large instances typically require more time to solve. Thus, programmers calculate the time required to solve problems (or the space required, or any other measure of complexity) as a function of the size of the instance. This is usually taken to be a function on the size of the input in bits. Complexity theory is interested in how algorithms scale with an increase in the input size. The evolutionary computation community has adopted this way of seeing computational complexity in genetic programming: explicitly, the community bases the study of bloat and the proposals to control it on regulating the size of the individuals, [3], [9]–[11]. In other words, programmers express the time taken as a function of size, and the idea is to create mechanisms that manage the size of individuals to reduce bloat.

As we have explained, the reasoning for measuring bloat through the size of individuals has historical and mathematical foundations. We offer here a fresh perspective to study bloat through running time in the machine. Time is an abstract concept that allows humans to organize temporal events mentally. This view offers new ways of looking at bloat. For our purposes, time is associated with the resources, functions, and terminals that are necessary to complete the task/program. Before describing the details, we review first

the bloat-associated literature, to better understand the novelty of the approach.

III. THE BLOAT PHENOMENON

Even in his seminal publication [1], Koza already provided some hints on how to fight this undesirable phenomenon that frequently makes the algorithm run out of memory or consume so much computing time that the discovery of useful solutions is hampered. Koza proposed that the employment of size-related penalty values be combined with standard fitness functions, and that a maximum depth limit be established so that trees cannot continue growing without restrictions. Programmers established depth and size limits as *de-facto* techniques and combined them with any other new proposal.

The bloat problem has been addressed frequently in the GP literature since then. Although we do not provide an exhaustive review of the topic (some reviews of the literature are available for interested readers, such as [10], [12]), we refer to several studies analyzing the reasons for bloating, like [2], hitchhiking exposed in [13], defense against crossover by [14], removal bias explained in [9], fitness as the primary source for bloat introduced in [5], depth correlation theory by [15], crossover bias detailed in [16], an so on.

Among the first techniques proposed, as described above, when referring to [1], penalties associated with fitness functions were the first countermeasures. Soon researchers devised more sophisticated approaches, such as establishing dynamic depth or size limits for individuals [12], but also variable population sizes were indirectly applied to globally control the total number of nodes to be managed ([17], [18]). Also, multiobjective approaches, where two objectives are simultaneously considered, such as (i) fitness value and (ii) individual's size, have been applied to control bloat, [19].

Other techniques are available that take into account individual sizes and shapes ([20], [21]). However, we want to focus on an alternative method called the waiting room approach, introduced by [22]. The idea is for individuals to add a pre-birth phase to all newly created individuals. Children must wait for some time proportional to their size before they are allowed to enter the population and compete. Although the authors conceded that the idea was associated with the relationship between individual sizes and evaluation times, they maintained the emphasis on the size-control mechanism and hence did not elaborate on the time concept, nor did they take into account the possibilities associated with parallel and distributed infrastructures available, given their influence on evaluation time when individuals migrate to available processors. Thus, they relied on the total number of nodes individuals feature, similarly to all other methods, although using a somewhat different approach.

We must also mention some work that took initial inspiration from *operator equalization* presented in [23], aimed at controlling the distribution of program sizes at each generation, defining a specific shape for the distribution. Some of the best results were achieved by using a uniform or even distribution [24], and also by applying speciation, fitness sharing or elitism, see [4]. However, again, difficulties arise related to effectively applying the method–for example, in how to control the distribution shape without changing the search problem? Also, how should the method efficiently account for individuals' sizes and shapes?

As described before, many size-related approaches to bloat control in GP have already been proposed and applied. Here we explore a deeper analysis of computing times that sheds light on the problem, in contrast to the standard approach of directly controlling individuals' sizes. In particular, we explore the relationship between size and evaluation time, not only in the standard (sequential) approach, but also when using parallel and distributed systems. The aim is not only to save computing time but also to address the bloat phenomenon more naturally.

A. PARALLEL MODELS AND THE BLOAT PHENOMENON

Some authors have already considered the intrinsic features of parallel models that they use when hard problems are faced. Notably, the island model, one of the best known parallel models, has already been studied from the perspective of the bloat phenomenon, as we review below.

When GP deals with real-world problems, researchers have quickly noticed that parallel and distributed systems are frequently the only feasible approach for finding solutions in a reasonable time. Several systems rely on spatial structures [25], such as the island [26] and cellular models [27], while others resort to the embarrassingly parallel one. The advantages of every model are well known, and two sources of improvement have been described: on the one hand, the speed gained from the parallel architecture running the algorithm, and on the other hand, the spatial distribution of individuals [25]. In other words, breeders are not well mixed, which helps to promote diversity [28], thus allowing researchers to find better solutions in fewer evaluations.

This observation is applicable in general to EAs [29] but also to GP [25]. Moreover, the employment of variable-size chromosomes in the latter introduces some differences that have already been analyzed: in particular, when the model is connected with the bloat phenomenon, an interesting relationship between spatial structure and the bloat phenomenon has been found.

In a series of papers, a method based on the island model offers some possibilities for fighting bloat ([6], [30], [31]), and this methodology has led to a new proposal by [7] considering the spatial distribution of islands in GP. The connection between the dynamics of some parallel models for GP and the bloat phenomenon, as explored there, proved to be mainly due to spatial structure, which relies on islands of individuals. Nevertheless, there is still a second source of possible improvement in parallel EAs, as we described above: the number of computing resources employed to run the algorithm, which has not been studied yet from its influence on the algorithm's bloating behavior. Even when we select the simplest embarrassingly parallel model for running a GP experiment, a load balancing technique must decide

how to distribute individuals among the available computing resources, and this may also influence the bloat phenomenon, as we show below.

IV. LOAD-BALANCING AND PARALLEL GP

Among the above-mentioned parallel models, the only one that does not change an algorithm's behavior is the embarrassingly parallel model. All of the algorithm's steps are performed as in the sequential version, and the only change is introduced in the most expensive part of the algorithm: the fitness evaluation step. Thus, instead of each individual being sequentially evaluated in the population, they are distributed among the available processors, which compute fitness values in parallel.

Although a simple approach, this is probably the most widely employed model, usually allowing researchers to use large population sizes. Given that a large number of individuals interact across the available processors, which are typically smaller in number than the population size, some load-balancing mechanism must be applied. This mechanism is in charge of sending individuals to idle processors, and might also provide new hidden properties: sometimes, a more in-depth analysis of the new version of the algorithm allows us to discover some properties unnoticed before. We are here interested in both the parallel model itself and the load-balancing technique that can be used and considered as the basis for a new proposal that we describe and analyze below.

A. LOAD-BALANCING TECHNIQUES

Scheduling and load-balancing are very active areas of research in parallel and distributed computing. They aim to distribute tasks among multiple computing resources accurately while reducing makespan. Scheduling algorithms, also called load-balancing methods, aim to determine how to distribute workloads most effectively. It can be done statically, so that assignments never change for given tasks, or dynamically, so the load-balancing method can migrate tasks when processors become idle while others are busy. Frequently, information about task size is not available at execution time, so decisions to be taken by the load-balancing algorithm are not easy. As we explain next, this may be the case in genetic programming, in which individual sizes and complexities of programs evolved are variable. Interested readers can find a taxonomy of load-balancing methods in [32], while [33] presents a comparison of different strategies.

Load-balancing has also been a topic of interest for researchers in parallel EAs. Moreover, researchers have frequently employed EAs as a tool for finding better load-balancing methods and scheduling policies in parallel and distributed systems (see, for instance, [34]–[37]). Although GAs are more prevalent in the area, other members of the family, such as evolution strategies, have also been applied [38]; similarly, static methods predominate, but dynamic load-balancing has also been studied from the EA standpoint [39]. Here we are not interested in how to improve a given load-balancing mechanism for application to general problem-solving in parallel and distributed systems. Still, we are interested in the role that load-balancing techniques play within parallel-EAs, and more specifically, in parallel-GP when addressing the bloat problem.

Researchers have considered load balancing techniques as an implicit component of parallel versions of genetic programming. Since the nineties, static load-balancing mechanisms–the ones we consider here–have been applied within parallel versions of GP, when facing hard, real-world problems. For instance, in [40], the authors describe a parallel version of GP that considers the complexity of individuals as the basis for establishing the load-balancing policy. Given the function set employed there, made up of arithmetic operations, their method orders the population according to individual sizes, and then enters a loop that looks for idle processors, which receive an individual from the list until the whole list is completed. Although those authors were aware of the importance of load-balancing techniques, they did not call out the relationship that may exist with the evolution of individuals' size, which is what we aim to exploit here.

Few papers since then have studied the importance of load-balancing techniques in GP. We may refer to [41], where authors tested several methods. Nonetheless, the authors did not present a specific study on their work's relationship with the bloat phenomenon. In any case, we describe some of the main features that are involved in load-balancing approaches for GP.

B. THE STRUCTURAL COMPLEXITY OF GP INDIVIDUALS

When any load-balancing technique is to be employed, a prediction of computing time for the task must be applied, so that the method can properly decide when to launch the task. In GP, an essential feature of GP individuals is their structural complexity [42]. This value is typically computed with the number of nodes, as in the case of evaluating either *computing effort* [43] or lexicographic complexity [44]. Both are approximate estimates of the real value required–in other words, the evaluation time of the individuals' fitness function.

Nevertheless, we can adopt a different point of view, as we do in the approach we present below: given that we estimate the measurement of the real complexity of an individual through the individual evaluation, we can partially characterize individuals using that computing time, so that we can employ it in future decisions. Although that value is not available when the load-balancing mechanism must decide when to launch the evaluation of a new individual; nevertheless, it is available after the individual's evaluation. This aspect could be useful, if not for that individual, given that it was already sent to be evaluated, then at least for its children, as a value to use somehow in approximating their evaluation times.

Our approach thus takes into account an individual's computing time, as a value to decide how to distribute children among available computing resources, and ultimately, to reduce computing time while simultaneously reducing the bloat phenomenon. We thus need to keep a record of the

time that each program (individual) requires during testing and use that information to create clusters of programs with similar durations that are useful for load-balancing of individuals. We use the computer's clock to give a value to the runtime of a program; this, of course, is correlated with the size of the computer program and the number of instruction cycles required to execute it. We create all clusters without regard to the fitness function. We do not measure the size of the individuals directly, nor use any information about the complexity of breeding programs other than time. We explain below all these aspects and show that we can address the bloat problem through this simple strategy without increasing the computational complexity of the algorithm.

V. METHODOLOGY

This section covers the description of the new approach we follow when fighting bloat: First, we describe the bloat control mechanism; Second, we explain the implementation of the mechanism, which runs using an available tool, [45]. Finally, we describe the methodology that has been used to perform the experiments and obtain the results.

A. THE ALGORITHM'S KEY CHANGE

This method does not use a traditional tree size or depth measure to estimate the computational complexity of an individual. Instead, it considers execution time as a measure of an individual's complexity. Although the system may never use some nodes of a tree, or it may traverse some nodes several times when loops are allowed within the program trees, there is still a correlation between program size and running time that can be exploited to prevent bloat in GP, and this is the crucial idea developed below–the connection between time and space as both sides of a single concept.

This idea can be easily applied at the times when individuals are evaluated: we have to take elapsed time during an individual's evaluation as the complexity value required. Moreover, it is reasonable to think that this method better describes an individual's complexity since it depends not only on the number of operations performed–the number of nodes traversed–but also on the nodes' complexities. Therefore, we measure time explicitly by capturing the starting and finishing time of the evaluation and calculating the difference. This idea is particularly useful when multicore or manycore computer architectures are to be employed: ideally, all of the individuals in the population could be evaluated simultaneously, and therefore, their evaluation times obtained also simultaneously.

Even though the idea of using evaluation time looks interesting, measuring this time precisely can be tricky. Ideally, evaluation time should represent the actual amount of time the task of evaluating the individual has been running on the processor–the amount of time that is also known as ''wall clock time''. Depending on the parallel architecture, the elapsed evaluation time could be influenced by any I/O operation, OS-related task or other processes not related to the evaluation–an undesired effect. This step is of particular

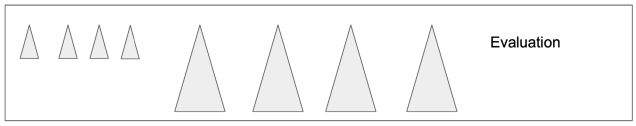


FIGURE 1. First step: The system sends individuals to different threads.

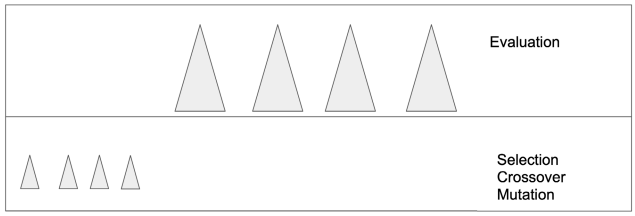


FIGURE 2. Second step: The fitness of individuals with fastest execution time is available first.

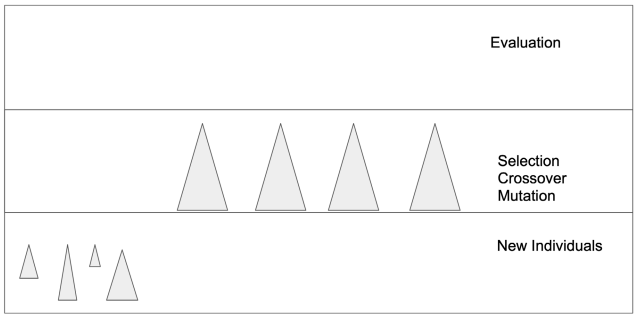


FIGURE 3. Third step: Fastest individuals produce children while slower ones are coming back from the evaluation.

importance in a multi-threaded process where threads can be influenced by each other. Nevertheless, we have considered that such circumstances, on average, equally influence all evaluations. If so, and taking into account that we use evaluation time only for comparison among individuals, we simplify the measurements by directly using the elapsed evaluation time as the representation of the individual's complexity.

Once the individuals' evaluation times have been obtained, and with the hypothesis that individuals of similar sizes tend to produce offspring of similar sizes, our proposed method groups individuals by computing time, always understanding it as an indirect–and more natural to compute–measure of an individual's size. We must again consider that in a parallel system with as many processors as individuals, individuals of similar sizes finish their evaluations nearly simultaneously and are ready to reproduce. Therefore, an automatic grouping mechanism naturally arises from these parallel architectures, see figures 1-4. If the number of processors in the system is small, then the load balancing mechanism, which is always in charge of distributing tasks among processors, could decide to group certain individuals together into single tasks, and may thus group them according to the ending time of individuals' evaluations.

After grouping, we perform selection and breeding phases within each group, so only individuals of similar size-time are

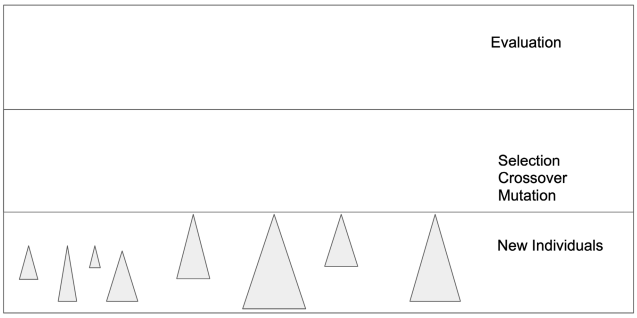


FIGURE 4. Fourth step: Slower individuals produce children.

allowed to crossover. We run grouping after all individuals have been evaluated, creating groups of the same cardinality by evenly dividing the whole population.

Our hypothesis states that individuals of similar size produce offspring of similar size. This point might not be the case if the crossover operation does not divide the individual into two similar-size parts. Indeed, crossover often produces small and large individuals whose sizes do not follow our hypothesis. Nevertheless, if we consider a random division of individuals, we expect a central tendency which results in a size between those both parents. Some readers may see this aspect as a weak point of our offspring-size prediction, which could be improved in a future version. However, generally speaking, we expect that offspring have program sizes that range similarly in comparison with their runtimes, as we show in the experiments.

B. IMPLEMENTATION

We took inspiration from the above-described mechanism by which parallel computer systems run population-based algorithms. Therefore, when designing a specific bloat control mechanism for GP that uses the new time-based perspective, the parallel version of the algorithms, in particular, multi-thread-based models, has been initially chosen, which are available today in some of the most popular EA tools. Thus, one thread executes all operations performed within each group. Hence, the number of threads created corresponds to the number of groups. Each thread collects its corresponding individuals and performs the selection and breeding steps. In this way, each group isolates all returning operations following a straightforward process based on evaluation time.

After the breeding phase, the mechanism takes advantage of the independence among groups of individuals contained in different threads, and it evaluates all corresponding individuals. Each group/thread contains the same number of individuals, individuals with similar execution times. As a result, the evolutionary process is considerably speeded up by parallelizing of the evaluation phase. Afterward, once all individuals of the population finish the evaluation stage, obtaining their computing times, they are sent to the corresponding thread using their computing time (size surrogate) value, so that the next breeding operations can continue.

There are many ways of implementing such a method, particularly when considering the parallel execution of the algorithm and the load balancing mechanism. However, the key idea is the change of perspective, from individuals' size to its computing time, although, indeed, the best way of proving the usefulness of the idea is to develop a first method based on it. Therefore we present a first implementation and then test whether it achieves the proposed goal: to prevent the bloat phenomenon by applying an alternative to the size-based complexity estimation measure. We believe that this approach naturally adapts to parallel environments, and it provides for future and improved developments, although the essential idea can also be demonstrated in a sequential version of the algorithm, as we study below.

1) SOFTWARE TOOL AND FURTHER DETAILS

Next, we describe a way to make the bloat method easily usable; it has been implemented based on a popular existing tool, [45]. We built such a system in modules to facilitate the replacement of any part involved in the evolutionary process. In our case, we replace the module that carries out the breeding phase with the new time-based approach. Our bloat control mechanism slightly modifies the breeding mechanism. We implement two new operations to apply the bloat control mechanism.

- *GroupBreeder* orchestrates the breeding phase and starts the corresponding threads.
  - As the first step, the system groups individuals according to evaluation times. During their evaluations, we record elapsed time, so each individual already contains it as a new feature. Before grouping them, we sort all individuals of the population by evaluation time.
  - Then, the same number of individuals goes to each group, taking care of ranking before making the groups. In case individuals cannot be equally split into groups, the initial groups are filled with additional individuals.
  - Next, each group instantiates one thread, to which it assigns all individuals in the group.
  - The program starts threads and continues until all threads have finished.
- *GroupBreederThread* represents the threads that perform the selection, breeding, and evaluation of individuals.
  - Next, the program performs a call to the module that runs selection and breeding, specifying the group to which these operations need to be applied. Here, the only change to the original implementation is to apply these operations only to individuals that correspond to the specified group–the group that corresponds to the thread.
  - Once selection and breeding phases have finished, evaluation of new individuals generated by this thread takes place.

Note that we have not adjusted the evaluation step. Therefore, the timings of the computations gone through by all individuals are in memory.

C. EXPERIMENTS

A set of experiments was run to observe how the bloat control mechanism affects size growth and fitness across generations of individuals. This section describes the experiments and results obtained.

All experiments described below run on an Intel(R) Xeon(R) CPU (E5530) that offers 8 cores at 2.4 GHz and 8 GB of memory. We use default configuration parameters as setup in ECJ for each of the pre-done GP benchmark problem domains. Generations set to 50 in almost every problem, and 30 runs of each of the experiments were launched, for statistical purposes. As described above, benchmark problems from the GP literature were used with the necessary configurations already available in the ECJ toolkit: parity, ant, lawnmower, multiplexer, and regression. Regarding the proposed group-based mechanism, for which we must specify the number of groups to which individuals are distributed in every generation, we have run several experiments: 1 group, which corresponds with the standard GP algorithm, and also 2, 4, 8, 16, 32, 64, and 128 groups.

VI. RESULTS

In this section, we discuss and show the results extracted from the experiments. For each of the studied problems, we plot average fitness and size. The proposed method uses parallel execution in all results presented here. However, we also show the results of a sequential approach in this section. Finally, plots are shown to depict the computational effort used during the evolutionary process against the fitness obtained along with generations for all groups.

A. ANT

The artificial ant problem is a more sophisticated yet classical GP problem, in which the evolved individuals have to control an artificial ant so that it can eat all the food located in a given environment. For the ant problem, we observe the average fitness of individuals in Figure 5. Each line represents executions for runs with different numbers of groups. As can be seen, fitness is slightly degraded when the number of groups increases. In any case, the observed trend is more deeply examined below through a statistical test, trying to see if differences are statistically significant.

While looking at the average size of individuals in Figure 6, we observe how all experiments employing more than one group provide better results than the standard one-group approach. At generation 50, 4 groups produced the best average fitnesses, with much smaller sizes than with one group.

B. LAWNMOWER

The lawnmower is a classical and basic GP problem. The goal is to find a program for a lawn mower that directs

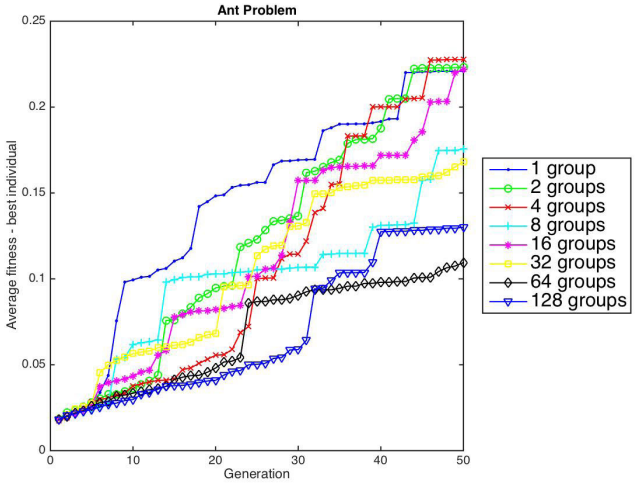


FIGURE 5. Best-fitness evolution along generations (averaged over 30 runs) for the ant problem (maximizing fitness).

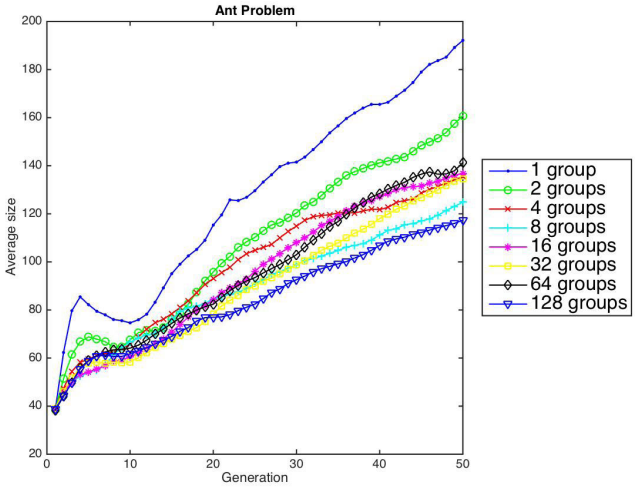


FIGURE 6. Size-evolution along generations (averaged over 30 runs) for the ant problem.

the mower over the whole lawn, so it is similar to the ant problem. For the lawnmower problem, we depict the average fitness in Figure 7. Here, the number of groups affects fitness gradually. In any case, differences in fitness quality are small.

Nevertheless, we observe the opposite in Figure 8, where we study size. Size is dramatically reduced in this case, reaching less than half of the size of a typical run (one group). Considering the associated fitnesses, it seems a good trade-off.

C. MULTIPLEXER

The multiplexer problem is another extensively used GP problem. Basically, it trains a program to reproduce the behavior of an electronic multiplexer. Usually, a 3-8 multiplexer is used (3 address entries, from A_0 to A_2, and 8 data entries, from D_0 to D_7), but virtually any size of multiplexer can be used. In the case of the multiplexer problem, we follow the parameters given in the example in ECJ, and we plot

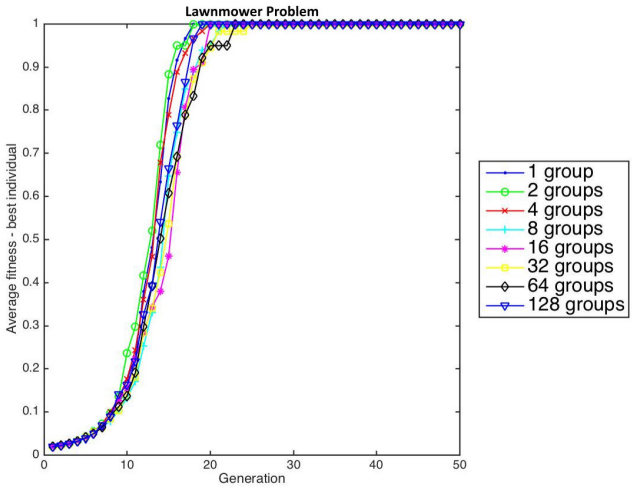


FIGURE 7. Best-fitness evolution along generations (averaged over 30 runs) for the lawnmower problem (maximizing fitness).

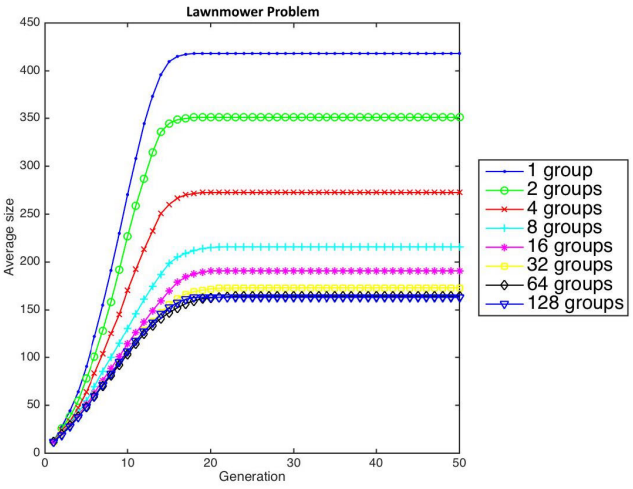


FIGURE 8. Size-evolution along generations (averaged over 30 runs) for the lawnmower problem.

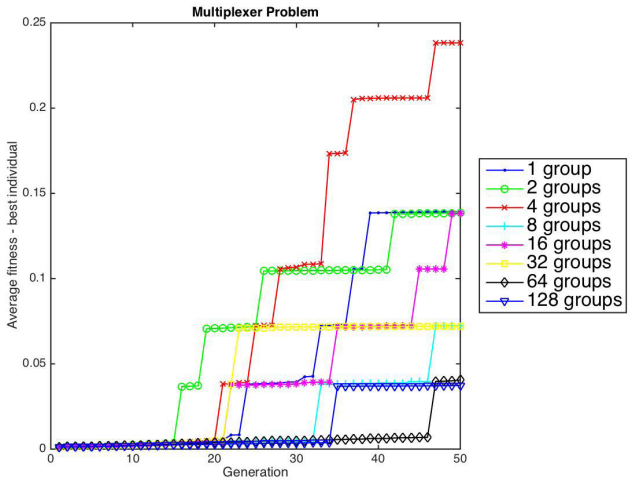


FIGURE 9. Best-fitness evolution along generations (averaged over 30 runs) for the multiplexer problem (maximizing fitness).

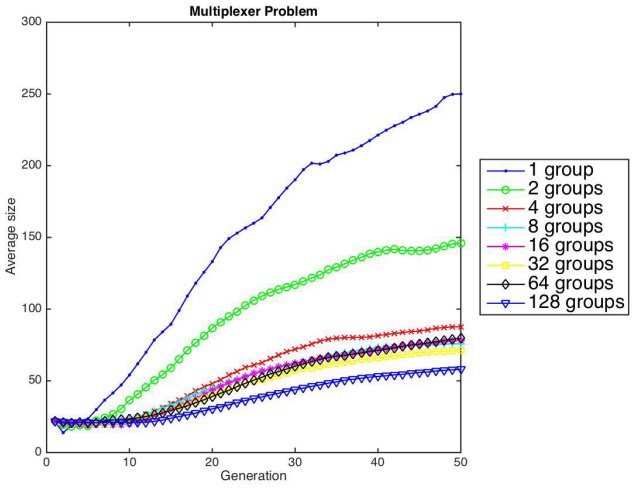


FIGURE 10. Size-evolution along generations (averaged over 30 runs) for the multiplexer problem.

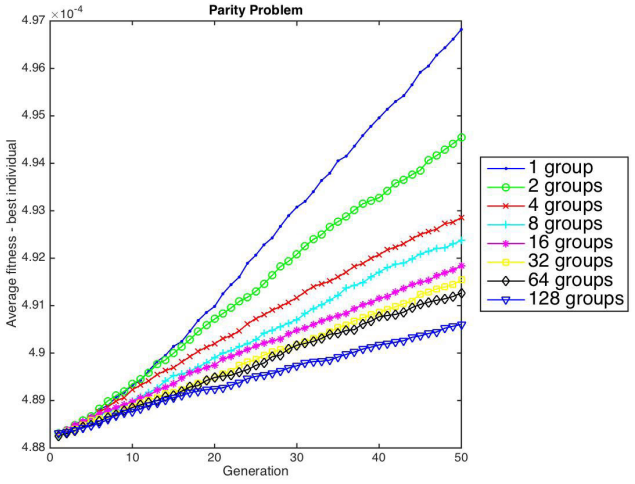


FIGURE 11. Best-fitness evolution along generations (averaged over 30 runs) for the parity problem (maximizing fitness).

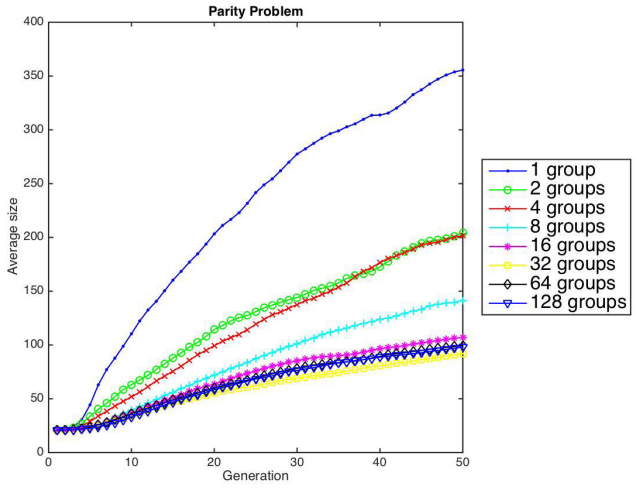


FIGURE 12. Size-evolution along generations (averaged over 30 runs) for the parity problem.

the average fitness in Figure 9. Here, it is challenging to observe clear tendencies, although the 4-group runs obtained good fitness while on the other hand, fitness quality for the 128-group runs was poorer. Again the 4-group runs line is

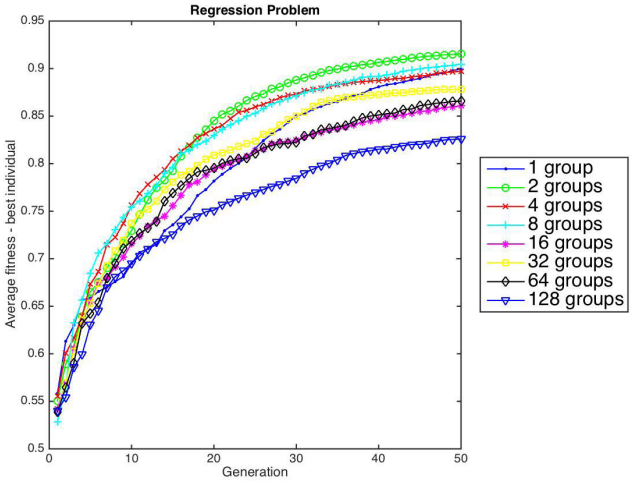


FIGURE 13. Best-fitness evolution along generations (averaged over 30 runs) for the regression problem (maximizing fitness).

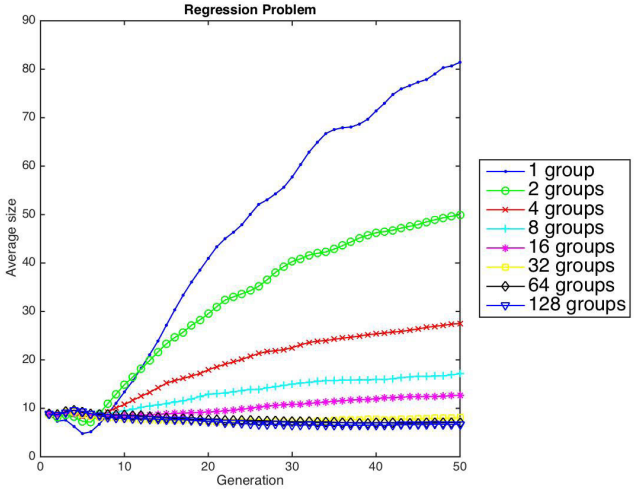


FIGURE 14. Size-evolution along generations (averaged over 30 runs) for the regression problem.

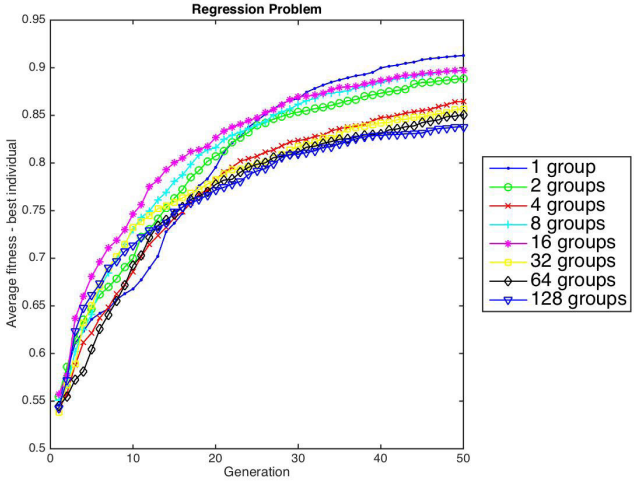


FIGURE 15. Best-fitness evolution along generations (averaged over 30 runs) for the regression problem (sequential execution) (maximizing fitness).

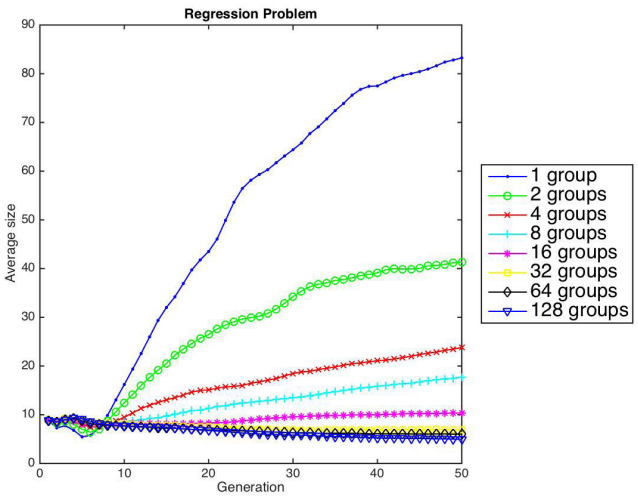


FIGURE 16. Size-evolution along generations (averaged over 30 runs) for the regression problem (sequential execution).

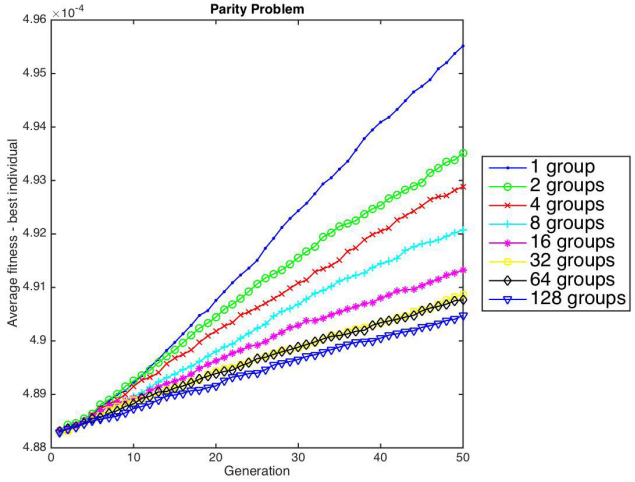


FIGURE 17. Best-fitness evolution along generations (averaged over 30 runs) for the parity problem (sequential execution) (maximizing fitness).

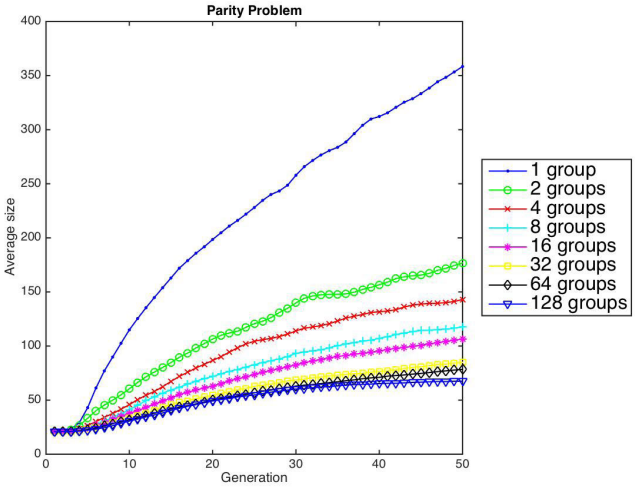


FIGURE 18. Size-evolution along generations (averaged over 30 runs) for the parity problem (sequential execution).

the winner in fitness with a reduced size in comparison with one-group runs.

In the case of the size represented in Figure 10, we can see a dramatic improvement in size for 2-group and 4-group runs.

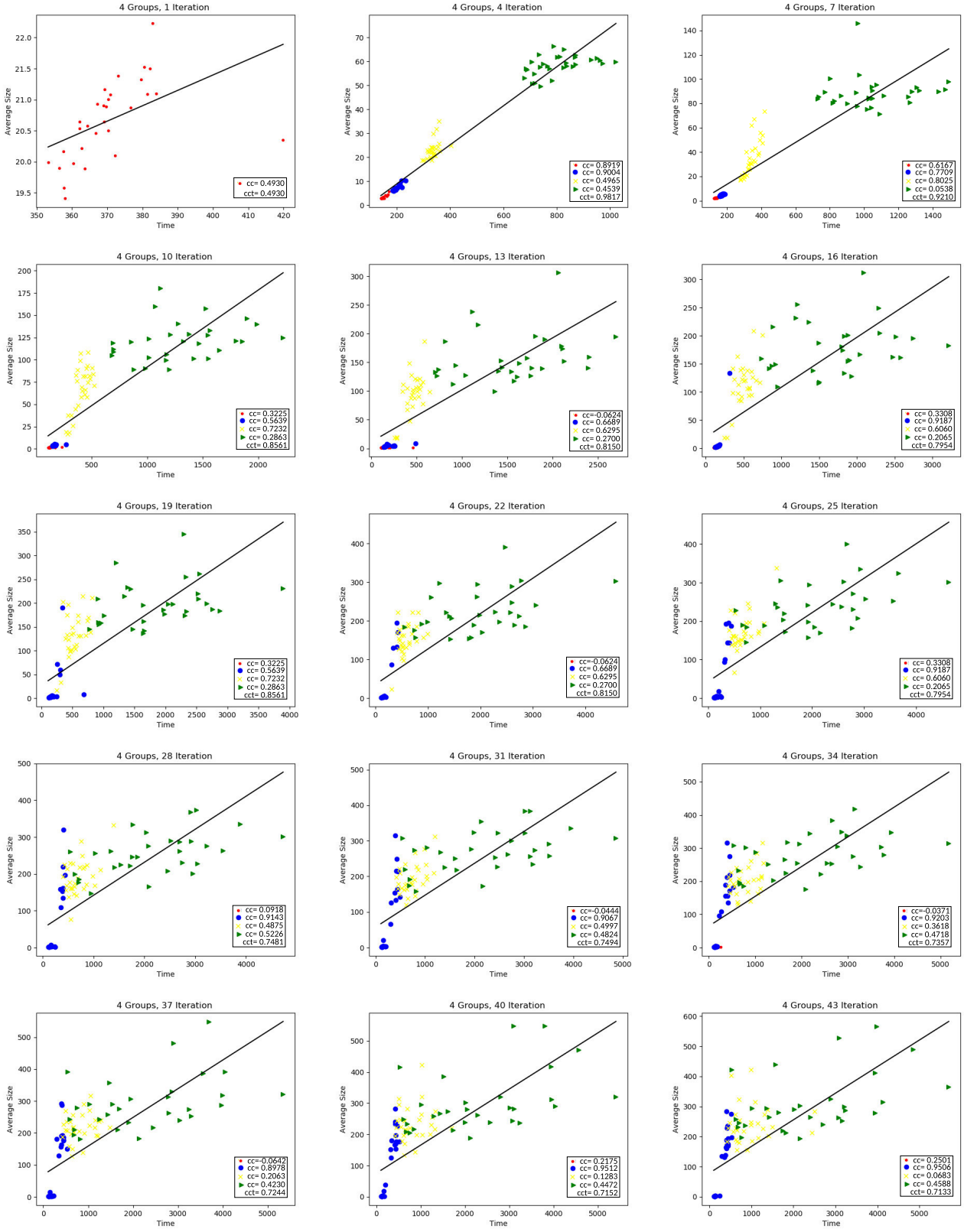


FIGURE 19. These plots show the average size of 30 jobs run for 50 generations of the parity problem against the corresponding times for the case of 4 groups. We can observe that for a given time the individuals in each group have different sizes. Note how the groups (blue and yellow) align along the vertical axis and even the sparse green group presents elements aligned with the same trend.

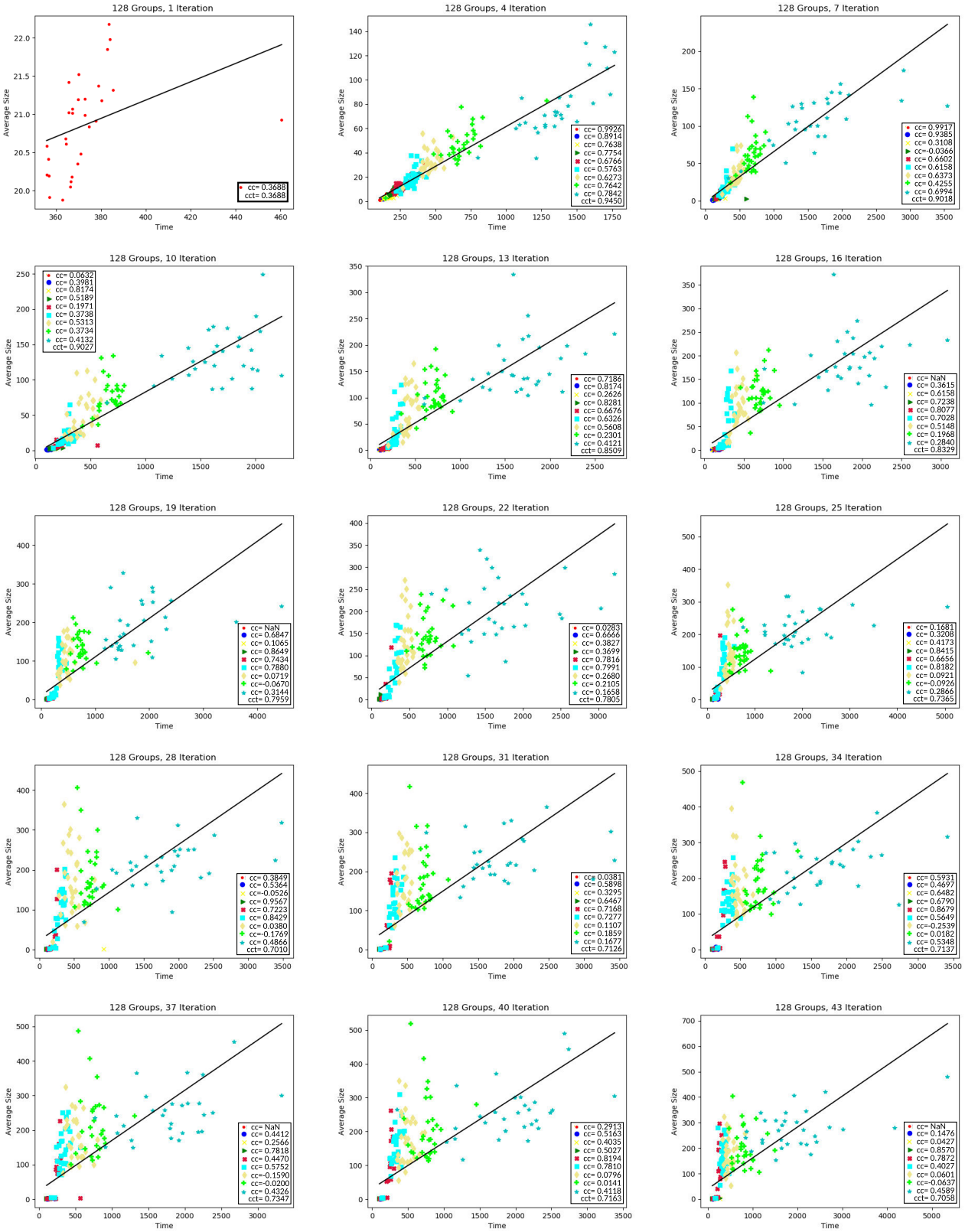


FIGURE 20. These plots show the average size of 30 jobs run for 50 generations of the parity problem against the corresponding time for the case of 128 groups. We observe that for a given time the same pattern of different individual sizes arises in the experiments. For visualization purposes, the 128 groups are divided into 8 intervals with a step size of 16, so we plot only 9 groups.

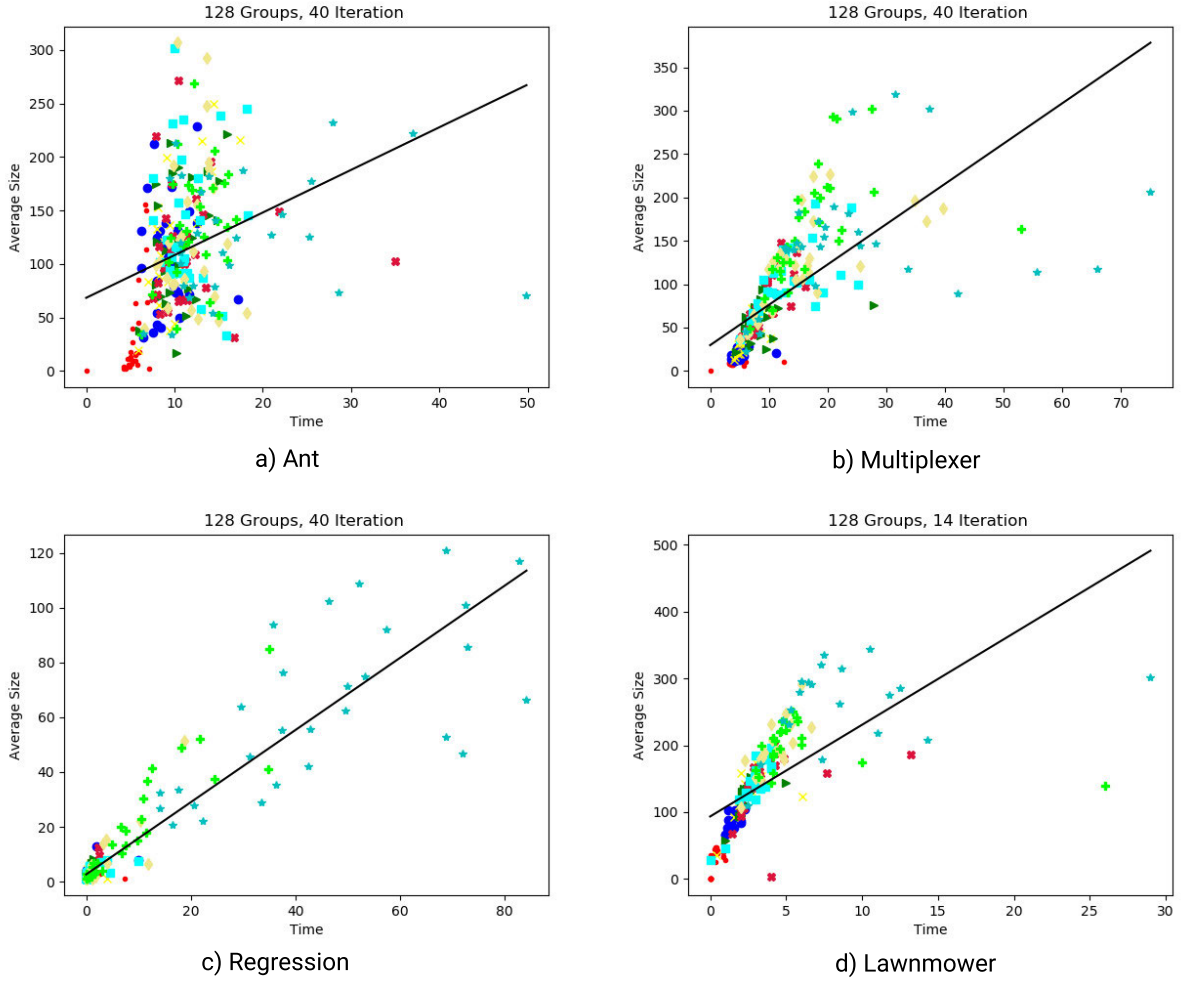


FIGURE 21. These plots show the average size of 30 jobs run for 50 generations on the ant, multiplexer, regression, and lawnmower problems. We observe that for a given time, similar patterns to those seen in the Parity problem for different individual sizes arise in the four experiments.

TABLE 1. Ant experiment. Best-fitness statistical results.

1 Group Vs. ...	Kolmogorov Smirnov	Levene	Test used	p_value	H_0
2 Groups	0.0	0.0	ANOVA	0.979	No Reject
4 Groups	0.0	0.0	ANOVA	0.942	No Reject
8 Groups	0.0	0.0	ANOVA	0.612	No Reject
16 Groups	0.0	0.0	ANOVA	0.990	No Reject
32 Groups	0.0	0.0	ANOVA	0.532	No Reject
64 Groups	0.0	1.0	WELCH	0.000	Reject
128 Groups	0.0	1.0	WELCH	0.000	Reject

TABLE 2. Ant experiment. Size-evolution statistical results.

1 Group Vs. ...	Kolmogorov Smirnov	Levene	Test used	p_value	H_0
2 Groups	0.0	0.0	ANOVA	0.095	Reject
4 Groups	1.0	1.0	KRUSKAL-WALLIS	0.002	Reject
8 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
16 Groups	1.0	1.0	KRUSKAL-WALLIS	0.002	Reject
32 Groups	1.0	1.0	KRUSKAL-WALLIS	0.001	Reject
64 Groups	1.0	1.0	KRUSKAL-WALLIS	0.006	Reject
128 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject

TABLE 3. Lawnmower experiment. Best-fitness statistical results.

1 Group Vs. ...	Kolmogorov Smirnov	Levene	Test used	p_value	H_0
2 Groups	0.0	1.0	WELCH	1.0	No Reject
4 Groups	0.0	1.0	WELCH	1.0	No Reject
8 Groups	0.0	1.0	WELCH	1.0	No Reject
16 Groups	0.0	1.0	WELCH	1.0	No Reject
32 Groups	0.0	1.0	WELCH	1.0	No Reject
64 Groups	0.0	1.0	WELCH	1.0	No Reject
128 Groups	0.0	1.0	WELCH	1.0	No Reject

However, size remains similar for 8-, 16-, 32-, and 64-group runs, while for 128 groups the size is again slightly decreased.

D. PARITY

Parity is one of the classical GP problems. The goal is to find a program that produces the value of the Boolean even parity function given n independent Boolean inputs. Usually, 6 Boolean inputs are used (Parity-6), and the goal is to match the good parity bit value for each of the $2^6 = 64$ possible entries. The problem can be made harder by increasing the number of inputs. In the parity problem, fitness was monotonically affected by the number of groups, as can be observed in Figure 11. Again, taking into account the scale, the effect was quite small.

TABLE 4. Lawnmower experiment. Size-evolution statistical results.

1 Group Vs. ...	Kolmogorov Smirnov	Levene	Test used	p_value	H_0
2 Groups	1.0	0.0	KRUSKAL-WALLIS	0.000	Reject
4 Groups	1.0	0.0	KRUSKAL-WALLIS	0.000	Reject
8 Groups	1.0	0.0	KRUSKAL-WALLIS	0.000	Reject
16 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
32 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
64 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
128 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject

TABLE 5. Multiplexer experiment. Best-fitness statistical results.

1 Group Vs. ...	Kolmogorov Smirnov	Levene	Test used	p_value	H_0
2 Groups	0.0	0.0	ANOVA	0.994	No Reject
4 Groups	0.0	0.0	ANOVA	0.326	No Reject
8 Groups	0.0	0.0	ANOVA	0.393	No Reject
16 Groups	0.0	0.0	ANOVA	0.990	No Reject
32 Groups	0.0	0.0	ANOVA	0.391	No Reject
64 Groups	0.0	1.0	WELCH	0.000	Reject
128 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject

TABLE 6. Multiplexer experiment. Size-evolution statistical results.

1 Group Vs. ...	Kolmogorov Smirnov	Levene	Test used	p_value	H_0
2 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
4 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
8 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
16 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
32 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
64 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
128 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject

TABLE 7. Parity experiment. Best-fitness statistical results.

1 Group Vs. ...	Kolmogorov Smirnov	Levene	Test used	p_value	H_0
2 Groups	1.0	0.0	KRUSKAL-WALLIS	0.000	Reject
4 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
8 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
16 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
32 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
64 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
128 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject

In this problem, we obtain a dramatic reduction up to a third of the size as compared with a regular run (1 group), see Figure 12. The slightly affected fitness may be acceptable, taking into account the considerable reduction in size.

E. REGRESSION

Symbolic regression is one of the best known problems in GP. It is commonly used as a tuning problem for new algorithms, but is also widely used with real-life distributions, where other regression methods may not work. It is conceptually a simple problem, and therefore makes a good introductory example. In the regression problem, fitness is slightly affected when using 2, 4, and 8 groups. When using 16 or more groups, fitness deteriorates, as can be seen in Figure 13.

Regarding size, as observed in Figure 14, a very notable reduction in size comes as a result of a higher number of groups. Clearly and monotonically affected by the number of groups, size moves gradually from 80 for a regular run (1 group), to a size of 6 with 128 groups. This may provide attractive tradeoffs, particularly as regards overfitting.

F. SEQUENTIAL EXECUTION

We extracted all results presented so far from runs in which we applied parallel execution. As described before, parallel models naturally allow individuals to be grouped according to running time, when a number of them are simultaneously launched and evaluated in different processors. However, the proposed bloat control method can also be carried out

TABLE 8. Parity experiment. Size-evolution statistical results.

1 Group Vs. ...	Kolmogorov Smirnov	Levene	Test used	p_value	H_0
2 Groups	1.0	0.0	KRUSKAL-WALLIS	0.000	Reject
4 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
8 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
16 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
32 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
64 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
128 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject

TABLE 9. Regression experiment. Best-fitness statistical results.

1 Group Vs. ...	Kolmogorov Smirnov	Levene	Test used	p_value	H_0
2 Groups	0.0	1.0	WELCH	0.000	Reject
4 Groups	0.0	0.0	ANOVA	0.919	No Reject
8 Groups	0.0	1.0	WELCH	0.000	Reject
16 Groups	1.0	0.0	KRUSKAL-WALLIS	0.012	Reject
32 Groups	1.0	1.0	KRUSKAL-WALLIS	0.010	Reject
64 Groups	1.0	0.0	KRUSKAL-WALLIS	0.027	Reject
128 Groups	1.0	0.0	KRUSKAL-WALLIS	0.001	Reject

TABLE 10. Regression experiment. Size-evolution statistical results.

1 Group Vs. ...	Kolmogorov Smirnov	Levene	Test used	p_value	H_0
2 Groups	1.0	0.0	KRUSKAL-WALLIS	0.000	Reject
4 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
8 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
16 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
32 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
64 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject
128 Groups	1.0	1.0	KRUSKAL-WALLIS	0.000	Reject

without parallelization, executing it sequentially, with minimal changes–allowing individuals to be grouped according to running time–applied to the main algorithm. We must bear in mind that what we do in the sequential version is to emulate the behavior observed in the parallel model. Thus, no population structures or subpopulations are considered here, although some resemblance with structured models occurs, and probably many other possibilities may be applied to benefit from the idea studied above for parallel systems. Nevertheless, this most straightforward implementation analyzed here allows us to see how the idea works in sequential models.

Next, we present results from sequentially executed runs for some of the previous problems–in particular, for regression and parity. In the case of the regression problem, similar to what is seen in the results from parallel execution (Figures 13 and 14), the fitness (depicted in Figure 15) is not strongly affected, while the individual size (shown in Figure 16) is notably reduced. This phenomenon is produced solely by the fact of grouping individuals by the running time surrogate for size before carrying out selection and crossover stages.

In the case of the parity problem, we observe similar behaviors. The results from parallel execution were previously plotted in Figures 11 and 12. For this problem executed sequentially, the fitness (depicted in Figure 17) is significantly affected when the number of groups grows, but size is significantly reduced (as shown in Figure18).

G. CORRELATION BETWEEN AVERAGE SIZE AND TIME

In this subsection we want to show the behavior achieved while comparing average size against time for the selected GP application problem domains. Figure 19 shows the behavior of our proposed scheme when assembling the solutions in four groups on the Parity problem. We observe that solutions on average make well-defined sets with some overlap. In particular, for a given execution time, we notice high variability in program size. Nevertheless, groups with long execution times feature larger individual sizes in comparison with groups with shorter execution times. Therefore, this correlation behavior corresponds to the hypothesis shown in Figures 1 to 4. We plot the results of the Parity problem using 128 groups. Figure 20 illustrates the effect of 9 groups taken out of 128 groups. The coefficient of correlation is high in general for all groups. Figure 21 shows similar patterns for all other test problems.

H. A STATISTICAL ANALYSIS OF THE RESULTS

We have seen above that differences found among experiments seem to be quite striking, particularly when size

evolution is analyzed. Nevertheless, statistical analysis allows us to be more confident regarding the technique introduced in this paper.

The fair assessment of heuristic algorithms is a big concern in computational intelligence, and statistical analysis has proven to be one of the main approaches to carry out such study [46]–[51]. Recently, nonparametric statistical analysis brings researchers' attention to a method enabling the performance of a rigorous comparison among algorithms, considering independence, normality, and homoscedasticity. Such procedures perform both pairwise and multiple comparisons for multiple-problem analysis. In our case, we apply pairwise statistical procedures to perform individual comparisons between two algorithms (standard GP and time-GP). When the results of the designed algorithms for the same problem achieved the conditions expressed before, the most common test is the ANOVA. In case that the distributions are not normal, we must use a nonparametric test like Kruskal-Wallis. If the distributions are normal but do not achieve the property of homoscedasticity, the analysis required is the Welch test. The statistical tests serve the purpose of enabling comparison of the sample distributions, attending to the required conditions, to apply a suitable assessment a posteriori to contrast the results.

Tables 1-10 present a summary of the statistical tests performed to assess the comparisons between the methods tested above. All statistical tests were computed using Matlab. We have first studied data normality (Lilliefors, Kolmogorov-Smirnov) and homoscedasticity (Levene test); then, according to the results, we have applied the appropriate statistical test (Kruskal-Wallis, Welch, Anova) to determine if the differences are significant, using a p-value < 0.05. Similarly, Figure 22 shows sample graphs obtained from the statistical studies detailed in the tables. This result illustrates why the test rejects *Ho* (typically when size is analyzed–the results are statistically different) while in some cases it is accepted (fitness analysis for the ant and multiplexer problems–the results are not significantly different).

We can notice that when considering size-evolution differences, Tables 2, 4, 6, 8, and 10, the statistical tests show that results are different (null hypothesis *Ho* is rejected), hence we can conclude that the technique introduced in this paper reached the pursued goal. The proposed technique allows individuals to keep the size under control. Moreover, in three out of five experiments we notice that fitness differences are not statistically significant (ant, lawnmower, and multiplexer problems, see Tables 1, 3, and 5 respectively), which is our main goal (size-controlled and with fitnesses similar quality to those of regular GP). In the case that statistical tests show fitness evolution differences (parity and regression, see Tables 7 and 9 respectively), we notice that size differences are much larger than fitness differences. Also, in some specific cases, such as using 2 groups for the regression problem, fitness quality even improves. In any case, a proper comparison of computing effort and fitness attained for these

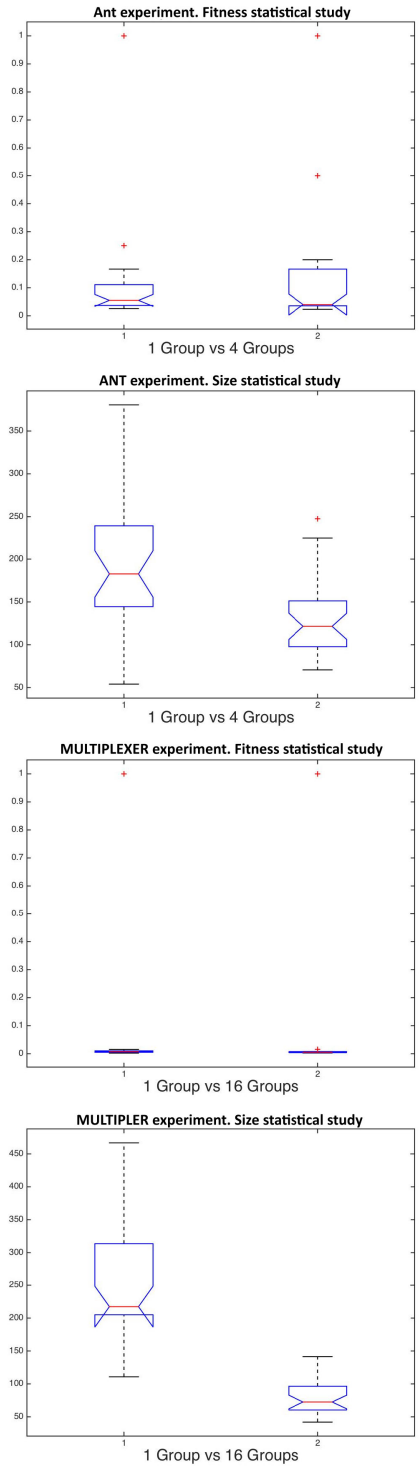


FIGURE 22. Graphical statistical analysis.

experiments allow us to more clearly observe the behavior of the experiments under the new approach.

I. EFFORT VS. FITNESS

Although in Evolutionary Algorithms, the standard fitness-generation graphs usually allow understanding of the behavior of a given algorithm, the situation changes when we

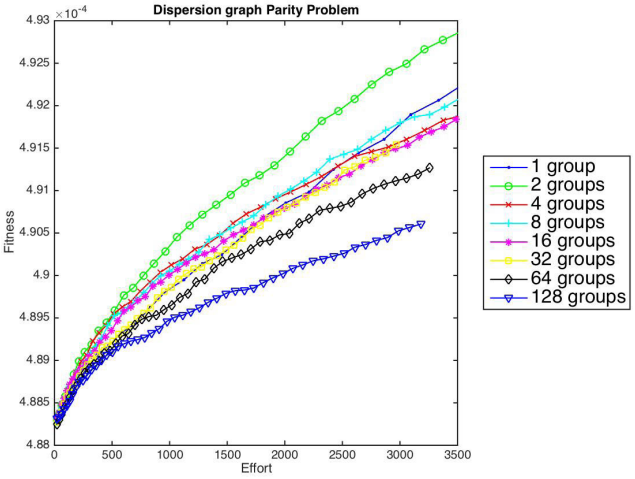


FIGURE 23. Effort vs. fitness along generations for parity problem.

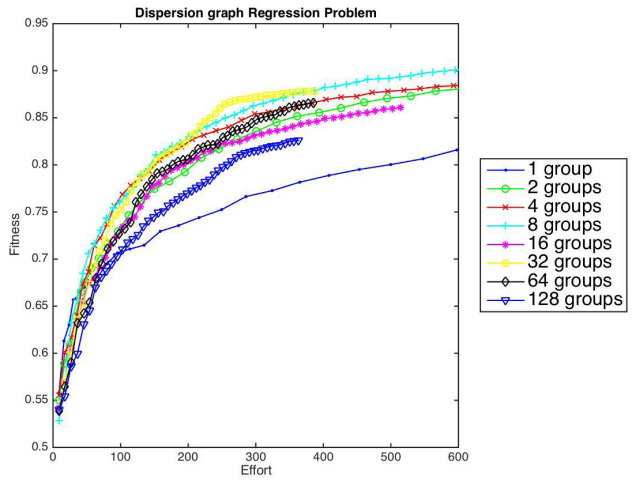


FIGURE 24. Effort vs. fitness along generations for regression problem.

compare experiments using different population sizes, or variable-size chromosomes. Here, fitness-generation comparison is not an option, given that computing effort required to complete a generation depends on individuals and population sizes. Instead, dispersion graphs showing computing effort and fitness quality must be employed.

This situation is particularly the case for GP, where individual size differences greatly influence computing time, and therefore, performance is a crucial factor. In many cases, generalizability and overfitting are also strongly influenced by individual size differences. By plotting computing effort, the effort being the number of tree nodes evaluated up to a given generation, against the fitness attained for that specific generation, we grasp a more general idea about the efficiency of the evolutionary process, [43]. We show below some of these comparisons, selecting those experiments for which fitness differences, following a statistical analysis, have been found.

Figures 23 and 24 show that for both the parity and regression problems, the selection of a proper number of groups in the experiments provides better results than the standard

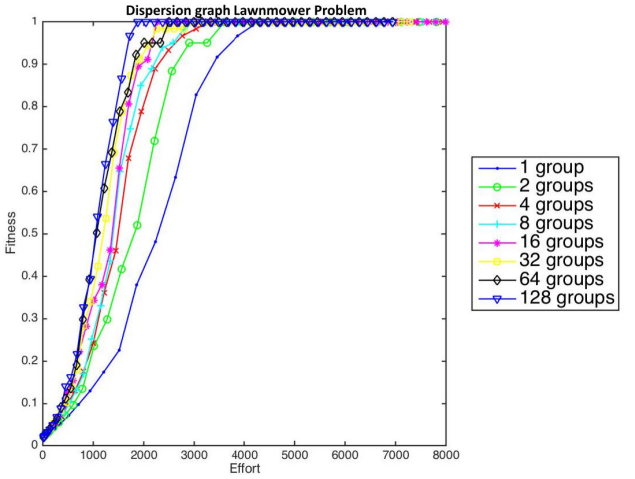
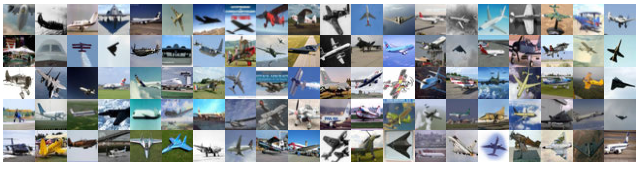
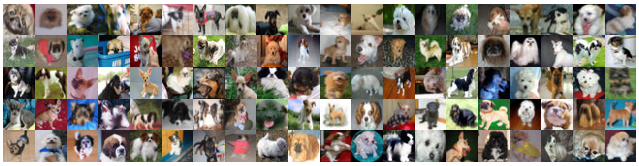


FIGURE 25. Effort vs. fitness along generations for lawnmower problem.



(a) Airplanes



(b) Dogs

FIGURE 26. Size-evolution along generations (averaged over 30 runs) for the parity problem (sequential execution).

GP model. In the case of 2, 4, or 8 groups, the best results are achieved for the parity problem, while in the regression experiments, all groups achieve better results than the standard GP.

We also show in Figure 25 the effort-fitness comparison obtained for the lawnmower problem. Again the statistical tests provided the evidence that confirms the interest of our approach. As we can see, the standard GP approach, one group, provides the worst results again.

Summarizing, we have shown that after thorough statistical analysis, the proposed time-and-individual duration method points to new research avenues where GP could be studied to address the problem of size growth from a computing time perspective. Moreover, under the new approach, this paper introduces a first method for controlling individual sizes. Its main idea was to group individuals according to evaluation time, and it naturally allows keeping individuals' sizes under control, while fitness quality remains high. The experiments and statistical data shown here allow us to see the interest of the approach in both sequential and parallel models.

Although we have focused here on the time-size relationship as well as on the specific method implemented under this perspective, we believe that new approaches may be

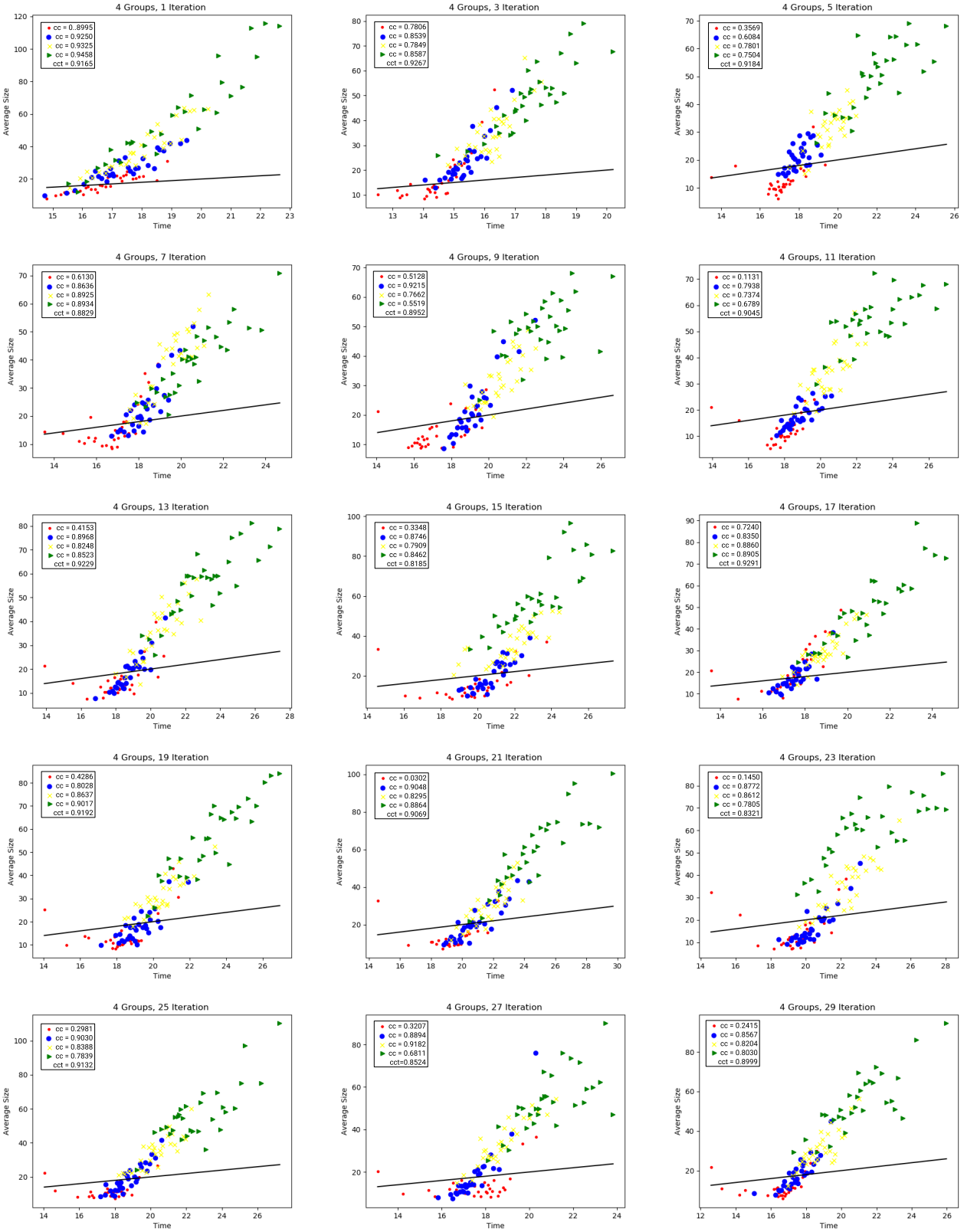


FIGURE 27. These plots show the average size of 30 jobs run for 30 generations of the object recognition problem against the corresponding time for the case of 4 groups. We can observe that for a given time the individuals at each group have different size. Note how the groups align along the vertical axis and present some sparsed elements aligned with the same trend.

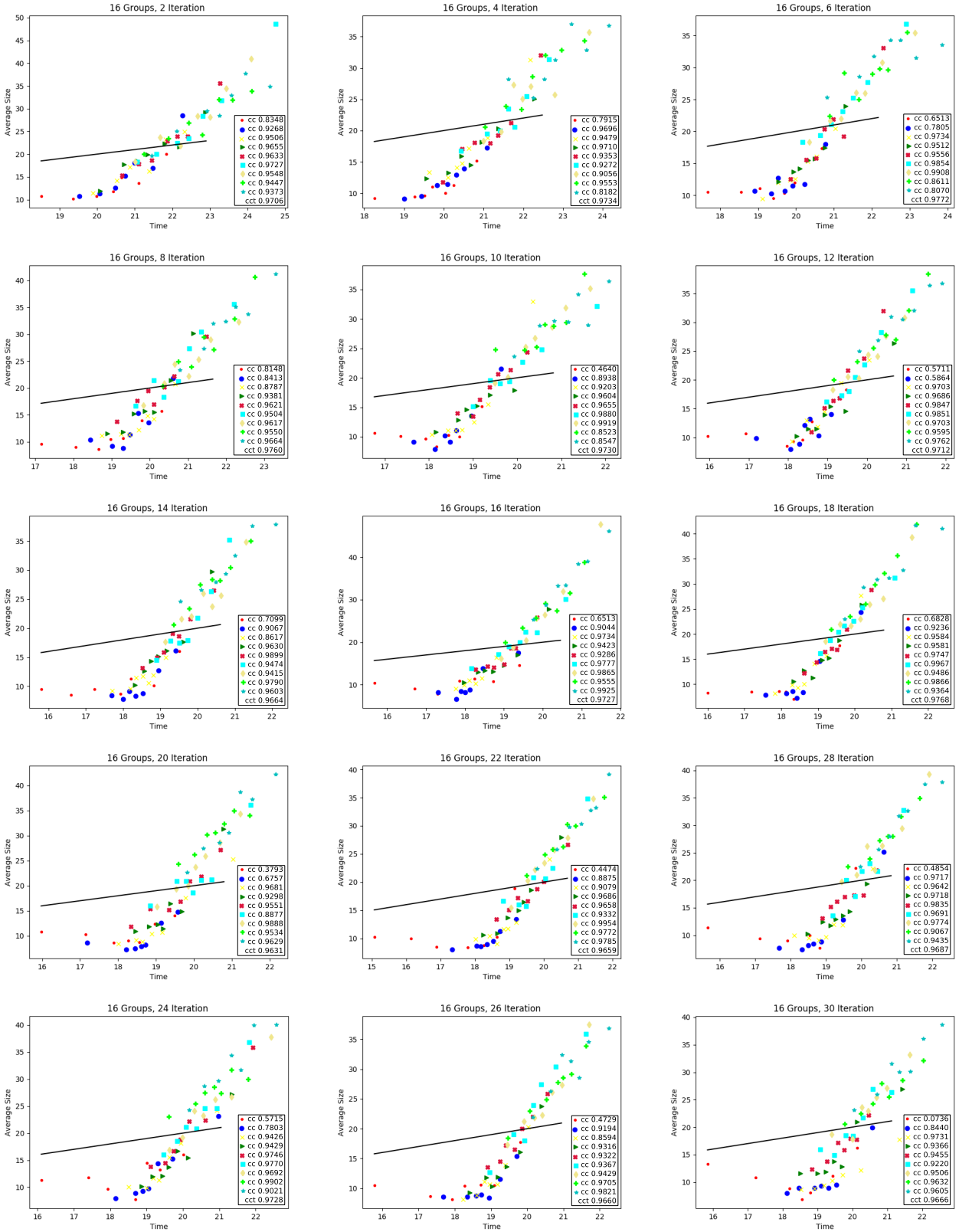


FIGURE 28. **These plots show the average size of 30 jobs run for 30 generations of the object recognition problem against the corresponding time for the case of 16 groups. We observe again that for a given time the same pattern of different individual size arise in the experiments. For visualization purpose the 16 groups are divided in 8 intervals with a step of 2 so we plot only 9 groups.**

developed in the future considering time-space relationships in variable-size chromosome-based evolutionary techniques.

Moreover, we think that specific improvements in the method presented here for GP pave the way to attain new results when

we apply these methods in parallel environments and use a more sophisticated load-balancing technique.

VII. REAL WORLD CASE: OBJECT RECOGNITION

This section presents the results of applying the idea of grouping individuals according to the computing time required for evaluation in the case of an object recognition test solved with a kind of genetic programming. The goal is to show that the correlation between execution-time and individual-size exists in real-world problems, given that this aspect is paramount to the idea and the proposed technique. The test is part of the CIFAR-10 dataset, a collection of images commonly used to train machine learning and computer vision algorithms, from which we select two classes (airplanes and dogs) to test a bioinspired algorithm, see Figure 26. The algorithm is known as the artificial visual cortex (AVC) part of the brain programming (BP) paradigm [52]. BP is a kind of deep-learning approach where neuroscience knowledge is merged with genetic programming to synthesize artificial models of the brain [53]–[57]. In the experiments, we use one thousand images per class, and each solution has multiple trees embedded within a hierarchical structure. Hence, each evaluation requires more computational resources, i.e., between 4 and 12 trees. We adapt the BP algorithm according to the proposal presented in this article. Due to multiple trees representation, the consensus to characterize individual size was to add all nodes of each tree within the AVC structure.

We offer graphs of average size vs. time similar to section VI-G. We are thus adopting the idea of evaluation per group based-on time execution, and the results of Figures 27 and 28 show 30 jobs for 30 generations of the object recognition problem considering 4 and 16 groups, respectively. Each run takes about 24 to 48 hours on a PC running Linux Ubuntu 18.04 LTS (Bionic Beaver) using Matlab 2017 with an Intel Xeon(R) Silver 4114 CPU@2.20GHz x 20 and 32 GiB. We report 30 experiments per group distribution. In the case of 4 groups with a total of 128 individuals, each group contains 32 AVC programs on average. As in the parity problem, we observe that for a given time, the individuals at each group have different sizes. Again solutions on average make well-defined sets with some overlap. In the case of 16 groups, we plot the effect of 8 groups observing that the correlation behavior corresponds to the hypothesis shown in Figures 1 to 4. The coefficient of correlation is even higher for this experiment in comparison with the previous tests.

VIII. CONCLUSION

This paper addresses the problem of measuring complexity in variable-size chromosomes based evolutionary algorithms, presents the relationship between chromosome-size and computing time, and opens opportunities that may arise when we use the latter.

Therefore, we analyzed the problem of size growth in GP under a new perspective. Instead of using an individual's size as the measure for designing a new bloat control method, we describe how a different perspective–a new one based on computing time–can shed light on the bloating phenomenon and provide clues for a new set of countermeasures. Although size and time are both sides of a single coin, associated with computational complexity, to the best of our knowledge, this is the first approach that considers computing time as the measure of individual complexity in the context of GP bloating phenomenon.

Through a series of experiments on several well-known GP benchmarks, we show that indeed, the idea works, and the specific method helps to prevent bloat. We provided some clues on how to naturally apply the proposal within GP. The idea is also elaborated to show that the method is applicable both in serial and parallel computing models and that other possible–and may be better–methods may in the future be developed based on individuals' fitness evaluation times.

To demonstrate the usefulness of the approach, we present a first method based on an individual's computing time–automatically obtained when fitness is computed–as a trait employed for characterizing and grouping individuals together in a natural way so that they can only breed within their groups. The reason for this idea is to keep computing time–and thus, indirectly, an individual's size growth–under control. Although indeed, the relationship between size and time is not direct, the method tries for the first time to use computing time to prevent bloat.

We believe that the new perspective for addressing the bloat phenomenon may open doors for research that allows the development of methods to avoid wasting time in size measurement tasks. Moreover, these new methods also allow better use of parallel and distributed infrastructures, where the bloat phenomenon profoundly influences load-balancing techniques when GP–or other variable-sized-chromosome approaches–is applied.

REFERENCES

[1] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992. [Online]. Available: https://mitpress.mit.edu/books/genetic-programming

[2] W. Banzhaf and W. B. Langdon, "Some considerations on the reason for bloat," *Genetic Program. Evolvable Mach.*, vol. 3, no. 1, pp. 81–91, 2002, doi: 10.1023/A:1014548204452.

[3] M.-A. Gardner, C. Gagné, and M. Parizeau, "Controlling code growth by dynamically shaping the genotype size distribution," *Genetic Program. Evolvable Mach.*, vol. 16, no. 4, pp. 455–498, Feb. 2015, doi: 10.1007/s10710-015-9242-8.

[4] L. Trujillo, L. Muñoz, E. Galván-López, and S. Silva, "Neat genetic programming: Controlling bloat naturally," *Inf. Sci.*, vol. 333, pp. 21–43, Mar. 2016, doi: 10.1016/j.ins.2015.11.010.

[5] W. B. Langdon and R. Poli, "Fitness causes bloat," in *Soft Computing in Engineering Design and Manufacturing*. London, U.K.: Springer, 1998, pp. 13–22, doi: 10.1007/978-1-4471-0427-8_2.

[6] F. F. de Vega, G. G. Gil, J. A. G. Pulido, and J. L. Guisado, "Control of bloat in genetic programming by means of the island model," in *Parallel Problem Solving from Nature-PPSN VIII* (Lecture Notes in Computer Science), vol. 3242. Berlin, Germany: Springer, 2004, pp. 263–271, doi: 10.1007/978-3-540-30217-9_27.

[7] P. A. Whigham and G. Dick, "Implicitly controlling bloat in genetic programming," *IEEE Trans. Evol. Comput.*, vol. 14, no. 2, pp. 173–190, Apr. 2010, doi: 10.1109/TEVC.2009.2027314.

[8] F. Fernández de Vega, G. Olague, F. O. Chávez de la, D. Lanza, W. Banzhaf, and E. Goodman, "It is time for new perspectives on how to fight bloat in GP," in *Genetic Programming Theory and Practice XVII*. Cham, Switzerland: Springer, 2019, p. 14. [Online]. Available: https://www.springer.com/gp/book/9783030399573, doi: 10.1007/978-3-030-39958-0.

[9] W. B. Langdon, T. Soule, R. Poli, and J. A. Foster, "The evolution of size and shape," in *Advances in Genetic Programming*. Cambridge, MA, USA: MIT Press, 1999, ch. 8, pp. 163–190. [Online]. Available: https://mitpress.mit.edu/books/advances-genetic-programming-volume-3

[10] S. Silva, S. Dignum, and L. Vanneschi, "Operator equalisation for bloat free genetic programming and a survey of bloat control methods," *Genetic Program. Evolvable Mach.*, vol. 13, no. 2, pp. 197–238, Nov. 2011, doi: 10.1007/s10710-011-9150-5.

[11] B. Doerr, T. Kötzing, J. A. G. Lagodzinski, and J. Lengler, "Bounding bloat in genetic programming," in *Proc. Genetic Evol. Comput. Conf. (GECCO)*, 2017, pp. 921–928, doi: 10.1145/3071178.3071271.

[12] S. Silva and E. Costa, "Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories," *Genetic Program. Evolvable Mach.*, vol. 10, no. 2, pp. 141–179, Jan. 2009, doi: 10.1007/s10710-008-9075-9.

[13] W. A. Tackett, "Recombination, selection, and the genetic construction of computer programs," Ph.D. dissertation, Dept. Comput. Sci., Univ. Southern California, Los Angeles, CA, USA, 1994. [Online]. Available: https://dl.acm.org/doi/book/10.5555/222025

[14] L. Altenberg, "The evolution of evolvability in genetic programming," in *Advances in Genetic Programming*. Cambridge, MA, USA: MIT Press, 1994, ch. 3, pp. 47–74. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.4199

[15] M. J. Streeter, "The root causes of code growth in genetic programming," in *Proc. Eur. Conf. Genetic Program.*, in Lecture Notes in Computer Science, vol. 2610. Berlin, Germany: Springer, 2003, pp. 443–454, doi: 10.1007/3-540-36599-0_42.

[16] S. Dignum and R. Poli, "Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat," in *Proc. 9th Annu. Conf. Genetic Evol. Comput. (GECCO)*, 2007, pp. 1588–1595, doi: 10.1145/1276958.1277277.

[17] F. Fernandez, L. Vanneschi, and M. Tomassini, "The effect of plagues in genetic programming: A study of variable-size populations," in *Proc. Eur. Conf. Genetic Program.*, in Lecture Notes in Computer Science, vol. 2610. Berlin, Germany: Springer, 2003, pp. 317–326, doi: 10.1007/3-540-36599-0_29.

[18] N. Wagner and Z. Michalewicz, "Genetic programming with efficient population control for financial time series prediction," in *Genetic and Evolutionary Computation Conference Late Breaking Papers*, vol. 1. San Mateo, CA, USA: Morgan Kaufmann, 2001, pp. 458–462. [Online]. Available: ftp://ftp.cs.bham.ac.uk/pub/authors/W.B.Langdon/biblio/gp-html/wagner_2001_gpepcftsp.html

[19] S. Bleuler, M. Brack, L. Thiele, and E. Zitzler, "Multiobjective genetic programming: Reducing bloat using SPEA2," in *Proc. Congr. Evol. Comput.*, vol. 1, May 2001, pp. 536–543. [Online]. Available: https://ieeexplore.ieee.org/document/934438

[20] R. Poli, "A simple but theoretically-motivated method to control bloat in genetic programming," in *Proc. Eur. Conf. Genetic Program.*, in Lecture Notes in Computer Science, vol. 2610. Berlin, Germany: Springer, 2003, pp. 204–217, doi: 10.1007/3-540-36599-0_19.

[21] E. Alfaro-Cid, J. J. Merelo, F. F. de Vega, A. I. Esparcia-Alcázar, and K. Sharman, "Bloat control operators and diversity in genetic programming: A comparative study," *Evol. Comput.*, vol. 18, no. 2, pp. 305–332, Jun. 2010. [Online]. Available: https://www.mitpressjournals.org/doi/10.1162/evco.2010.18.2.18206

[22] S. Luke and L. Panait, "A comparison of bloat control methods for genetic programming," *Evol. Comput.*, vol. 14, no. 3, pp. 309–344, Sep. 2006, doi: 10.1162/evco.2006.14.3.309.

[23] S. Dignum and R. Poli, "Operator equalisation and bloat free gp," in *Proc. Eur. Conf. Genetic Program.*, in Lecture Notes in Computer Science, vol. 4971. Springer, 2008, pp. 110–121, doi: 10.1007/978-3-540-78671-9_10.

[24] S. Silva, "Reassembling operator equalisation: A secret revealed," *ACM SIGEVOlution*, vol. 5, no. 3, pp. 10–22, Sep. 2011, doi: 10.1145/2043118.2043120.

[25] M. Tomassini, *Spatially Structured Evolutionary Algorithms*. Berlin, Germany: Springer-Verlag, 2005. [Online]. Available: https://www.springer.com/gp/book/9783540241935

[26] F. Fernández, M. Tomassini, and L. Vanneschi, "An empirical study of multipopulation genetic programming," *Genetic Program. Evolvable Mach.*, vol. 4, no. 1, pp. 21–51, 2003, doi: 10.1023/A:1021873026259.

[27] G. Folino, C. Pizzuti, and G. Spezzano, "A cellular genetic programming approach to classification," in *Proc. Conf. Genetic Evol. Comput.* San Mateo, CA, USA: Morgan Kaufmann, 1999, pp. 1015–1020. [Online]. Available: https://dl.acm.org/doi/pdf/10.5555/2934046.2934058

[28] M. Tomassini, L. Vanneschi, F. Fernández, and G. Galeano, "A study of diversity in multipopulation genetic programming," in *Proc. Int. Conf. Artif. Evol.*, in Lecture Notes in Computer Science, vol. 2936. Springer, 2004, pp. 243–255, doi: 10.1007/978-3-540-24621-3_20.

[29] E. Cantú Paz, *Efficient and Accurate Parallel Genetic Algorithms*. Boston, MA, USA: Springer, 2001. [Online]. Available: https://www.springer.com/gp/book/9780792372219, doi: 10.1007/978-1-4615-4369-5.

[30] G. Galeano, F. Femdndez, M. Tomassini, and L. Vanneschi, "Studying the influence of synchronous and asynchronous parallel GP on programs length evolution," in *Proc. Congr. Evol. Comput. (CEC)*, vol. 2, May 2002, pp. 1727–1732, doi: 10.1109/CEC.2002.1004503.

[31] F. Fernandez, G. Galeano, J. A. Gomez, and J. M. Sanchez, "Efficient use of computational resources in genetic programming: Controlling the bloat phenomenon by means of the island model," in *Proc. IEEE 28th Annu. Conf. Ind. Electron. Society. (IECON)*, vol. 3, Nov. 2002, pp. 2520–2524, doi: 10.1109/IECON.2002.1185370.

[32] A. Osman and H. Ammar, "Dynamic load balancing strategies for parallel computers," in *Proc. Int. Symp. Parallel Distrib. Comput.*, vol. 11, 2002, pp. 110–120. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.375

[33] M. J. Zaki, W. Li, and S. Parthasarathy, "Customized dynamic load balancing for a network of workstations," *J. Parallel Distrib. Comput.*, vol. 43, no. 2, pp. 156–162, Jun. 1997, doi: 10.1006/jpdc.1997.1339.

[34] S. Nesmachnow, H. Cancela, and E. Alba, "Heterogeneous computing scheduling with evolutionary algorithms," *Soft Comput.*, vol. 15, no. 4, pp. 685–701, Mar. 2010, doi: 10.1007/s00500-010-0594-y.

[35] K. Mesghouni, S. Hammadi, and P. Borne, "Evolutionary algorithms for job-shop scheduling," *Int. J. Appl. Math. Comput. Sci.*, vol. 14, no. 1, pp. 91–103, 2004. [Online]. Available: https://www.researchgate.net/publication/228926955

[36] T. Estrada, O. Fuentes, and M. Taufer, "A distributed evolutionary method to design scheduling policies for volunteer computing," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 36, no. 3, pp. 40–49, 2008, doi: 10.1145/1366230.1366282.

[37] J. K. Cochran, S.-M. Horng, and J. W. Fowler, "A multi-population genetic algorithm to solve multi-objective scheduling problems for parallel machines," *Comput. Oper. Res.*, vol. 30, no. 7, pp. 1087–1102, 2003, doi: 10.1016/S0305-0548(02)00059-X.

[38] G. Greenwood, A. Gupta, and V. Mahadik, "Multiprocessor scheduling of high concurrency algorithms," in *Proc. 7th Annu. Florida Artif. Intell. Res. Symp., Session Genetic Algorithms Artif. Intell.*, 1994, pp. 265–269. [Online]. Available: https://www.researchgate.net/publication/2426383

[39] A. Y. Zomaya and Y.-H. Teh, "Observations on using genetic algorithms for dynamic load-balancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 9, pp. 899–911, Sep. 2001, doi: 10.1109/71.954620.

[40] M. Oussaidène, B. Chopard, O. V. Pictet, and M. Tomassini, "Parallel genetic programming and its application to trading model induction," *Parallel Comput.*, vol. 23, no. 8, pp. 1183–1198, 1997, doi: 10.1016/S0167-8191(97)00045-8.

[41] F. F. de Vega, J. G. A. Sánchez, and C. Cotta, "A preliminary analysis and simulation of load balancing techniques applied to parallel genetic programming," in *Proc. Int. Work-Conf. Artif. Neural Netw.*, in Lecture Notes in Computer Science, vol. 6692. Springer, 2011, pp. 308–315, doi: 10.1007/978-3-642-21498-1_39.

[42] L. Vanneschi, M. Castelli, and S. Silva, "Measuring bloat, overfitting and functional complexity in genetic programming," in *Proc. 12th Annu. Conf. Genetic Evol. Comput. (GECCO)*, 2010, pp. 877–884, doi: 10.1145/1830483.1830643.

[43] F. F. de Vega, "Distributed genetic programming models with application to logic synthesis on FPGAS," Ph.D. dissertation, Departamento de Tecnología Computadores y Comunicaciones, Univ. Extremadura, Badajoz, Spain, 2001. [Online]. Available: http://gpbib.pmacs.upenn.edu/gp-html/fernandez_thesis.html

[44] S. Luke and L. Panait, "Lexicographic parsimony pressure," in *Proc. Conf. Genetic Evol. Comput.*, 2002, pp. 829–836. [Online]. Available: https://dl.acm.org/doi/10.5555/2955491.2955636

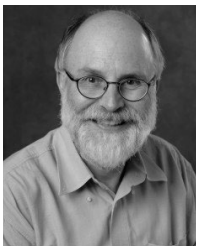
[45] D. R. White, "Software review: The ECJ toolkit," *Genetic Program. Evolvable Mach.*, vol. 13, no. 1, pp. 65–67, Aug. 2011, doi: 10.1007/s10710-011-9148-z.
[46] M. Coffin and M. J. Saltzman, "Statistical analysis of computational tests of algorithms and heuristics," *INFORMS J. Comput.*, vol. 12, no. 1, pp. 24–44, Feb. 2000, doi: 10.1287/ijoc.12.1.24.11899.
[47] Z. Lei and L. Ren-hou, "Designing of classifiers based on immune principles and fuzzy rules," *Inf. Sci.*, vol. 178, no. 7, pp. 1836–1847, Apr. 2008, doi: 10.1016/j.ins.2007.11.019.
[48] S. García, A. Fernández, J. Luengo, and F. Herrera, "A study of statistical techniques and performance measures for genetics-based machine learning: Accuracy and interpretability," *Soft Comput.*, vol. 13, no. 10, pp. 959–977, Dec. 2008, doi: 10.1007/s00500-008-0392-y.
[49] J. Luengo, S. García, and F. Herrera, "A study on the use of statistical tests for experimentation with neural networks: Analysis of parametric test conditions and non-parametric tests," *Expert Syst. Appl.*, vol. 36, no. 4, pp. 7798–7808, May 2009, doi: 10.1016/j.eswa.2008.11.041.
[50] S. García, A. Fernández, J. Luengo, and F. Herrera, "Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power," *Inf. Sci.*, vol. 180, no. 10, pp. 2044–2064, May 2010, doi: 10.1016/j.ins.2009.12.010.
[51] J. Derrac, S. García, D. Molina, and F. Herrera, "A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms," *Swarm Evol. Comput.*, vol. 1, no. 1, pp. 3–18, Mar. 2011, doi: 10.1016/j.swevo.2011.02.002.
[52] G. Olague, E. Clemente, D. E. Hernandez, A. Barrera, M. Chan-Ley, and S. Bakshi, "Artificial visual cortex and random search for object categorization," *IEEE Access*, vol. 7, pp. 54054–54072, 2019, doi: 10.1109/ACCESS.2019.2912792.
[53] G. Olague, D. E. Hernández, P. Llamas, E. Clemente, and J. L. Briseño, "Brain programming as a new strategy to create visual routines for object tracking," *Multimedia Tools Appl.*, vol. 78, no. 5, pp. 5881–5918, Sep. 2018, doi: 10.1007/s11042-018-6634-9.
[54] G. Olague, D. E. Hernandez, E. Clemente, and M. Chan-Ley, "Evolving head tracking routines with brain programming," *IEEE Access*, vol. 6, pp. 26254–26270, 2018, doi: 10.1109/ACCESS.2018.2831633.
[55] D. E. Hernández, G. Olague, B. Hernández, and E. Clemente, "CUDA-based parallelization of a bio-inspired model for fast object classification," *Neural Comput. Appl.*, vol. 30, no. 10, pp. 3007–3018, Feb. 2017, doi: 10.1007/s00521-017-2873-3.
[56] D. E. Hernández, E. Clemente, G. Olague, and J. L. Briseño, "Evolutionary multi-objective visual cortex for object classification in natural images," *J. Comput. Sci.*, vol. 17, pp. 216–233, Nov. 2016, doi: 10.1016/j.jocs.2015.10.011.
[57] E. Clemente, F. Chavez, F. Fernandez de Vega, and G. Olague, "Self-adjusting focus of attention in combination with a genetic fuzzy system for improving a laser environment control device system," *Appl. Soft Comput.*, vol. 32, pp. 250–265, Jul. 2015, doi: 10.1016/j.asoc.2015.03.011.

FRANCISCO FERNÁNDEZ DE VEGA (Senior Member, IEEE) received the Ph.D. degree, in 2001. He is currently a Professor of computer architecture with the University of Extremadura. He is interested in parallel and distributed evolutionary algorithms, with applications to computational creativity. He has received several awards, including the 2013 ACM Gecco Evolutionary Art, Design, and Creativity Competition. He received the Best Ph.D. Engineering Award from the University of Extremadura. He is also the Vice-Chair of the Task Force on Creative Intelligence and the IEEE Computational Intelligence Society and the Director of the GEA Research Group and the Escuelas Municipales de Jóvenes Científicos (Municipal Schools for Young Scientists) founded, in 2015, largest STEM initiative in Spain. His evolutionary produced artworks have been displayed around the world: Vancouver, Cancún, Amsterdam, Madrid, and Paris.

GUSTAVO OLAGUE (Senior Member, IEEE) was born in Chihuahua, Mexico, in 1969. He received the B.S. and M.S. degrees in industrial and electronics engineering from the Instituto Tecnológico de Chihuahua (ITCH), in 1992 and 1995, respectively, and the Ph.D. degree in computer vision, graphics, and robotics from the Institut Polytechnique de Grenoble (INPG) and Institut National de Recherche en Informatique et en Automatique (INRIA), France, in 1998. He is currently a Professor with the Department of Computer Science, Centro de Investigación Científica y de Educación Superior de Ensenada (CICESE), Mexico, and the Director of the EvoVisión Research Team. He is also an Adjunct Professor of engineering with the Universidad Autonóma de Chihuahua (UACH). He has authored over 100 conference proceedings papers and journal articles. His main research interests are evolutionary computing and computer vision. He is the author of *Evolutionary Computer Vision: The First Footprints* (Springer) in the Natural Computing Series. He has received numerous distinctions, among them the Talbert Abrams Award presented by the American Society for Photogrammetry and Remote Sensing (ASPRS) for authorship and recording of current and historical engineering and scientific developments in photogrammetry, the Best Paper Awards at major conferences, such as GECCO, the European Workshop on Evolutionary Computation in Image Analysis, Signal Processing, and Pattern Recognition (EvoIASP), and European Workshop on Evolutionary Hardware Optimization (EvoHOT), and twice the Bronze Medal at the Humies (GECCO Award for Human-Competitive results produced by genetic and evolutionary computation). He co-edited special issues in *Pattern Recognition Letters*, *Evolutionary Computation* (MIT Press), *Applied Optics*. He has served as the Co-Chair for the Real-World Applications track at the leading international evolutionary computing conference, the GECCO (ACM SIGEVO Genetic and Evolutionary Computation Conference).

DANIEL LANZA received the B.S. degree in telematics engineering from the University of Extremadura, Spain, in 2015, and the M.S. degree in visual analytics and big data from the Universidad Internacional de la Rioja, Spain, in 2016. He made research stays at CICESE, CERN, and Michigan State University. He is currently a Software Engineer at Google working in YouTube. He loves to work in the problem of joining the following two research areas: big data and evolutionary computation.

FRANCISCO CHÁVEZ DE LA O received the Ph.D. degree in computer science from the University of Extremadura, in 2012. Since 2001, he has been with the Department of Engineering of Computer Science and Telematics Systems, University of Extremadura. He is currently an Assistant Professor and belongs to the Artificial Evolution Research Group. It has over 50 National and International publications. He has worked as a Researcher in several research projects granted by the Spanish Government, he has been a Principal Investigator in EPHEMCH and DEEPBIO. He has also worked as a Researcher in several research projects awarded by the Autonomous Community of Extremadura. He recently leads the project granted by the Autonomous Community of Extremadura, where he performs the tasks of the Principal Investigator. Finally, he has worked on several projects with companies in the sector in the field of technology transfer with companies. He has three patents. His lines of research focus on systems based on fuzzy rules and diffuse genetic systems, image processing through deep learning, and massive data processing.



WOLFGANG BANZHAF is currently the John R. Koza Chair of genetic programming and a Professor with the Department of Computer Science and Engineering, Michigan State University. His research interests are in the field of bio-inspired computing, notably evolutionary computation, and complex adaptive systems. He is also interested in the studies of self-organization and the field of artificial life. He recently become more involved with network research as it applies to natural and man-made systems.

ERIK GOODMAN is currently a Professor of electrical and computer engineering, and an Adjunct Professor of mechanical engineering with Michigan State University. He is also the Executive Director of the BEACON Center for the Study of Evolution in Action, an NSF Science and Technology Center founded, in August 2010. BEACON conducts multidisciplinary research on evolution in the lab and field, in digital organisms in the computer, and in evolutionary computation used to solve problems in engineering and computer science. His personal research centers on evolutionary computation, particularly heterogeneous and parallel genetic algorithms and genetic programming. He is also a Co-Founder of Red Cedar Technology, Inc., where he wrote the SHERPA design optimization software now sold by Siemens PLM Software. He is also studying optimization of solid fuel rocket grains, and involved in information and communications technology outreach in schools in Africa.

JOSE MENENDEZ-CLAVIJO was born in Cienfuegos, Cuba, in 1988. He received the B.S. degree in informatics engineering from the University of Cienfuegos "Carlos Rafael Rodríguez," in 2012. He is currently pursuing the M.S. degree in computer science with the EvoVisión Laboratory, CICESE Research Center. He is also working at the EvoVisión Laboratory, CICESE Research Center. His research interests are computer vision, genetic programming, evolutionary algorithms, and brain programming.

AXEL MARTINEZ was born in Distrito Federal, Mexico, in 1989. He received the B.S. degree in mechatronics engineering from Universidad Anáhuac México Norte, in 2017. He is currently pursuing the M.S. degree in computer science with the EvoVisión Laboratory, CICESE Research Center. He is also working at the EvoVisión Laboratory, CICESE Research Center. His research interests are computer vision, genetic programming, evolutionary algorithms, and 3-D Reconstruction. He also enjoys surfing and cooking.