Improving the Generalisation of Genetic Programming Models with Evaluation Time and Asynchronous Parallel Computing*

Firstname Lastname
Institute Name
City, City
email@institution.com

Firstname Lastname
Institute Name
City, City
email@institution.com

Firstname Lastname
Institute Name
City, City
email@institution.com

Firstname Lastname
Institute Name
City, City
email@institution.com

Firstname Lastname
Institute Name
City, City
email@institution.com

ABSTRACT

A reason for seeking simple machine-learning models is to avoid overfitting. In genetic programming(GP), controlling complexity often means reducing the size of evolved expressions. However, previous studies showed that size reduction may not avoid model overfitting. Therefore, in this study, we use the evaluation time—the computational time required to evaluate a GP model on data — as the estimate of model complexity. The evaluation time depends not only on the size of evolved expressions but also their composition, thus acting as a more nuanced measure of model complexity than the expression size alone. To constrain complexity using this measure of complexity, we employ an explicit control technique and a method that creates a race condition during the evolution, named the asynchronous parallel GP(APGP). To facilitate the study of overfitting, we boosted the training performance of GP by using a method that discovers features and leverages multiple linear regression (MLRGP). While using MLRGP, we compared the evaluation time-control methods with MLRGP with no complexity control and MLRGP with an effective bloat-control technique. Also we compare the methods with MLRGP version that discourages the unnecessary growth of the features that MLRGP discovers. The results show that constraining evaluation time leads to better generalisation than constraining size.

CCS CONCEPTS

• **Computing methodologies** → **Genetic programming**; **Classification and regression trees**;

KEYWORDS

genetic programming, generalisation, regression, complexity, feature engineering

*Produces the permission block, and copyright information

GECCO '21, July 10–14, 2021, Lille, France

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Firstname Lastname, Firstname Lastname, Firstname Lastname, Firstname Lastname, and Firstname Lastname. 2021. Improving the Generalisation of Genetic Programming Models with Evaluation Time and Asynchronous Parallel Computing. In *Proceedings of the Genetic and Evolutionary Computation Conference 2021 (GECCO '21).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

It has always been an important challenge in machine learning (ML) to avoid generating models that fit the training data very well but without generalising to the unseen data; this is termed overfitting. Often these overfitting models are overly complex model [30]; however, determining how much complexity is just enough is a challenge. The challenge is exacerbated because we cannot universally define what complexity in ML is and that in turn makes it hard to find an effective mechanism to control it.

In this study we focus on the complexity challenges in Genetic Programming (GP) [20] [21], and in particular, in GP systems aided by Multiple Linear Regression (MLR) [33]. Such *MLRGP* systems [1, 12, 29] have become increasingly popular lately because they improve the training performance significantly; this is because the traditional GP often underfits the data because it can not efficiently generate numeric constants [4, 18]. However, this improved training accuracy can still result in serious overfitting [33].

Another traditional concern in GP is complexity, which is often manifested by a tendency to grow model sizes to a point that renders the evolutionary search process ineffective[37]. The most popular approach to controlling complexity in GP is bloat control, that is to limit the growth in size of the evolved expressions. However, previous studies have shown that bloat control alone does not always overcome the model overfitting problem [5, 40]. This begs the question: is bloat control really complexity control?

To address the above limitation in bloat control, recent literature [9, 34, 35] has proposed alternative approaches to control the computational complexity of models in GP. Instead of using size as a measure of complexity, these approaches propose the use of evaluation time — the computational time it takes to evaluate a GP model on data. The use of evaluation time as a measure of complexity is built on the observation that a model that is made up of computationally expensive building blocks or that has large structures takes

a long time to be evaluated, and hence it is computationally complex. The work in [34] empirically shows how the functional and structural complexity are different by plotting the evaluation times of identically sized but functionally diverse GP models. Therefore, if the evaluation time of evolving models are constrained then the growth in the structural as well as the functional complexity will be discouraged. The same work also recommended various techniques to significantly minimise the noise in measuring evaluation times. This paper adopts the use of all these recommendations to measure complexity of the evolving models in MLRGP.

The next question is how to control the evaluation times. We used two approaches to control evaluation times. First, we use bloat control methods that explicitly penalise or discourage high evaluation times in the same way bloat control techniques penalise size. Instead of subjectively penalising the slow evaluation times, the second approach takes a simple view: induce a *race* among competing models that allows a model to join the breeding population as soon as it has finished evaluating; this GP method is known as the *Asynchronous Parallel Genetic Programming* (APGP) [34]. With APGP, the faster models can (if their fitness is competitive) join the breeding population before their slower counterparts and gain an evolutionary advantage. This advantage arises because the competing models compete in terms of not only their accuracy but also their evaluation times due to the race; this is quite unlike in standard GP where each evaluation (or a batch of evaluations, as in generational replacement) is allowed to finish before the next evaluation (or a batch of evaluations) can start. Note, however, that all individuals have to go through selection as usual to get into the population; selection is solely based on accuracy. Therefore, APGP facilitates a dynamic interplay between accuracy and simplicity. To induce this race, APGP breeds and evaluates multiple models simultaneously across multiple asynchronous threads.

All the experiments used the MLRGP method. We compare the performance of the methods that use evaluation time (APGP and explicit time control) with MLRGP without complexity control and MLRGP with an effective bloat control technique. Also, them with MLRGP version that discourages the unnecessary growth of the features that MLRGP discovers. We used ten test problems that tend to overfit when MLRP is applied.

The rest of this paper is organised as follows. Section 2 provides some background information, Section 3 details the methods that were employed in this study, and Section 4 details the experimental setup. The results of the experiments are presented and discussed in Section 5. Finally, the conclusions and further work are given in Section 6.

2 BACKGROUND

Genetic programming (GP) uses a guide random approach to enable computers to automatically build models and to program themselves; the approach is inspired by the Darwinian principle of natural evolution. The most popular representation of a GP individual is a variable tree-based structure[14]; several other representations exist [2, 8, 15, 32, 38, 45]. Variability of the structures is a common feature of the representations that introduces challenges such as uncontrolled growth in the size of evolving code, also termed code bloat [31].

2.1 Complexity in Genetic Programming

Finding an appropriate notion of complexity and implementing a means of controlling it can be challenging tasks [40]. Many notions and techniques have been proposed that can not be fully reviewed here due to space constraints. Therefore, we briefly examine some approaches, broadly.

The traditional and most popular approach of managing complexity in GP is to work with the representation of a model, the structural complexity. This approach considers the number of nodes, the encapsulated subtrees and the number of layer to determine complexity; this is termed bloat control. However, they ignore the underlying functional or computational complexity of the expression. This is a significant shortcoming. While bloat control penalises the expression of a linear equation $5x + 4x + 3x + 2x + x$ that is large, a smaller but computationally more complex expression $sin(x)$ will not be penalised [6]; the smaller $sin(x)$ is more complex because it is equivalent to its Taylor series expansion $\sum_{n=0}^{\infty}(-1)^n \frac{x^{2n+1}}{(2n+1)!}$ and will use up more resources to execute than the linear equation. Therefore, the complexity of a GP individual is more than the size of its expression. Many bloat control techniques have been proposed [24]. Other related studies have proposed bloat control measurements [40], bloat control approaches that are Kolmogorov-based [11], minimum description length principle based [13][16], and those that use invariance theorem [26].

Another approach, known as functional based approaches, recognises that small structures may be more complex than large structures. An example of these approaches determines functional complexity by approximating the the evolved expressions by polynomials [44]. The study considers expressions that are approximated by polynomials of a high degree as more complex than those approximated with low degrees; this is based on the idea that the complex ones are approximated by polynomials of high degree because of a large degree of oscillations in the response of the function, termed the *order of non-linearity*. However, the minimising this measure of complexity requires that the evolved expressions are twice differentiable, this is a property that can not be guaranteed. To avoid the requirement of twice-differentiability, another study [40] proposed a measure of functional complexity that uses the slope of the response surface of the expression along each feature dimension. However, the this approach is error-prone and computationally expensive; also, the study only showed how complexity can be measured and did not show how it can be controlled.

Complexity is also being managed using concepts from statistical learning theory. These include the generalisation error-bound Vapnik–Chervonenkis (VC) theory and Rademacher complexity theory [22] [43]. As a general measure of the capacity or complexity of a learning machine [42] [41], the VC dimension is the maximum number of vectors that can be separated (shattered) into two classes in all possible ways by a set of functions [42]. This has been used to provide various estimations of generalisation errors. Another related framework is Structural risk minimisation (SRM). It uses the VC dimension to assess the generalisation ability of a learning machine [7]; this involves predicting the distance between the test and training errors to define the upper bound of the generalisation error. While the proposed method generalised better than standard GP, the authors acknowledged the expensive computational cost

of the method and a need for further study of parameters. Finally, another related study [3] measured functional complexity by calculating the variance of the outputs of evolving expressions and explicitly minimised it in a multi-objective optimisation approach.

The functional complexity approaches and the statistical learning approaches are referring to mathematical expression. Therefore, their application is limited in scope, especially because GP is a versatile tool that has broad application, such as in automatic programming, design and data modelling. Also, implementing these techniques is often non-trivial. In contrast, the measure of complexity that we use in this study considers computational complexity, the evaluation time. This promises to be more broadly applicable than the complexity measures highlighted above. The use of evaluation time is nascent in GP [34][35][9].

2.2 Time as an Indicator of Complexity

We ran some test to empirically verify that controlling evaluation time (computational complexity) is not the same as controlling size, as argued in the previous section. This enabled us to determine if evaluation time can differentiate between the functional complexity of expressions that are of the same size; also, as evaluation time increases with evaluation times it is important to verify the impact of functional complexity on the evaluation time.

2.2.1 **Time is not Size.** To demonstrate that the evaluation time reflects more than size, we grouped mathematical operators (functions) into four groups by their complexity and then generated symbolic regression models using each of the groups. Each group had different sized individuals ranging from 10 to 300 node lengths; also for each size 30 random expressions were produced. All models were then evaluated and the average evaluation times of each node size in a group was plot in Figure 1; the four plots represent the four groups, respectively.

From the graphs in Figure 1, we can make some clearly observations. First, the evaluation times of the individuals that were produced with functionally complex operators are consistently higher than same size individuals made up of less functionally complex operators. Therefore, evaluation time is able to discriminate between functional complexity. Secondly, as expected, the evaluation time shows a strong correlation with size. From these observations, we can determine that controlling complexity through controlling evaluation time will work conditionally: if the sizes have converged to similar sizes, within a certain tolerance, it will curbs functional complexity; otherwise, it curbs the growth in the sizes of the individuals. The tolerance in increases as the sizes of the individuals increase. As an example, an ADD-SUB individual of size 175 has the same evaluation time as the COS-SIN of size 75.

These findings predict the limiting behaviour of evaluation time control in GP. When the population if functionally diverse but a size-converged (where bloat control is not useful) the evaluation times discriminates between functional complexities. On the other hand, when sizes in the population is diverse and the functional complexity has converged then evaluation times discriminate between sizes.

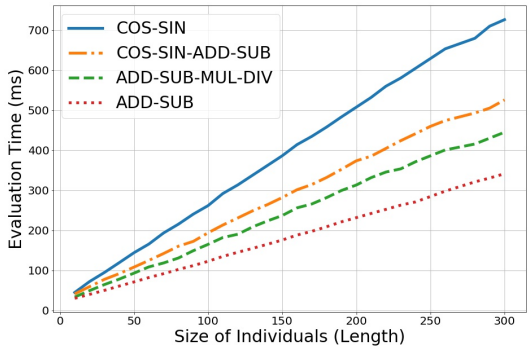


Figure 1: The evaluation time of models is affected by both their size and their composition. Higher average evaluation times were given by individuals made up of COS and SIN operators than same-sized individuals made up of simpler functions sets. Also, evaluation time correlates with size.

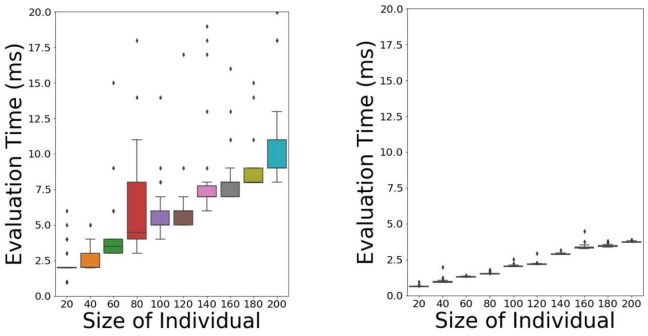
2.2.2 **Complexity and Number of Features.** Instead of a monolithic structure, the MLRGP method that we use for our experiments finds features that make up an individual model; this is detailed in Section 2.3. Also, the method evaluates each feature independently with the training data. This means that increasing the number of features will increase the evaluation times. When we examined the final populations that were produced by MLRGP, we observed a stronger correlation between the evaluation times and the number of features of the individuals than with with evaluation times and the sizes of the individuals. Figure 3 shows this correlation in the final populations of MLRGP on the test problem 2; this represents 1,500 individuals across 30 generations.

This implies that discouraging evaluation times will discourage all that contribute to evaluation times including the size, the computational complexity of the components (e.g. sin vs add) and the number of features. On the other hand, discouraging size alone (bloat control) ignores the complexity of the components and the number of features within the individual. After all, two individuals of the same size can have different numbers of features and exhibit different functional behaviours.

2.2.3 **Reliability of Evaluation Time Measurements.** Evaluation times measurements can vary across multiple executions. To allow for the use of a single evaluation to get a reliable estimate of complexity, we had to address this variability. Though this challenge can not be totally eliminated (CPU scheduling is the prerogative of the operating system kernel), we found ways to significantly minimise the variation. This include apply CPU management options: (1) disabling background services, (2) locking CPU Speed so that the operating system does not change it during operations like power management, (3) running the GP runs on dedicated processors, and (4) raising the priority of the GP runs on the processor. Also, our implementation used a Python (3.3 and above) CPU-time-based function that uses CPU performance counters [36] to measure the evaluation time. The function offers high

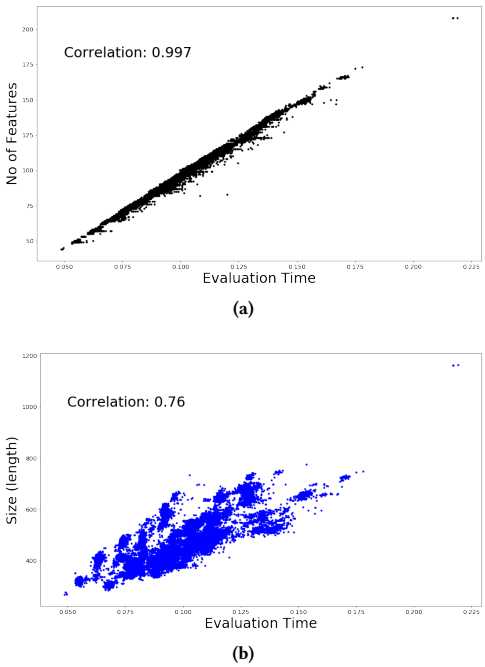
resolution (in nanoseconds) across platforms, while the returned values are in fractional seconds.

The impact of these settings is show in Figure 2. The box-plots in the figures are for multiple evaluation time readings for individuals of different sizes; Figure 2a and Figure 2b show readings before and after applying the CPU settings, respectively. These approaches allowed us to reliably use a single execution to measure the evaluation time.



(a) Before (b) After

Figure 2: Improvement in the consistency of evaluation time measurements after applying CPU options.



(a)

(b)

Figure 3: The correlation between the evaluation time and (a) the number of features (a) and (b) size.

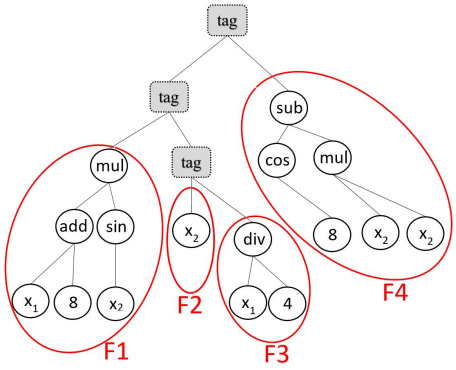


Figure 4: An MLRGP representation of a model. The nodes labeled *tag* are placeholders that are not evaluated; and the subtrees F1 - F4 are treated as features.

2.3 Overcoming Under-fitting in GP

In this section, we motivate and introduce the MLRGP method that we have used for the experiments. The history of GP's struggle with manufacturing the requisite constants is well documented [4, 19] and dates back to Koza's early work [21]. Recent work such as [33] and [17] in regression and classification respectively, however, has shown the merit in hybridising GP with statistical and machine learning techniques to help circumvent the problem of tuning constants in GP; in fact, the performance boost can be significant. Because our work targets regression problems, we adopt the approach discussed in [33] that identifies *features* in GP trees and then leverages multiple linear regression to tune the coefficients of these features. As shown in Figure 3, the time cost of a model with a high number of features is different from that of a model that is simply big in size. Therefore, a complexity control method that can differentiate between time and size is better able to control the complexity of such a system.

The use of MLRGP is also motivated by the fact that containing overfitting truly makes sense if the training scores are high. The work in [33] argues that GP often heavily underfits without using MLR, but overfits with it. Therefore, much as the cited work on evaluation-time based complexity control has reduced overfitting in GP without the use of MLR, we test if the same can also reduce overfitting in MLRGP.

The MLRGP uses a modified version of the standard tree representation of symbolic regression models; an example is shown in Figure 4. The new type of node (labeled *tag*) is introduced to act as a placeholder, that is not evaluated. The placeholder was defined with an arity of two. This enables it to either branch out further creating more placeholders or contain a feature directly below it. As marked in Figure 4, F1, F2, F3 and F4 are subtrees that act as independent features. The only constraint required for the MLRGP tree to remain valid is that *tag* nodes can only have *tag* nodes as parent nodes. The genetic operators (mutation and crossover) must obey this constraint. Besides respecting this constraint, genetic operators (crossover and mutation) proceed as usual. This means that they can act within a feature or on a set of features.

To evaluate an MLRGP individual, the followings steps must be taken. First, the features are identified by noting the *tag* nodes. Second, each feature is evaluated with data. Third, the output of the evaluating features are used to create a design matrix (a column per feature) that the MLR will then use to tune the coefficients (one coefficient per column, plus an intercept term). Next, the coefficients that MLR produced for the features are used to create a final model. The final model for the example in Figure 4 will take the following form:

$$Y = \beta_0 + \beta_1 F_1 + \beta_2 F_2 + \beta_3 F_3 + \beta_4 F_4$$

where Y represents the output of the model, and β_0 is the intercept, β_1 to β_4 are the coefficients of the features F_1 to F_4 respectively. The generated model is then also tested on the test data.

The effect of MLRGP on our dataset is shown in Figure 5. The charts show the average training and test scores of the population by generation for both MLRGP and standard GP. Clearly, MLRGP boosted the fitness scores to a point that it exhibits signs of overfitting on our dataset, as desired in this study.

3 COMPLEXITY CONTROL USING TIME

In this section, we highlight the two approaches that we used to control complexity using our measure of complexity, that is, evaluation time. As detailed later, one method controls the complexity *explicity* while the other does so *implicitly*.

3.1 Explicit Time Control (TC)

The explicit time control [35] leverages the mechanisms of established bloat control techniques but instead of containing size growth it contains the evaluation times. The work in [35] used compared a variety of bloat control techniques but found that for time-control the so-called *Double Tournament* (DT) [23, 25] worked the best. DT has also proven to be a success bloat control technique [25][23][35] that balances accuracy and bloat control.

Double Tournament (DT) [23, 25] runs two rounds of tournaments. The first round runs n probabilistic tournaments (each with a tournament of size 2) to return a set of n individuals. In the tournaments, the smaller individual is selected with a set probability. Then, in the second round, the fittest out of the n individuals is selected. This increase the chances of selecting small and accurate parents and subsequently, increase the chance of producing offspring.

As in [35], to control time we simply replace size with time; the evaluation times of the individuals are measured and saved as an attribute of the individual. Also, we used the recommended problem-independent settings [25] - a probability of 0.7 for choosing the smaller individual in a tournament of size two in the first round. Further, we compare the use of DT to control time with DT to control size (bloat control). This forms a fair basis of comparing the two measures of complexity (size and time).

3.2 Implicit Time Control with Asynchronous Parallel GP

Instead of explicitly discouraging the increase in evaluation times, the so-called *Asynchronous Parallel GP (APGP)* [34] implicitly controls evaluation times by creating a race condition in steady state GP using asynchronous parallel computing. This race means that the models that evaluate quickly can join the steady-state population quicker than their slower counterparts that are still evaluating. Thus, this delayed entry into the breeding population puts the complex models at a disadvantage and the delay is proportional to the computational expense of that model. After all, in natural evolution the population does not wait for it's members to be tested against their environment before creating the new generation. Nevertheless, the traditional GP approach is to stop the evolution and waiting for the population to finish evaluating.

However, once in the population all the models compete based on their fitness; consequently, APGP allows simple and accurate solutions to be produced when and where possible. Therefore, APGP does not explicitly discriminate between model complexity, and the complexity control is gentler than in the explicit time control methods described earlier.

Also [34] shows that APGP improves the training speed of GP; it takes fewer evaluations to match the training accuracy of standard GP and GP with bloat control.

3.2.1 **The APGP Algorithm.** By using the *Steady State* replacement scheme [39], APGP is able to breed and evaluate an offspring, and let it compete for a place in the population immediately. Several of such breed/evaluate operations are allowed to run in parallel and asynchronously on the same population. For example, we may set of 100 of such operations to run at the same time; as soon as one operation completes a new one is initiated. As discussed earlier, the evaluations will complete at different times depending on the complexity of the individual (evaluation times); naturally, all individuals are fairly evaluated by the same means (e.g. using the same dataset). This race to completion means that the less complex individuals (that take less time to evaluate) can potentially get into the population and breed before the more complex (and slow) counterparts.

The initialisation of the APGP algorithm includes the setting of the number of allowed concurrent breeding/evaluations operations and other standard GP parameters like the population size and the total number of offspring to produce (APGP is not generational). Next, the initial population is created and evaluated. Then, the parallel breeding begins that trigger the evaluations; the parallelisation is enable up to the set limit. Immediately after completing evaluation, it is engaged in a tournament. If it is more accurate than the worst of 5 randomly selected individuals in the current population, it replaces that individual and then releases the computing resources. Once the resource is free, a new breed operation begins. As several of these parallel operations are likely to be updating the same population at the same time, a temporary lock is set on an individual that is being replaced to avoid having multiple operations attempting to replace it at the same time.

Speed becomes an advantage only when it is accompanied by high accuracy because only accuracy is considered in the replacement tournament. When the more complex candidates are more accurate than their simpler counterparts, they will succeed and propagate in due course. Thus, APGP does not penalise or exclude complex models. The possibilities within the specific problem determines the simplicity of accurate models; APGP simply increase the changes of gaining accuracy and simplicity where this is possible.

4 EXPERIMENTS

4.1 Other Contending GP Methods

Besides the two time control GP methods detailed in Section 3, other GP methods we also used as follows:

4.1.1 Standard MLRGP (STD). The MLRGP with no complexity control (no bloat control nor time control) is run on the problems for comparison.

4.1.2 MLRGP with Bloat Control (BC). Our choice of bloat control technique is Double Tournament [23, 25]. This has proven to be a success bloat control technique [25][23][35] that balances accuracy and bloat control. The technique has been discussed in Section 3.1; while in this case the technique is to control the size of GP individuals, it will be also be used to control evaluation times in a separate GP methods.

4.1.3 MLRGP with Adjusted R^2 for Fitness. Unlike the well known R^2, the adjusted R^2 [27] is a performance measure used in MLR that balances out the accuracy of a model with the number of features it uses. The original R^2 (also known as the coefficient of determination) is a measure of how well a model explains the response variable. It takes a value between 0 and 1, where 1 is indicates the higher accuracy. Without the Adjusted R^2 MLRGP allows the addition of features that contribute very little or nothing to the fitness of a model. Using the Adjusted R^2 addresses this drawback by taking into account the contribution of additional features. The Adjusted R^2 increases only when the additional features significantly enhances the accuracy of the model and decreases otherwise. Thus, using the adjusted R^2 in the fitness function of MLRGP discourages unnecessary growth in the number of features unless they the features offer a significant improvement. In effect, this offers some bloat control.

For our experiments we used the Wherry/McNemar version [47] of the adjusted R^2 formula, as follows:

$$1 - \frac{(1 - R^2)(n - 1)}{(n - k - 1)},$$

where k denotes the number of features and n denotes the number of instances in the data. To compare different methods, however, we also record the normalised mean square error (NMSE) even where adjusted R^2 is employed in fitness functions.

4.2 Test Problems

Ten widely used and publicly available datasets were selected as test problems. While selecting the test problems, We considered the recommendations in [46] and we checked for overfitting when the enhanced GP is used. The data for Problem 1 – 7 are available at the Penn machine learning benchmark (PMLB) [28]; and the data for Problem 8 – 10 are available at [10]. They are summarised in in Table 1.

4.3 Configurations and Settings

Table 2 summarises the key parameters. Other considerations and settings include the following.

Population Initialisation: We used a Fixed Length Initialisation (FLI) in all the experiment. The motivation is to create a diverse

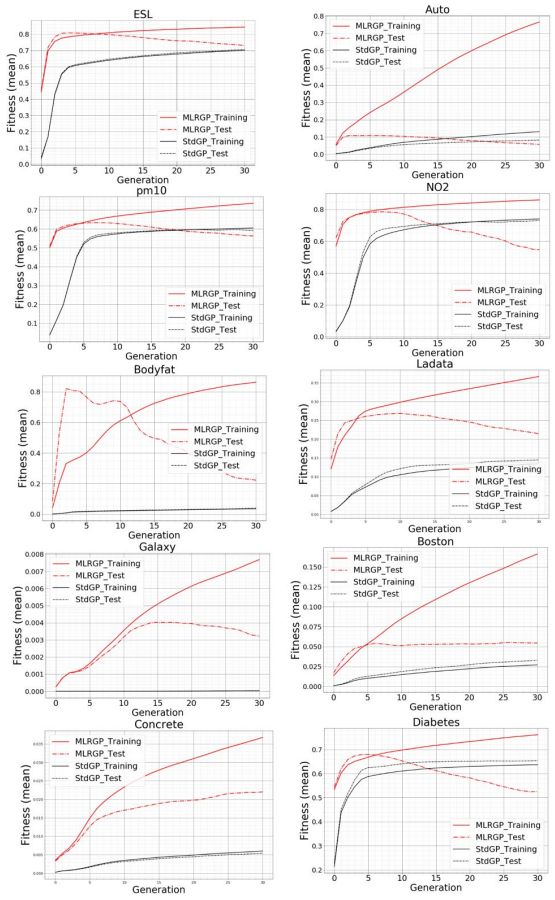


Figure 5: The training and test fitness values of MLRGP and standard GP are compared. MLRGP has improved training to the point of overfitting the data.

ID	Data-Set Name	Variables	Instances
1	027 ESL	3	488
2	207 AutoPrice	14	159
3	522 pm10	6	500
4	547 NO2	6	500
5	560 bodyfat	13	252
6	666 rmftsa ladata	9	508
7	690 Visualizing galaxy	3	323
8	Boston Housing	13	506
9	Concrete Strength	8	1030
10	Diabetes	10	442

Table 1: Overview of Test Problems

populations in terms of the makeup of the individuals ; therefore, as discussed in 1, time control may discriminates between their functional complexities. This initialisation scheme improves the performance of a variety of GP methods [35].

Zero Division error: When zero division errors are encountered during evaluation, the individual with the error is assigned the

Parameter	Setting
No. of runs	30
Size of population	500
Generations	30 (15,000 evaluations)
Random tree/ subtree production	Ramped half-and-half depth = ($min = 1, max = 4$)
Tree depth	max = 17
Operator types	crossover = One point; Point mutation
Operator settings	crossover = 0.9 ; mutation = 0.1
Function set	+, −, ∗, /, sin, cos, neg
Constants (ERC)	|ERC| = 100 (min = 0.05, step: 0.05)
Terminal set	{Input variables} U ERC
Selection	tournament (size = 3)
Replacement	steady state; inverse tournament (size = 5)

Table 2: Summary of Parameters

Method	Test Fitness	Train. Fitness	Size	Evaluat. Times	No of Features
Time-Control Success					
STD	9/10	0/10	10/10	10/10	10/10
BC	7/10	0/10	9/10	10/10	10/10
APGP	6/10	0/10	10/10	10/10	10/10
AR2	6/10	6/10	1/10	3/10	1/10
APGP Success					
STD	9/10	0/10	10/10	10/10	10/10
BC	7/10	6/10	0/10	0/10	0/10
TC	4/10	9/10	0/10	0/10	0/10
AR2	6/10	8/10	0/10	0/10	0/10

Table 3: Summary of the test for significance in difference. The figures show where TC and APGP produced significantly better results, respectively.

worst fitness score to make it uncompetitive. Previous studies [19] have shown that the popular practice of using protected operators can lead to poor generalisation.

Data Sets Splitting: 80% of the data-sets was allocated for training and 20% for testing. The splitting was done using random selection without replacement.

APGP Concurrency: We used threads (number of parallel threads) 100, 250, 500 which represents 20%, 50% and 100% of the population size. Although, threads 250 produced the most competitive results generally, we can explore how this can be improved in the future studies.

Replacement Scheme: As APGP uses a steady-state replacement scheme, we set up all the contending methods to use same.

Fitness function: We used a maximisation fitness functions that used used a fitness score that ranges between 0 and 1. The normalised mean squared error was used as follows: $\frac{1}{1+\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2}}$.

5 RESULTS AND DISCUSSION

First, we examine how the two evaluation time control methods fared against the other methods before examining how they fared against each other. Later, we examine how MLRGP with *Adjusted* R^2 fared against the other methods.

The Mann-Whitney U test was used to determine the significance of the difference in the final populations. While, the result of the tests are are summarised in Table 3, the details (including mean values and p-values) are captured in the colour coded tables in Figure 6 and Figure 7. For fitness, we compare both test fitness and training fitness; and for complexity, we compare the evaluation times, the sizes, and the number of features. The green coloured cells represent results that are favourable to the time control method and where the difference is significant. Contrary, the brown coloured cells are results that are not favourable to the time control method and the difference is significant. The yellow cells are results where the difference was found not to be significant. Figure 6 compares the explicit time control methods against other methods and Figure 7 details the result of comparing the APGP with the other methods.

In terms of test-fitness accuracy, both methods that use evaluation times (TC and APGP) prevailed over the MLRGP with bloat control (BC) and without bloat control (STD). The time control methods had matching results in terms of the number of tests they prevailed; they produced significantly higher test scores in 9 out of 10 tests against STD and 7 out of 10 against BC. However, the two time control methods differ in how they handle complexity; we consider the evaluation times, size and the number of features as measures of complexity. TC produced significantly simpler solutions against BC and STD with only one exception out of the the 60 tests. APGP produced significantly simpler solutions in all tests against STD but more complex solutions than BC in all test.

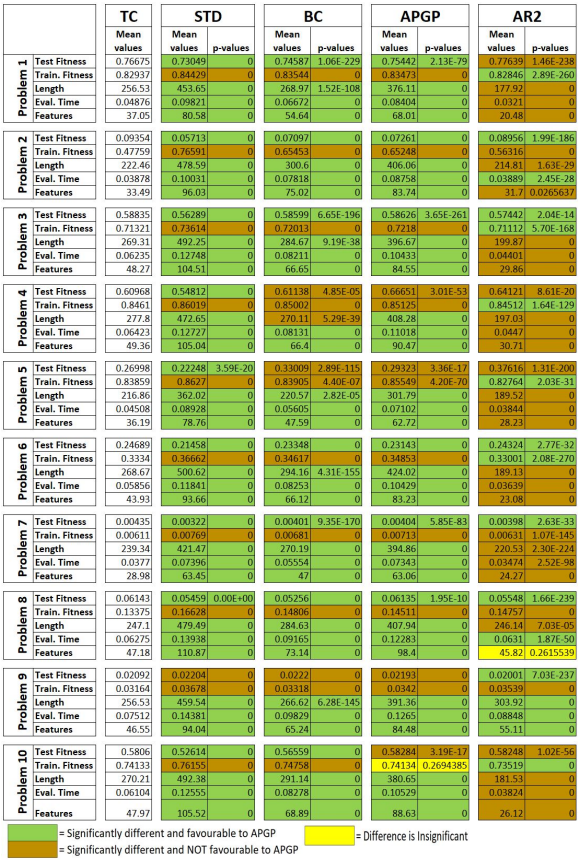
When the two time control techniques are compared, the results are divided. As seen earlier, they are close in terms of test fitness but differ in terms of complexity. TC won 6 out of 10 for test problems but all for complexity. APGP trained better than all but unconstrained MLRGP it prevailed in 9, 6 and 8 instances (out of the 10 each) against TC, BC and AR2, respectively.

MLRGP with *Adjusted* R^2 (AR2) shows effectiveness at containing the number of features. This in turn is reflecting in containing the size and the evaluation times of the individuals. It has produced simpler solutions against the other methods in most of the tests. Also, it is competitive in terms of test fitness accuracy, against the time-control methods; it prevailed in 6 out of 10 against both time control methods.

6 CONCLUSION

To facilitate our study on generalisation in GP, we implemented a version of MLRGP that lead to improvement in fitness to a point of overfitting our data easily. This implementation discovers features; we used the number of features as as another way of comparing the complexity of the solutions produced by the contending methods.

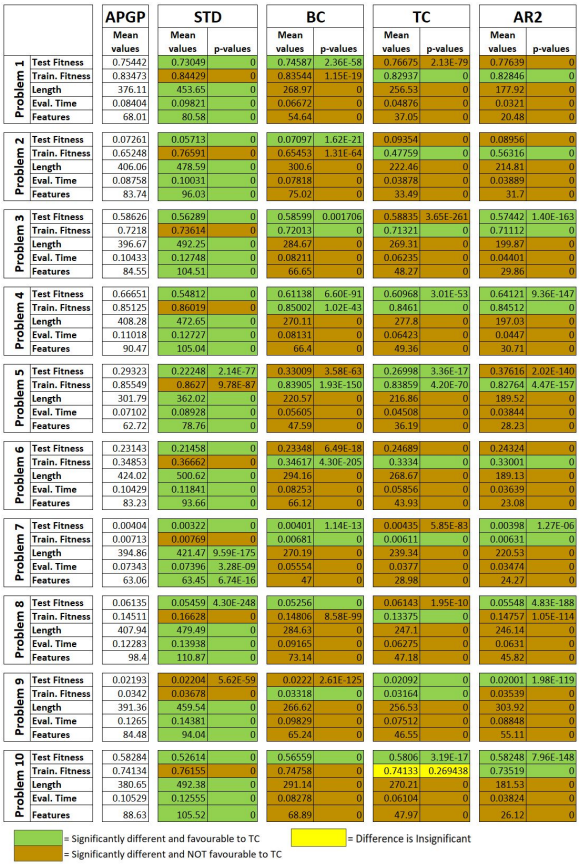
We showed that the evaluation time behaves differently from size. We demonstrated that it can discriminate between the size, the complex of the components, and the number of features of the MLRGP individual.



		TC	STD		BC		APGP		AR2	
		Mean values	Mean values	p-values	Mean values	p-values	Mean values	p-values	Mean values	p-values
Problem 1	Test Fitness	0.76675	0.73049	0	0.74587	1.06E-229	0.75442	2.13E-79	0.77639	1.46E-238
	Train. Fitness	0.82937	0.84429	0	0.83544	0	0.83473	0	0.82846	2.89E-260
	Length	256.53	453.65	0	268.97	1.52E-108	376.11	0	177.92	0
	Eval. Time	0.04876	0.09821	0	0.06672	0	0.08404	0	0.0321	0
	Features	37.05	80.58	0	54.64	0	68.01	0	20.48	0
Problem 2	Test Fitness	0.09354	0.05713	0	0.07097	0	0.07261	0	0.08956	1.99E-186
	Train. Fitness	0.47759	0.76591	0	0.65453	0	0.65248	0	0.56316	0
	Length	222.46	478.59	0	300.6	0	406.06	0	214.81	1.63E-29
	Eval. Time	0.03878	0.10031	0	0.07818	0	0.08758	0	0.03889	2.45E-28
	Features	33.49	96.03	0	75.02	0	83.74	0	31.7	0.0265637
Problem 3	Test Fitness	0.58835	0.56289	0	0.58599	6.65E-196	0.58626	3.65E-261	0.57442	2.04E-14
	Train. Fitness	0.71321	0.73614	0	0.72013	0	0.7218	0	0.71112	5.70E-168
	Length	269.31	492.25	0	284.67	9.19E-38	396.67	0	199.87	0
	Eval. Time	0.06235	0.12748	0	0.08211	0	0.10433	0	0.04401	0
	Features	48.27	104.51	0	66.65	0	84.55	0	29.86	0
Problem 4	Test Fitness	0.60968	0.54812	0	0.61138	4.85E-05	0.66651	3.01E-53	0.64121	8.61E-20
	Train. Fitness	0.8461	0.86019	0	0.85002	0	0.85125	0	0.84512	1.64E-129
	Length	277.8	472.65	0	270.11	5.29E-39	408.28	0	197.03	0
	Eval. Time	0.06423	0.12727	0	0.08131	0	0.11018	0	0.0447	0
	Features	49.36	105.04	0	66.4	0	90.47	0	30.71	0
Problem 5	Test Fitness	0.26998	0.22248	3.59E-20	0.33009	2.89E-115	0.29323	3.36E-17	0.37616	1.31E-200
	Train. Fitness	0.83859	0.8627	0	0.83905	4.40E-07	0.85549	4.20E-70	0.82764	2.03E-31
	Length	216.86	362.02	0	220.57	2.82E-05	301.79	0	189.52	0
	Eval. Time	0.04508	0.08928	0	0.05605	0	0.07102	0	0.03844	0
	Features	36.19	78.76	0	47.59	0	62.72	0	28.23	0
Problem 6	Test Fitness	0.24689	0.21458	0	0.23348	0	0.23143	0	0.24324	2.77E-32
	Train. Fitness	0.3334	0.36662	0	0.34617	0	0.34853	0	0.33001	2.08E-270
	Length	268.67	500.62	0	294.16	4.31E-155	424.02	0	189.13	0
	Eval. Time	0.05856	0.11841	0	0.08253	0	0.10429	0	0.03639	0
	Features	43.93	93.66	0	66.12	0	83.23	0	23.08	0
Problem 7	Test Fitness	0.00435	0.00322	0	0.00401	9.35E-170	0.00404	5.85E-83	0.00398	2.63E-33
	Train. Fitness	0.00611	0.00769	0	0.00681	0	0.00713	0	0.00631	1.07E-145
	Length	239.34	421.47	0	270.19	0	394.86	0	220.53	2.30E-224
	Eval. Time	0.0377	0.07396	0	0.05554	0	0.07343	0	0.03474	2.52E-98
	Features	28.98	63.45	0	47	0	63.06	0	24.27	0
Problem 8	Test Fitness	0.06143	0.05459	0.00E+00	0.05256	0	0.06135	1.95E-10	0.05548	1.66E-239
	Train. Fitness	0.13375	0.16628	0	0.14806	0	0.14511	0	0.14757	0
	Length	247.1	479.49	0	284.63	0	407.94	0	246.14	7.03E-05
	Eval. Time	0.06275	0.13938	0	0.09165	0	0.12283	0	0.0631	1.87E-50
	Features	47.18	110.87	0	73.14	0	98.4	0	45.82	0.2615539
Problem 9	Test Fitness	0.02092	0.02204	0	0.0222	0	0.02193	0	0.02001	7.03E-237
	Train. Fitness	0.03164	0.03678	0	0.03318	0	0.0342	0	0.03539	0
	Length	256.53	459.54	0	266.62	6.28E-145	391.36	0	303.92	0
	Eval. Time	0.07512	0.14381	0	0.09829	0	0.1265	0	0.08848	0
	Features	46.55	94.04	0	65.24	0	84.48	0	55.11	0
Problem 10	Test Fitness	0.5806	0.52614	0	0.56559	0	0.58284	3.19E-17	0.58248	1.02E-56
	Train. Fitness	0.74133	0.76155	0	0.74758	0	0.74134	0.2694385	0.73519	0
	Length	270.21	492.38	0	291.14	0	380.65	0	181.53	0
	Eval. Time	0.06104	0.12555	0	0.08278	0	0.10529	0	0.03824	0
	Features	47.97	105.52	0	68.89	0	88.63	0	26.12	0

= Significantly different and favourable to APGP
= Significantly different and NOT favourable to APGP
= Difference is Insignificant

Figure 6: The result of the test for significance in difference in the fitness fitness values of the final population of Time-Control (TC) against the other GP Methods are detailed.



		APGP	STD		BC		TC		AR2	
		Mean values	Mean values	p-values	Mean values	p-values	Mean values	p-values	Mean values	p-values
Problem 1	Test Fitness	0.75442	0.73049	0	0.74587	2.36E-58	0.76675	2.13E-79	0.77639	0
	Train. Fitness	0.83473	0.84429	0	0.83544	1.15E-19	0.82937	0	0.82846	0
	Length	376.11	453.65	0	268.97	0	256.53	0	177.92	0
	Eval. Time	0.08404	0.09821	0	0.06672	0	0.04876	0	0.0321	0
	Features	68.01	80.58	0	54.64	0	37.05	0	20.48	0
Problem 2	Test Fitness	0.07261	0.05713	0	0.07097	1.62E-21	0.09354	0	0.08956	0
	Train. Fitness	0.65248	0.76591	0	0.65453	1.31E-64	0.47759	0	0.56316	0
	Length	406.06	478.59	0	300.6	0	222.46	0	214.81	0
	Eval. Time	0.08758	0.10031	0	0.07818	0	0.03878	0	0.03889	0
	Features	83.74	96.03	0	75.02	0	33.49	0	31.7	0
Problem 3	Test Fitness	0.58626	0.56289	0	0.58599	0.001706	0.58835	3.65E-261	0.57442	1.40E-163
	Train. Fitness	0.7218	0.73614	0	0.72013	0	0.71321	0	0.71112	0
	Length	396.67	492.25	0	284.67	0	269.31	0	199.87	0
	Eval. Time	0.10433	0.12748	0	0.08211	0	0.06235	0	0.04401	0
	Features	84.55	104.51	0	66.65	0	48.27	0	29.86	0
Problem 4	Test Fitness	0.66651	0.54812	0	0.61138	6.60E-91	0.60968	3.01E-53	0.64121	9.36E-147
	Train. Fitness	0.85125	0.86019	0	0.85002	1.02E-43	0.8461	0	0.84512	0
	Length	408.28	472.65	0	270.11	0	277.8	0	197.03	0
	Eval. Time	0.11018	0.12727	0	0.08131	0	0.06423	0	0.0447	0
	Features	90.47	105.04	0	66.4	0	49.36	0	30.71	0
Problem 5	Test Fitness	0.29323	0.22248	2.14E-77	0.33009	3.58E-63	0.26998	3.36E-17	0.37616	2.02E-140
	Train. Fitness	0.85549	0.8627	9.78E-87	0.83905	1.93E-150	0.83859	4.20E-70	0.82764	4.47E-157
	Length	301.79	362.02	0	220.57	0	216.86	0	189.52	0
	Eval. Time	0.07102	0.08928	0	0.05605	0	0.04508	0	0.03844	0
	Features	62.72	78.76	0	47.59	0	36.19	0	28.23	0
Problem 6	Test Fitness	0.23143	0.21458	0	0.23348	6.49E-18	0.24689	0	0.24324	0
	Train. Fitness	0.34853	0.36662	0	0.34617	4.30E-205	0.3334	0	0.33001	0
	Length	424.02	500.62	0	294.16	0	268.67	0	189.13	0
	Eval. Time	0.10429	0.11841	0	0.08253	0	0.05856	0	0.03639	0
	Features	83.23	93.66	0	66.12	0	43.93	0	23.08	0
Problem 7	Test Fitness	0.00404	0.00322	0	0.00401	1.14E-13	0.00435	5.85E-83	0.00398	1.27E-06
	Train. Fitness	0.00713	0.00769	0	0.00681	0	0.00611	0	0.00631	0
	Length	394.86	421.47	9.59E-175	270.19	0	239.34	0	220.53	0
	Eval. Time	0.07343	0.07396	3.28E-09	0.05554	0	0.0377	0	0.03474	0
	Features	63.06	63.45	6.74E-16	47	0	28.98	0	24.27	0
Problem 8	Test Fitness	0.06135	0.05459	4.30E-248	0.05256	0	0.06143	1.95E-10	0.05548	4.83E-188
	Train. Fitness	0.14511	0.16628	0	0.14806	8.58E-99	0.13375	0	0.14757	1.05E-114
	Length	407.94	479.49	0	284.63	0	247.1	0	246.14	0
	Eval. Time	0.12283	0.13938	0	0.09165	0	0.06275	0	0.0631	0
	Features	98.4	110.87	0	73.14	0	47.18	0	45.82	0
Problem 9	Test Fitness	0.02193	0.02204	5.62E-59	0.0222	2.61E-125	0.02092	0	0.02001	1.98E-119
	Train. Fitness	0.0342	0.03678	0	0.03318	0	0.03164	0	0.03539	0
	Length	391.36	459.54	0	266.62	0	256.53	0	303.92	0
	Eval. Time	0.1265	0.14381	0	0.09829	0	0.07512	0	0.08848	0
	Features	84.48	94.04	0	65.24	0	46.55	0	55.11	0
Problem 10	Test Fitness	0.58284	0.52614	0	0.56559	0	0.5806	3.19E-17	0.58248	7.96E-148
	Train. Fitness	0.74134	0.76155	0	0.74758	0	0.74133	0.269438	0.73519	0
	Length	380.65	492.38	0	291.14	0	270.21	0	181.53	0
	Eval. Time	0.10529	0.12555	0	0.08278	0	0.06104	0	0.03824	0
	Features	88.63	105.52	0	68.89	0	47.97	0	26.12	0

= Significantly different and favourable to TC
= Significantly different and NOT favourable to TC
= Difference is Insignificant

Figure 7: The result of the test for significance in difference in the fitness values of the final population of APGP against the other GP Methods are detailed.

The results reasserts that using the evaluation time leads to better generalisation. Time control methods (both TC and BC) are producing better test score than bloat control (BC) and standard MLRGP. This is despite that fact that TC and BC used the same techniques to control time and size, respectively. Also, individually, the time control techniques produced similar results against BC and STD. This indicates that the evaluation time is the determining factor in the improvement. Further as anticipated, the APGP showed a less aggressive complexity control than TC while showing better training.

The MLRGP with Adjusted R^2 (AR2) effectively controls bloat by simply suppressing the unnecessary growth in the number of features in an MLRGP tree. This made it competitive against the time control techniques (TC and APGP) and it prevailed against BC and STD. However, AR2 is only applicable where GP is working with features.

Overall, this study highlights the feasibility and merits of using the evaluation time to improve generalisation and as a broad applicable complexity control measure.

REFERENCES

[1] Ignacio Arnaldo, Krzysztof Krawiec, and Una-May O'Reilly. 2014. Multiple regression genetic programming. In *Genetic and Evolutionary Computation Conference, GECCO '14, Vancouver, BC, Canada, July 12-16, 2014*, Dirk V. Arnold (Ed.). ACM, 879–886. http://dl.acm.org/citation.cfm?id=2576768

[2] Raja Muhammad Atif Azad. 2003. *A Position Independent Representation for Evolutionary Automatic Programming Algorithms - The Chorus System.* Ph.D. Dissertation. University of Limerick, Ireland. http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/azad_thesis.ps.gz

[3] R. Muhammad Atif Azad and Conor Ryan. 2011. Variance based selection to improve test set performance in genetic programming. In *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*, Natalio Krasnogor et al (Ed.). ACM, Dublin, Ireland, 1315–1322. https://doi.org/10.1145/2001576.2001754

[4] R. Muhammad Atif Azad and Conor Ryan. 2014. The Best Things Don't Always Come in Small Packages: Constant Creation in Grammatical Evolution. In *17th European Conference on Genetic Programming (LNCS)*, Miguel Nicolau, Krzysztof Krawiec, Malcolm I. Heywood, Mauro Castelli, Pablo Garcia-Sanchez, Juan J. Merelo, Victor M. Rivas Santos, and Kevin Sim (Eds.), Vol. 8599. Springer, Granada, Spain, 186–197. https://doi.org/10.1007/978-3-662-44303-3_16

[5] Raja Muhammad Atif Azad and Conor Ryan. 2014. A Simple Approach to Lifetime Learning in Genetic Programming based Symbolic Regression. *Evolutionary Computation* 22, 2 (June 2014), 287–317. https://doi.org/10.1162/EVCO_a_00111

[6] Raja Muhammad Atif Azad and Conor Ryan. 2014. A Simple Approach to Lifetime Learning in Genetic Programming-Based Symbolic Regression. *Evolutionary Computation* 22, 2 (2014), 287–317. https://doi.org/10.1162/EVCO_a_00111

[7] Qi Chen, Mengjie Zhang, and Bing Xue. 2019. Structural Risk Minimisation-Driven Genetic Programming for Enhancing Generalisation in Symbolic Regression. *IEEE Transactions on Evolutionary Computation* 23, 4 (Aug. 2019), 703–717.

https://doi.org/10.1109/TEVC.2018.2881392

[8] Gopinath Chennupati, Raja Muhammad Atif Azad, and Conor Ryan. 2015. Performance Optimization of Multi-Core Grammatical Evolution Generated Parallel Recursive Programs. In *GECCO '15: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, Sara Silva et al (Ed.). ACM, Madrid, Spain, 1007–1014. https://doi.org/10.1145/2739480.2754746

[9] Francisco Fernández de Vega, Gustavo Olague, Daniel Lanza, Wolfgang Banzhaf, Erik Goodman, Jose Menendez-Clavijo, Axel Martinez, et al. 2020. Time and Individual Duration in Genetic Programming. *IEEE Access* 8 (2020), 38692–38713.

[10] Dheeru Dua and Efi Karra Taniskidou. 2017. UCI Machine Learning Repository. (2017). http://archive.ics.uci.edu/ml

[11] I De Falco, Aniello Iazzetta, Ernesto Tarantino, A Delia Cioppa, and Giuseppe Trautteur. 2000. A Kolmogorov Complexity-based Genetic Programming tool for string compression. In *Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation*. Morgan Kaufmann Publishers Inc., Morgan Kaufmann, Las Vegas, Nevada, USA, 427–434.

[12] Amir Hossein Gandomi and Amir Hossein Alavi. 2012. A new multi-gene genetic programming approach to nonlinear system modeling. Part I: materials and structural engineering problems. *Neural Comput. Appl* 21, 1 (2012), 171–187.

[13] Peter Griinwald. 2005. Introducing the minimum description length principle. *Advances in minimum description length: Theory and applications* 3 (2005), 3–22.

[14] Nguyen Xuan Hoai, Robert I McKay, and Daryl Essam. 2006. Representation and structural difficulty in genetic programming. *IEEE Transactions on evolutionary computation* 10, 2 (2006), 157–166.

[15] Ting Hu, Joshua Payne, Wolfgang Banzhaf, and Jason Moore. 2012. Evolutionary dynamics on multiple scales: a quantitative analysis of the interplay between genotype, phenotype, and fitness in linear genetic programming. *Genetic Programming and Evolvable Machines* 13, 3 (Sept. 2012), 305–337. https://doi.org/10.1007/s10710-012-9159-4 Special issue on selected papers from the 2011 European conference on genetic programming.

[16] Hitoshi Iba, Hugo de Garis, and Taisuke Sato. 1994. Genetic Programming Using a Minimum Description Length Principle. In *Advances in Genetic Programming*, Kenneth E. Kinnear, Jr. (Ed.). MIT Press, Cambridge, MA, USA, Chapter 12, 265–284. http://cognet.mit.edu/sites/default/files/books/9780262277181/pdfs/9780262277181_chap12.pdf

[17] Vijay Ingalalli, Sara Silva, Mauro Castelli, and Leonardo Vanneschi. 2014. A Multi-dimensional Genetic Programming Approach for Multi-class Classification Problems. In *17th European Conference on Genetic Programming (LNCS)*, Miguel Nicolau, Krzysztof Krawiec, Malcolm I. Heywood, Mauro Castelli, Pablo Garcia-Sanchez, Juan J. Merelo, Victor M. Rivas Santos, and Kevin Sim (Eds.), Vol. 8599. Springer, Granada, Spain, 48–60. https://doi.org/10.1007/978-3-662-44303-3_5

[18] Maarten Keijzer. 2003. Improving Symbolic Regression with Interval Arithmetic and Linear Scaling. In *Genetic Programming, Proceedings of EuroGP'2003 (LNCS)*, Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa (Eds.), Vol. 2610. Springer-Verlag, Essex, 70–82. https://doi.org/10.1007/3-540-36599-0_7

[19] Maarten Keijzer. 2003. Improving symbolic regression with interval arithmetic and linear scaling. In *European Conference on Genetic Programming*. EuroGP, Springer, Essex, UK, 70–82.

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

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

[22] Sanjeev R Kulkarni and Gilbert Harman. 2011. Statistical learning theory: a tutorial. *Wiley Interdisciplinary Reviews: Computational Statistics* 3, 6 (2011), 543–556.

[23] Sean Luke and Liviu Panait. 2002. Fighting Bloat with Nonparametric Parsimony Pressure. In *Parallel Problem Solving from Nature - PPSN VII (Lecture Notes in Computer Science, LNCS)*, Juan J. Merelo-Guervos, Panagiotis Adamidis, Hans-Georg Beyer, Jose-Luis Fernandez-Villacanas, and Hans-Paul Schwefel (Eds.). Springer-Verlag, Granada, Spain, 411–421. https://doi.org/10.1007/3-540-45712-7_40

[24] Sean Luke and Liviu Panait. 2006. A Comparison of Bloat Control Methods for Genetic Programming. *Evolutionary Computation* 14, 3 (Sept. 2006), 309–344. https://doi.org/10.1162/evco.2006.14.3.309

[25] Sean Luke and Liviu Panait. 2006. A Comparison of Bloat Control Methods for Genetic Programming. *Evolutionary Computation* 14, 3 (Nov. 2006), 309–344. https://doi.org/10.1162/evco.2006.14.3.309

[26] Yi Mei, Su Nguyen, and Mengjie Zhang. 2017. Evolving Time-Invariant Dispatching Rules in Job Shop Scheduling with Genetic Programming. In *EuroGP 2017: Proceedings of the 20th European Conference on Genetic Programming (LNCS)*, Mauro Castelli, James McDermott, and Lukas Sekanina (Eds.), Vol. 10196. Springer Verlag, Amsterdam, 147–163. https://doi.org/10.1007/978-3-319-55696-3_10

[27] Jeremy Miles. 2014. R Squared, Adjusted R Squared. (2014), 0 pages. https://doi.org/10.1002/9781118445112.stat06627 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118445112.stat06627

[28] Randal S Olson, William La Cava, Patryk Orzechowski, Ryan J Urbanowicz, and Jason H Moore. 2017. PMLB: a large benchmark suite for machine learning evaluation and comparison. *BioData mining* 10, 1 (2017), 1–13.

[29] Mihai Oltean and Crina Grosan. 2004. A Comparison of Several Linear Genetic Programming Techniques. *Complex Systems* 14, 4 (2004), 285–313. http://www.cs.ubbcluj.ro/~cgrosan/030409_edited.pdf

[30] Gregory Paris, Denis Robilliard, and Cyril Fonlupt. 2003. Exploring Overfitting in Genetic Programming. In *Evolution Artificielle, 6th International Conference (Lecture Notes in Computer Science)*, Pierre Liardet, Pierre Collet, Cyril Fonlupt, Evelyne Lutton, and Marc Schoenauer (Eds.), Vol. 2936. Springer, Marseilles, France, 267–277. https://doi.org/10.1007/b96080 Revised Selected Papers.

[31] Riccardo Poli. 2003. A Simple but Theoretically-motivated Method to Control Bloat in Genetic Programming. In *Genetic Programming, Proceedings of EuroGP'2003 (LNCS)*, Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa (Eds.), Vol. 2610. Springer-Verlag, Essex, 204–217. https://doi.org/10.1007/3-540-36599-0_19

[32] Conor Ryan, Michael O'Neill, and J. J. Collins (Eds.). 2018. *Handbook of Grammatical Evolution*. Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-319-78717-6

[33] Aliyu Sani Sambo, R. Muhammad Atif Azad, Yevgeniya Kovalchuk, Vivek Padmanaabhan Indramohan, and Hanifa Shah. 2020. Feature Engineering for Improving Robustness of Crossover in Symbolic Regression. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion (GECCO '20)*. Association for Computing Machinery, internet, 249–250. https://doi.org/10.1145/3377929.3390078

[34] Aliyu Sani Sambo, R. Muhammad Atif Azad, Yevgeniya Kovalchuk, Vivek Padmanaabhan Indramohan, and Hanifa Shah. 2020. Leveraging Asynchronous Parallel Computing to Produce Simple Genetic Programming Computational Models. In *The 35th ACM/SIGAPP Symposium On Applied Computing*, Federico Divina and Miguel Garcia Torres (Eds.). ACM, Brno, Czech Republic, 521–528. https://doi.org/10.1145/3341105.3373921

[35] Aliyu Sani Sambo, R. Muhammad Atif Azad, Yevgeniya Kovalchuk, Vivek Padmanaabhan Indramohan, and Hanifa Shah. 2020. Time Control or Size Control? Reducing Complexity and Improving Accuracy of Genetic Programming Models. In *Genetic Programming - 23rd European Conference, EuroGP 2020, Held as Part of EvoStar 2020, Seville, Spain, April 15-17, 2020, Proceedings (Lecture Notes in Computer Science)*, Ting Hu, Nuno Lourenço, Eric Medvet, and Federico Divina (Eds.), Vol. 12101. Springer, Berlin, Heidelberg, 195–210. https://doi.org/10.1007/978-3-030-44094-7_13

[36] Cameron Simpson, Jim Jewett, Stephen Turnbull, and Victor Stinner. [n. d.]. PEP 418: Add monotonic time, performance counter, and process time functions. Website. ([n. d.]). https://www.python.org/dev/peps/pep-0418/

[37] Terence Soule, James A. Foster, and John Dickinson. 1996. Code Growth in Genetic Programming. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo (Eds.). MIT Press, Stanford University, CA, USA, 215–223. http://cognet.mit.edu/sites/default/files/books/9780262315876/pdfs/9780262315876_chap26.pdf

[38] Lee Spector and Alan Robinson. 2002. Genetic Programming and Autoconstructive Evolution with the Push Programming Language. *Genetic Programming and Evolvable Machines* 3, 1 (March 2002), 7–40. https://doi.org/10.1023/A:1014538503543

[39] Gilbert Syswerda. 1991. A study of reproduction in generational and steady-state genetic algorithms. In *Foundations of genetic algorithms*. Vol. 1. Elsevier, Amsterdam, 94–101.

[40] Leonardo Vanneschi, Mauro Castelli, and Sara Silva. 2010. Measuring bloat, overfitting and functional complexity in genetic programming. In *GECCO '10: Proceedings of the 12th annual conference on Genetic and evolutionary computation*, Juergen Branke et al (Ed.). ACM, Portland, Oregon, USA, 877–884. https://doi.org/10.1145/1830483.1830643

[41] Vladimir Vapnik. 1998. *Statistical learning theory. 1998*. Vol. 3. Wiley, New York.

[42] V. Vapnik. 2013. *The Nature of Statistical Learning Theory*. Springer New York.

[43] Vladimir Naumovich Vapnik. 1998. *Statistical learning theory*. Wiley, New York. OCLC: 845016043.

[44] Ekaterina J. Vladislavleva, Guido F. Smits, and Dick den Hertog. 2009. Order of Nonlinearity as a Complexity Measure for Models Generated by Symbolic Regression via Pareto Genetic Programming. *IEEE Transactions on Evolutionary Computation* 13, 2 (April 2009), 333–349. https://doi.org/10.1109/TEVC.2008.926486

[45] James Alfred Walker and Julian Francis Miller. 2008. The Automatic Acquisition, Evolution and Reuse of Modules in Cartesian Genetic Programming. *IEEE Transactions on Evolutionary Computation* 12, 4 (Aug. 2008), 397–417. https://doi.org/10.1109/TEVC.2007.903549

[46] David R. White, James McDermott, Mauro Castelli, Luca Manzoni, Brian W. Goldman, Gabriel Kronberger, Wojciech Jaskowski, Una-May O'Reilly, and Sean Luke. 2013. Better GP benchmarks: community survey results and proposals. *Genetic Programming and Evolvable Machines* 14, 1 (March 2013), 3–29. https://doi.org/10.1007/s10710-012-9177-2

[47] Ping Yin and Xitao Fan. 2001. Estimating R 2 shrinkage in multiple regression: A comparison of different analytical methods. *The Journal of Experimental Education* 69, 2 (2001), 203–224.