Artificial Intelligence 174 (2010) 1254–1276

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

# Practical performance models of algorithms in evolutionary program induction and other domains

Mario Graff*, Riccardo Poli

*School of Computer Science and Electronic Engineering, University of Essex, Colchester, CO4 3SQ, UK*

ARTICLE INFO

*Article history:*
Received 22 October 2008
Received in revised form 18 July 2010
Accepted 19 July 2010
Available online 23 July 2010

*Keywords:*
Evolution algorithms
Program induction
Performance prediction
Algorithm taxonomies
Algorithm selection problem

ABSTRACT

Evolutionary computation techniques have seen a considerable popularity as problem solving and optimisation tools in recent years. Theoreticians have developed a variety of both exact and approximate models for evolutionary program induction algorithms. However, these models are often criticised for being only applicable to simplistic problems or algorithms with unrealistic parameters. In this paper, we start rectifying this situation in relation to what matters the most to practitioners and users of program induction systems: performance. That is, we introduce a simple and practical model for the performance of program-induction algorithms. To test our approach, we consider two important classes of problems — symbolic regression and Boolean function induction — and we model different versions of genetic programming, gene expression programming and stochastic iterated hill climbing in program space. We illustrate the generality of our technique by also accurately modelling the performance of a training algorithm for artificial neural networks and two heuristics for the off-line bin packing problem.

We show that our models, besides performing accurate predictions, can help in the analysis and comparison of different algorithms and/or algorithms with different parameters setting. We illustrate this via the automatic construction of a taxonomy for the stochastic program-induction algorithms considered in this study. The taxonomy reveals important features of these algorithms from the performance point of view, which are not detected by ordinary experimentation.



## 1. Introduction

Evolutionary Algorithms (EAs) are popular forms of search and optimisation [1–6]. Their invention dates back many decades (e.g., see [7]). So, one might imagine that, by now, we should have a full theoretical understanding of their operations and a rich set of theoretically-sound guidelines for their parametrisation and customisation. However, this is not the case.

Despite the simplicity of EAs, sound theoretical models of EAs and precise mathematical results have been scarce and hard to obtain, often emerging many years after the proposal of the original algorithm (e.g., see [8–18]). A key reason for this is that each algorithm, representation, set of genetic operators and, often, fitness function requires a different theoretical model. In addition, the randomness, non-linearities and immense number of degrees of freedom present in a typical EA make life very hard for theoreticians.

This applies also to techniques for the automatic evolution of computer programs, or Evolutionary Program-induction Algorithms (EPAs), including Genetic Programming (GP) [6,15,19], Cartesian GP (CGP) [20], Grammatical Evolution (GE)

* Corresponding author.
*E-mail addresses:* mgraff@essex.ac.uk (M. Graff), rpoli@essex.ac.uk (R. Poli).

0004-3702/$ – see front matter 
doi:10.1016/j.artint.2010.07.005

[21] and Gene Expression Programming (GEP) [22] among others. Our theoretical understanding of EPAs has been even slower to develop than for other EAs chiefly because of the objective difficulty of modelling stochastic searchers in infinite spaces (programs have unbounded size) where search operators can dynamically change the dimension and structure of the solutions being explored, as it is the case for most EPAs. So, despite recent successes in developing solid theory for GP and related EPAs (e.g., see [15,17,23] and the review in [19]), there is a growing gap between EPA theory and practice.

Often theoretical studies and models of EAs are criticised for not being easily applicable to realistic situations (e.g., see [24]). One reason for this is that producing a comprehensive theory for complex adaptive systems such as EAs is objectively hard and slow, as we mentioned earlier. Another reason is that, sometimes, theoreticians focus on approaches and problems that are too distant from practice. So, despite the proven effectiveness of EAs and EPAs (see for example [19]), there is an urgent need for a theory that can clarify the applicability of different types of algorithms to particular problems, provide design guidelines and, thereby, avoid the current, very time-consuming, practice of hand-tuning algorithms, parameters and operators.

This paper attempts to rectify this situation by proposing a practical model of EPAs. The model, by design, does not capture all the characteristics of an algorithm nor models it exactly (which is extremely difficult). Instead, it focuses on what matters the most to practitioners, the *performance* of EAs in realistic problems, accepting the fact that, in practice, modelling performance cannot be done exactly. This model will allow us to give answers to questions such as: How likely is it that a particular algorithm will solve a particular problem of interest? What fitness should we expect to find at the end of a run? What's the best algorithm to solve a problem or a class of problems? Since no alternative model of EPA performance is available at present, the only alternative is to seek answers to these questions by direct empirical experimentation!

Although our approach was initially aimed at modelling EPAs, it can easily be extended beyond program induction by stochastic search to capture the characteristics of other forms of search and problem solving. To illustrate this we will also model the performance of two heuristics for the off-line bin packing problem and one learning algorithm for feed-forward neural networks.

Our models are related to techniques used to solve the algorithm selection problem [25] (i.e., the problem of deciding which tool to choose to solve a problem out of a set of available tools) and, in particular, to the modelling techniques used in algorithm portfolios [26–35] (i.e., collections of algorithms that are run in parallel or in sequence to solve a problem). The methodology presented here is complementary to (but competitive with) such approaches, as we will illustrate through in the creation of effective portfolios of program induction algorithms: an area where no algorithm selection technique had been tested before.

Our models can also be used beyond the pure prediction of performance. For example, they enable the analysis of the similarities and differences between algorithms in relation to performance. To illustrate this, from a collection of models of different algorithms (or the same algorithms but with different parameters) we will obtain a meaningful and informative taxonomy of evolutionary and stochastic program-induction algorithms, with a completely automatic process.

The paper is organised as follows. In Section 2 we review related theoretical work in the field of EAs. In Section 3 we describe our performance model, how we arrived at it and the methodology used to instantiate it. Section 4 presents the problems used to test the approach, while Section 5 describes the systems and parameter settings used in the experimentation. Experimental results that corroborate the validity of the models' predictions are presented in Section 6. Section 7 looks at the algorithm selection problem surveying relevant literature and applying our models to the creation of two algorithm portfolios for program-induction problems. The similarities and differences with other approaches are also discussed. Applications of our models in the comparison and categorisation of algorithms are discussed in Section 8. Some conclusions and possible directions for future work are given in Section 9.

## 2. Related work

Our work is related to the problem of understanding what makes a problem easy or hard for EAs. Problem-difficulty studies in EAs focused initially on the building-block hypothesis for Genetic Algorithms (GAs) [2] and the related notion of deception [36]. The approach consisted in constructing artificial fitness functions that, based on certain *a priori* assumptions, would be easy or hard for GAs. This produced useful results but also some puzzling counter examples [37].

The notion of *fitness landscape*, originally proposed in [38], underlies many recent approaches to problem difficulty. It is clear, for example, that a smooth landscape with a single optimum will be relatively easy to search for many algorithms, while a very rugged landscape, with many local optima, may be more problematic [39,40]. However, the graphical visualisation of fitness landscapes is rarely possible given the size of typical search spaces. So, one really needs to condense useful information on fitness landscapes into one or a few numeric descriptors.

In [41], Jones introduced one such descriptor of problem difficulty for GAs: the *fitness distance correlation* (*fdc*). The study of *fdc* has been extended to GP [42–45]. These studies show that *fdc* is often a reliable indicator of problem hardness. However, it has one big flaw: it requires the optimal solution(s) to be known beforehand. This prevents the use of *fdc* to estimate problem difficulty in practical applications. A measure that does not suffer from this problem, the *negative slope coefficient* (*nsc*), has recently been proposed [46]. This is based in the concept of fitness cloud (a scatter plot of parent/offspring fitness pairs). The *nsc* uses the idea of first dividing the cloud into a certain number of bins along the parent-fitness axis, then computing the mean offspring fitness for each bin, and finally analysing the changes in slope between adjacent bins in the resulting histogram. The *nsc* has been shown to be a reliable measure in a number of different

benchmark problems in GP [46–48]. A slightly modified version of *nsc* called *fitness proportional nsc* has also given good results in GAs [49].

While *fdc* and *nsc* (and other measures of problem difficulty) provide reasonable indications of *whether* a problem is hard or easy, they do not really give a direct estimation of *how hard* or easy a problem is in relation to any particular performance measure, such as the success rate of an EA or the expected end-of-run fitness. To the best of our knowledge, no approach has ever been proposed that achieves this for EPAs, and this paper attempts to fill precisely this important gap in our knowledge. However, there are a small number of approaches (including one that has been a source of inspiration for the work presented here) that have achieved a good degree of success at predicting actual performance in other areas of EAs. So, we briefly review them below.

Precise bounds on the expected run time for a variety of EAs can be obtained by using computational complexity techniques [50–56]. Typically, this has been done only for specific classes of functions although there are some exceptions where run-time bounds have been derived for more general combinatorial optimisation problems [57–62].

At the opposite end of the generality spectrum is the No Free Lunch (NFL) theorem [63], which shows that averaged over all possible algorithms, all problems have the same difficulty, and, averaged over all possible problems, all algorithms have the same efficiency. The implications and applicability of NFL have been clarified (e.g., see [64–67]). While NLF-type results are very important as they limit what can or cannot be achieved by a search algorithm performance-wise, they can only provide indications of the performance of an algorithm only if the class of problems is closed under permutation [66]. However, we know that for function and program induction only very artificial problem classes are closed under permutation [68–70].

The situation is not much better for continuous optimisation, where we have a reasonably clear understanding of behaviour and performance essentially only for Evolutionary Strategies [71] applied to particularly simple functions (e.g., spheres and ridges). Markov chain models with continuous state spaces can be defined and general results have been obtained using them [72]. However, the complexity of the calculations involved makes them impractical for the analysis of the performance of continuous optimisers. These can also be studied using *discrete* Markov chain models that can approximate them on arbitrary continuous problems to any precision [73]. While this is promising, it is not clear how to extend the work in [72,73] to EPAs, given that they explore either a space of discrete structures or a hybrid discrete space with continuous subspaces (corresponding to real-valued numerical constants) embedded within it. In principle, it would be possible to apply Markov chains to predict the performance of EPAs (without real-valued constants) using some recently developed models [18], but these models are immense, making their application to practical problems effectively impossible.

The simple approach in [74], where the performance of a GA was modelled with surprisingly reliable results, has particularly inspired the work presented in this paper. The idea there was that, when selection is based on comparing the fitness of different solutions (as is often the case), the performance of an EA really only depends on *relative* fitness values. The fitness function can, therefore, be re-represented using a comparison matrix. The matrix represents the outcomes of all the possible comparisons between pairs of solutions that the selection mechanism might need to perform. So, the value of the element $(i, j)$ in the matrix is the sign of $f(i) - f(j)$, $f$ being the fitness function. Because of the matrix is skew-symmetric, all the relevant information is stored in its upper triangle. By reading the elements of the upper triangle one by one, we obtain a vector $v = (v_1, v_2, \ldots)$, called an *information landscape*, that represents all the information the EA can ever need about $f$ to perform its search. Thus, the performance of an EA on a particular problem can only be a function of $v$. Clearly, this function is expected to be non-linear, quite complex, and practically impossible to derive from first principles for any algorithm of any complexity. However, [74] obtained good results by modelling the performance of a GA using the simple function

$$P(v) \approx a_0 + \sum a_i v_i, \tag{1}$$

where $a_i$ are coefficients. These were found by applying the least squares method to a training set containing a sufficiently large set of $(v, P(v))$ tuples obtained by running the GA on a variety of problems and recording the associated performance.

In principle, any comparison-based algorithm can be modelled in this way. Also, potentially users are free to choose any performance measure. However, the technique does not scale well with the size, $n$, of the search space, the number of elements in $v$ being $\frac{n(n-1)}{2}$. Thus, to uniquely identify the coefficients $a_i$, one needs a training set containing at least $\frac{n(n-1)}{2} + 1$ problem/performance pairs. Also, because of the stochasticity of GAs, performance evaluation on a problem typically requires gathering statistics over multiple runs. Thus, the construction of a training set suitable for the application of the method is practically impossible, except for very small search spaces.[1]

[1] As one of the reviewers observed, if $n$ is so small that enumerating $n$ objects is feasible, the optimisation problem is not interesting since it can be solved in a trivial way. This implies that for very small problems, modelling the performance of algorithms other than the simplest, say random search or enumeration, is essentially an academic exercise, since complex algorithms are unlikely to be used in such trivial cases. However, when the search space is large, as the search spaces we consider in this paper, performance models may become very important.

## 3. Modelling EPA performance

In this section we describe our performance model and how we arrived at it. We start by considering the applicability of information landscapes to program spaces.

### 3.1. Information landscapes for search in program spaces?

Inspired by the information-landscape idea of [74] summarised in Section 2, we tried to model the performance of a GP system using the $v$ vector re-representation of the fitness function and Eq. (1). Unfortunately, upon a detailed study, we found that there are problems that limit the applicability of the original information landscape modelling technique to EPAs.

The first issue is the scalability of the information-landscape approach, which in program-induction is particularly problematic because of the relatively slow fitness evaluations associated with typical induction and regression problems. In fact, the information landscape technique could not even be applied to program induction because, at least in principle, the size of program search spaces is infinite. So, the required $v$ vector would be infinitely dimensional, and so, an infinitely large training set would be needed to determine its associated coefficients $a_i$. Of course, in practice there is always some upper bound on the size of the programs one is interested in exploring. Nonetheless, the problem remains serious because for typical primitive sets, the number of distinct programs up to a certain size grows exponentially with the size and the number of coefficients that need identifying grows like the square of that number. This makes it difficult to use the information landscape approach even for the smallest program spaces.

The size of the training set is not the only problem. Another problem is that, in most primitive sets, there are symmetries which imply that two syntactically different programs may, in fact, present the same functionality. For example, within the search space generated by $\{x, y, \sqrt{}, +, \times\}$, the expressions $p_1 = \sqrt{x \times y}$ and $p_2 = \sqrt{y \times x}$ are functionally indistinguishable. If fitness is computed (as usual) based on the behaviour of programs, syntactically distinct programs with identical behaviours will always have identical fitness irrespective of the problem. This translates into constraints between elements of the $v$ vector re-representation of fitness. For example, if $p_3 = x$ and $v_j$ represents the comparison between the fitness of programs $p_1$ and $p_3$ while $v_k$ represents the comparison between programs $p_2$ and $p_3$, it will always be the case that $v_j \equiv v_k$. As a result any choice of the coefficients $a_j$ and $a_k$ such that $a_j + a_k = \text{constant}$ produces a model of identical quality. This makes the problem of identifying such coefficients via ordinary linear regression generally ill-posed [75] in the sense that it does not have a unique solution. More generally, the coefficients in Eq. (1) associated with elements in the $v$ vector that remain identical across problems cannot univocally be determined.[2]

All this suggests that the information landscape approach, as originally formulated, is unsuitable to model EPAs. Some modification of the approach are thus necessary to overcome its limitations. Multiple alternatives are possible. In the next sections we will focus on the direction we took in this work, which is based on a different re-representation of the fitness function. We will also briefly discuss another possibility to make information landscapes viable for EPAs.

### 3.2. A sparse representation of fitness

Let $\Omega$ be a program search space and $f$ a fitness function over $\Omega$. We assume that $\Omega$ is ordered. One can then represent $f$ using a corresponding ordered set $\mathcal{F}(\Omega) = \{f(p) \mid p \in \Omega\}$. Clearly, $\mathcal{F}(\Omega)$ is an exact representation of the fitness function. So, borrowing ideas from information landscapes, one could imagine estimating the performance of an EPA using the linear model

$$P(f) \approx a_0 + \sum_{p \in \Omega} a_p f(p). \tag{2}$$

Note that this model has far fewer parameters than Eq. (1). Nonetheless, their identification via ordinary linear regression is problematic due to the size of typical program spaces and the indeterminacy resulting from the semantic equivalence of distinct programs. However, Eq. (2) can be transformed into a workable model if we accept to only partially represent the fitness function. That is, instead of using $\mathcal{F}(\Omega)$ as a representation for $f$, we use $\mathcal{F}(\mathcal{S}) = \{f(p)\colon\ p \in \mathcal{S}\}$, where $\mathcal{S} \subseteq \Omega$. This leads to in the following model

$$P(f) \approx a_0 + \sum_{p \in \mathcal{S}} a_p f(p). \tag{3}$$

The advantage of Eq. (3) is that the cardinality of $\mathcal{S}$ is under our control. Therefore, we can easily ensure that there are not too many coefficients to identify via regression, making the problem well-posed and keeping the size of the training set under control. For these reasons we adopted Eq. (3) as a stepping stone towards our performance model.

[2] The problem is not simply caused by Eq. (1) being linear in the $d(.,.)$ terms. For example, if one used the highly non-linear model $P(v) \approx \prod a_i^{v_i}$ and we had that $v_j \equiv v_k$, then $a_j^{v_j} a_k^{v_k} \equiv (a_j a_k)^{v_j}$ which implies that any model where $a_j \times a_k = \text{constant}$ would have the same quality, making the choice of $a_j$ and $a_k$ non-unique.

While, in principle, this approach could be applied to any domain, when the objective is to model EPAs, Eq. (3) can be further specialised, thereby revealing a deeper structure that we will exploit for modelling purposes.

In particular, it is common practice in EPAs to use a fitness function, $f(p)$, which evaluates how similar the functionality of a program $p \in \Omega$ is to a target functionality, $t$. Typically, $t$ is represented via a finite set of training input–output pairs (called *fitness cases* in GP). So, fitness is generally computed as $f(p) = \sum_{i=1}^{\ell} g(p(x_i), t(x_i))$ where $\{(x_i, t(x_i))\}$ is a set of fitness cases of cardinality $\ell$ (each $x_i$ being a set of inputs and $t(x_i)$ the corresponding desired output) and $g$ is a function which evaluates the degree to which the behaviour of $p$ (i.e., its outputs) matches $t$ in each fitness case.[3] Typically, the set of fitness-case inputs, $\{x_i\}$, is fixed, the corresponding outputs, i.e., $t(x_i)$ and $p(x_i)$, are numbers and $g(a, b) = |a - b|^k$ with $k = 1$ or $k = 2$.

Under these conditions (see [68]) we can think of both $t$ and $p$ as being $\ell$-dimensional vectors, $\mathbf{t} = (t_1, \ldots, t_\ell)$ and $\mathbf{p} = (p_1, \ldots, p_\ell)$, respectively, where $t_i = t(x_i)$ and $p_i = p(x_i)$. We can then represent $f(p) = d(\mathbf{p}, \mathbf{t})$ where $d$ is a similarity measure between vectors. So, different $\mathbf{t}$ vectors induce different fitness measures on the program search space. Also, Eq. (3) transforms into

$$P(\mathbf{t}) \approx a_0 + \sum_{p \in \mathcal{S}} a_p \cdot d\big(\mathbf{p}(p), \mathbf{t}\big) \tag{4}$$

where we used $P(\mathbf{t})$ instead of $P(f)$ since fitness is really determined by $\mathbf{t}$, and $\mathbf{p}(p)$ stands for the output produced by program $p$ when tested on the fitness cases associated to a problem.

Eq. (4) is specialised to model performance of program induction algorithms. In the following section, we will generalise it to make it possible to model performance of both EPAs and searchers in other domains. Before we do this, however, we should note that $\mathcal{F}(\mathcal{S})$ is only a sparse representation of $f$ and, so, Eqs. (3) and (4) will necessarily be less accurate than Eq. (2). Of course, since the latter is itself an estimate, it makes sense to ask whether a careful choice of $\mathcal{S}$ might still produce reasonable results.[4] We should also note that, as suggested by one of the reviewers, an alternative approach to making information landscapes viable for EPAs would be to sample them, i.e., to use only a subset of the components of $v$ in Eq. (1).[5]

### 3.3. Performance models and problem landscapes

In program induction, in principle, the search space may not contain a program with exactly the target functionality $\mathbf{t}$. This acts as the second term of comparison in the factors $d(\mathbf{p}(p), \mathbf{t})$ in Eq. (4). It stands to reason that by also allowing the first term of comparison, $\mathbf{p}(p)$, to not be the functionality of a program actually in $\Omega$ we can generalise Eq. (4), obtaining

$$P(\mathbf{t}) \approx a_0 + \sum_{\mathbf{p} \in \mathbb{S}} a_{\mathbf{p}} \cdot d(\mathbf{p}, \mathbf{t}) \tag{5}$$

where $\mathbb{S}$ is a subset of the set of all possible *program behaviours* (independently on whether or not there is a program in $\Omega$ implementing them). In particular, $\mathbb{S} \subset \mathbb{R}^\ell$ for continuous regression problems and $\mathbb{S} \subset \{0, 1\}^\ell$ for Boolean induction problems.

Note that, while we derived Eq. (5) for the case where $\mathbf{p}$ and $\mathbf{t}$ are $\ell$-dimensional vectors, there is no reason why we cannot generalise it to objects of any type and structure as long as one can define a similarity measure $d(\mathbf{p}, \mathbf{t})$ for them. With this further generalisation, effectively *Eq. (5) can also be applied to problems other than program induction*. In particular, we can now interpret $\mathbb{S}$ as a subset of the set of all possible *problems*, rather than a subset of the set of all possible *programs* (or, more precisely, program behaviours). For example, $\mathbb{S}$ could be a subset of all possible Boolean functions of 4 inputs as easily as it could be a subset of bin packing problems. In this case the terms $d(\mathbf{p}, \mathbf{t})$ in Eq. (5) effectively define a *problem landscape*, i.e., they represent the *degree of similarity between two problems*.[6]

This interpretation of Eq. (5) is the model we will explore in the rest of the paper. The idea is general and, as we will show, can successfully be applied to a variety of program-induction algorithms as well as other machine learners and problem solvers. However, there is an important question that needs answering before we can proceed with using Eq. (5). Clearly, we should expect different choices of $\mathbb{S}$ to produce models of different accuracy. So, how should we choose the elements of $\mathbb{S}$? The process we adopted in this work is described in the next section.

[3] This means that one is not free to associate any fitness to a program: fitness is always computed indirectly via some form of similarity measure. This constrains the number and type of fitness functions one can create for EPAs.

[4] Naturally, selecting $\mathcal{S}$ is a delicate task that, if done inappropriately, could introduce undesired biases. As discussed in Section 3.4, here we have successfully used a simple procedure that involves the sampling of $\Omega$ followed by a further selection step. Such procedures can be generalised to other domains. However, it is conceivable that in some cases the identification of a suitable $\mathcal{S}$ could be more difficult.

[5] We tested this idea in experiments (not reported for space limitations) with the class of 3-input Boolean-induction problems and all the different EPAs considered in this paper. The models produced were in all cases inferior to those obtained with sparse fitness landscapes. Nonetheless, results indicated that the approach is viable and worthy of further exploration and development in future research.

[6] In the case of program induction algorithms, problems and programs can be represented as structures (e.g., the vectors $\mathbf{t}$ and $\mathbf{p}$, respectively) within the same space whenever problems are described in terms of desired outputs to specific inputs. So, there is not a great deal of difference between Eqs. (4) and (5) nor between interpreting $\mathbb{S}$ as a set of program behaviours or as a set of problems. However, if our performance model is applied to general problem solving situations, then it is important to remember the distinction between the two.

### 3.4. Model identification

To instantiate Eq. (5) for EPAs and other algorithms we need a training set of problems $T$, a validation set $V$, a closeness measure $d$ and a set of problems $\Sigma$ from which to draw the elements of $\mathbb{S}$. The sets $T$ and $\mathbb{S}$ are used to identify the coefficients $a_{\mathbf{p}}$ so as to obtain a good fit. The validation set $V$ is used to test the generality of the model. $T$ and $V$ are composed by pairs $(\mathbf{t}, P(\mathbf{t}))$ where $\mathbf{t}$ is a problem (e.g., for EPAs, a target vector) and $P(\mathbf{t})$ is the corresponding performance of the algorithm under study, which, for stochastic algorithms, is estimated by averaging performance over multiple independent runs.

An important issue is the choice of $\Sigma$. For discrete domains of reasonably small size, one option is to use all the possible elements in a class of problems. Alternatively, if the cardinality of a class of problems is too big or infinite, one could construct $\Sigma$ by drawing a representative set of samples from the class. In this work we decided to use the latter approach. Since typically also the training set $T$ needs to be produced in via sampling, we decided to take $\Sigma \equiv T$. Section 4 describes the simple procedures used to obtain $T$ for different benchmark problems.

We used the Least Angle Regression (LARS) algorithm (see [76]) as a method to decide which problems from $T$ to include in $\mathbb{S}$. LARS works as follows. It starts by setting all the coefficients $a_{\mathbf{p}}$ to zero and finds the predictor most correlated with the response (i.e., the performance measure). Then it takes the largest possible step in the direction of this predictor until some other predictor has as much correlation with the residual. At this point, LARS proceeds in the direction of the equiangular between the two predictors until a third variable has as much correlation with the residual. The process continues until the last predictor is incorporated into the model.[7]

In our version of LARS we stop the algorithm after $m$ steps, where $m$ is the desired size for the set $\mathbb{S}$, and we pick the $m$ problems from $T$ chosen by the algorithm so far as the elements of $\mathbb{S}$. In this way we are certain to retain in $\mathbb{S}$ elements of $T$ having a high correlation with the performance, thereby increasing the accuracy of the model over alternative ways of choosing $\mathbb{S}$.

To automate the choice of the parameter $m$, we wrapped the procedure just described within a 5-fold cross-validation loop. That is, $T$ was split into five sets of equal size: four sets were used to produce a model while the remaining set was used to assess its generalisation. The process was repeated 5 times, each time leaving out a different fifth of $T$, resulting in a prediction of performance for all problems in $T$. This procedure was iterated for $m = 1, 2 \ldots, |T|$ with the aim of identifying the value of $m$ which provided the best generalisation. The overall generalisation error was measured via the Relative Squared Error (RSE) [77] which is defined as $\mathrm{RSE} = \sum_i (P_i - \tilde{P}_i)^2 / \sum_i (P_i - \bar{P})^2$, where $i$ ranges over $T$, $P_i$ is the average performance recorded for problem $i$, $\tilde{P}_i$ is the performance predicted by the model, and $\bar{P}$ is the average performance over all problems.

Having identified the optimal set $\mathbb{S}$ through the procedure described above, we finally used ordinary least squares to obtain the model's coefficients $a_{\mathbf{p}}$.

## 4. Test problems and performance measures

To illustrate the scope and effectiveness of the approach, we considered different classes of problems as well as different algorithms and performance measures. Firstly, we tested our approach in problems related to EPAs, namely continuous symbolic regression of rational functions and Boolean inductions problems using different versions of GP, GEP, and a Stochastic Iterated Hill Climber (SIHC). Secondly, we modelled an artificial neural network (ANN) training algorithm applied to Boolean induction problems. Finally, we also tested the approach on the one-dimensional off-line bin-packing problem modelling the well-known First Fit Decreasing (FFD) and Best Fit Decreasing (BFD) heuristics. We present the problems in more detail in the rest of this section, while we describe the algorithms used to solve them in Section 5.

### 4.1. EPA's problems

For program induction, we considered two radically different classes of problems, namely continuous symbolic regression and Boolean function induction problems, and three typical performance measures: Best of Run Fitness (BRF), a normalised version of BRF (see below), and success rate. Continuous *symbolic regression* requires finding a program (seen as a function that transforms some numerical inputs into one output) which fits a set of data points. In Boolean function induction problems an algorithm is asked to find a program that implements a given truth table. In both cases the sum of absolute errors was used as a fitness measure.

A benchmark set was created for continuous symbolic regression by generating 1100 different rational functions of the form $t(x) = W(x)/Q(x)$, where $W(x)$ and $Q(x)$ are polynomials, using the following procedure: $W(x)$ and $Q(x)$ were built by randomly choosing their degrees in the range 2 to 8, and then choosing random real coefficients in the interval $[-10, 10]$ for the powers of $x$ up to the chosen degrees. Each of the rational functions in the set was then sampled at 21

[7] We decided to use LARS because it is less greedy than other model selection algorithms like forward selection or all-possible-subsets regression. Moreover, simple modifications of it implement Lasso or the forward stepwise linear regression algorithm, which we would like to test in future refinements of this research.

points uniformly distributed in the interval $[-1, 1]$, resulting in a target vector $\mathbf{t} \in \mathbb{R}^{21}$ (the number of fitness cases and their range were chosen based on typical values used in the GP literature).

For symbolic regression problems, for each $\mathbf{t}$ vector, we performed 100 independent runs recording BRFs. The corresponding $P(\mathbf{t})$ value was obtained by averaging such BRFs. We also computed a normalised version of BRF (or NBRF for short), which involved first normalising the target vector and the behaviour of the best individual found in each run, and then summing the absolute differences between the components of these normalised vectors.[8]

We also created a benchmark set of 1100 random problems from the class of 4-input Boolean induction problems. Each problem was represented by a vector of length 16 (a truth table). So, in this case $\mathbf{t} \in \{0, 1\}^{16}$. As a performance measure for an algorithm we took the success rate, i.e., the fraction of successful runs, in 100 independent runs.

Each benchmark set of 1100 random problems was divided up into two sets: a training set $T$ composed of 500 problems, and a validation set $V$ comprising the remaining 600 problems.

### 4.2. Artificial neural network problems

We used Boolean induction problems also to test our approach when modelling a learning algorithm for ANNs. The job of the ANN was to learn functions with 3 inputs and 2 outputs. These were represented with $8 \times 2$ truth tables. We selected 1100 such tables randomly without replacement: 500 went in the training set $T$, and the remaining 600 formed the validation set $V$.

For each problem in the benchmark set, an ANN was iteratively trained using the 8 possible different inputs patterns. Training continued until the mean square error on the outputs was less than 0.05. The training process was repeated 500 times (starting with random sets of weights and biases). The average number of epochs required to train the network was then taken as the performance of the learning algorithm on the problem.

The ANN domain differs from program induction in two important ways: (a) typically, ANNs have multiple outputs while above, for simplicity, we only considered the case of programs with one output; (b) the functionality of ANNs is determined by their topology, weights, biases and activation functions, rather than a sequence of instructions. However, from the point of view of modelling performance the two domains are very similar. Like for programs, also the behaviour of ANNs can be represented via the collection of the outputs they produce in response to a set of input patters. Thus, we can linearise the $8 \times 2$ table representing an ANN-training problem (by reading its elements column by column and row by row) thereby obtaining a vector $\mathbf{t} \in \{0, 1\}^{16}$ which we can directly use in Eq. (5) to model of the performance of ANN training algorithms.

### 4.3. One-dimension off-line bin packing

The objective in bin packing is to pack a certain number of items using the smallest possible number of bins, such that no bin's content overflows. The main difference between this and the other domains described above is that, in bin packing, problems and algorithm outputs/behaviours do not live in the same space. In bin packing a solution to a problem is an assignment of items to bins, while a problem itself is a list detailing the size of each item to be packed. Also, typically there is no predefined target behaviour, e.g., in the form of an assignment, which could then be used to measure similarity with the output produced by an algorithm. This suggests that when modelling bin packers with Eq. (5) it is best to interpret the terms $\mathbf{p}$ and $\mathbf{t}$ as problems.

The natural representation for bin-packing problems is a list of item sizes. However, in the off-line version of the problem considered here, the order of the items in the list is unimportant because the solver can freely choose the order in which to bin them. For this reason, we represented problems using histograms indicating how many items of each size needed packing. The number of slots in a histogram of item sizes is upper-bounded by the size of the largest possible item to be packed. Since in our experiments item sizes were integers between 1 and 100, problems were represented with histograms including 100 slots. Such histograms were then represented as 100-dimensional vectors.

We created random bin-packing problems using the procedure presented in [78]. All problems required packing 1000 items. We considered five problem classes with the following ranges of item sizes: 1–100, 10–90, 20–80, 30–70, and 40–60. For each range, 800 different histograms were created, from which 400 were used in the training set, and the remaining 400 were included in the validation set. If $B_{min}$ and $B_{max}$ are the minimum and maximum size of the items in a problem class, respectively, the histograms followed a multinomial distribution with success probabilities given by: $p_i = \frac{1}{B_{max} - B_{min} + 1}$ for $i = B_{min}, \ldots, B_{max}$ and $p_i = 0$ otherwise. In total the training set consisted of 2000 different problems and the validation set included a further 2000 problems.

The performance of a bin-packing algorithm was the number of bins used to pack all the items. Since we considered only deterministic algorithms, we performed one run with each problem.

[8] If $\mu$ and $\sigma$ are the mean and standard deviation of the elements $t_i$ of $\mathbf{t}$, respectively, the elements of the normalised target vector were set to $\frac{t_i - \mu}{\sigma}$. The normalised program-behaviour vectors were similarly obtained by shifting and scaling the components of $\mathbf{p}$.

**Table 1**
Parameters and primitives used in the GP experiments. Only combinations of $p_{xo}$ and $p_m$ such that $p_{xo} + p_m \leqslant 100\%$ were used since crossover and mutation are mutually exclusive operators in GP.

| | |
|---|---|
| Function set (rational problems) | $\{+, -, *, /$ (protected to avoid divisions by 0)$\}$ |
| Function set (Boolean problems) | {*AND*, *OR*, *NAND*, *NOR*} |
| Terminal set (rational problems) | $\{x, \mathbb{R}\}$ |
| Terminal set (Boolean problems) | $\{x_1, x_2, x_3, x_4\}$ |
| Crossover rate $p_{xo}$ | 100%, 90%, 50%, and 0% |
| Mutation rate $p_m$ | 100%, 50%, and 0% |
| Maximum tree depth used in mutation | 4 |
| Selection mechanism | Tournament (size 2) and roulette-wheel |
| Population size | 1000 |
| Number of generations | 50 |
| Number of independent runs | 100 |

**Table 2**
Parameters used in the GEP experiments.

| | |
|---|---|
| Function set (rational problems) | $\{+, -, *, /$ protected$\}$ |
| Function set (Boolean problems) | {*AND*, *OR*, *NAND*, *NOR*} |
| Terminal set (rational problems) | $\{x, \mathbb{R}\}$ |
| Terminal set (Boolean problems) | $\{x_1, x_2, x_3, x_4\}$ |
| Head length | 63 |
| Number of genes | 3 |
| Mutation rate $p_m$ | 5% |
| 1 point recombination rate | 20% |
| 2 point recombination rate | 50% |
| Gene recombination rate | 10% |
| IS-transposition rate | 10% |
| IS elements length | 1, 2, 3 |
| RIS-transposition rate | 10% |
| RIS elements length | 1, 2, 3 |
| Gene transposition rate | 10% |
| Selection mechanism | Tournament (size 2) and roulette-wheel |
| Population size | 1000 |
| Number of generations | 50 |
| Number of independent runs | 100 |

## 5. Systems and parameter settings

### 5.1. GP systems

We used two different implementations of GP, both tree-based and both using subtree crossover and subtree mutation. One system was essentially identical to the one used by Koza [6], the only significant difference being that we selected crossover and mutation points uniformly at random, while [6] used a non-uniform distribution. The other system was TinyGP [79] with the modifications presented in [19] to allow the evolution of constants. The main difference between the two is that Koza's system is generational (all selected individuals in the population reproduce in parallel to create the next generation) while TinyGP uses a steady-state strategy (offspring are immediately inserted in the population without waiting for a full generation to be completed).

Table 1 shows the parameters and primitives used. With the generational GP system, besides the traditional roulette-wheel selection (which gives parents a probability of being chosen proportional to their fitness), we also used tournament selection (which sorts a random set of individuals based on fitness and picks the best). So, in total we tested three variants of GP: generational with tournament selection, generational with roulette-wheel selection, and steady-state with tournament selection.

### 5.2. GEP systems

The performance of GEP in preliminary runs was much worse than that of GP. We found that this was mainly due to the standard GEP initialisation method. Thus, we replaced it with a technique equivalent to the ramped-half-and-half method [6] obtaining a considerable performance improvement. We used three different versions of GEP: generational with tournament selection, generational with roulette-wheel selection, and steady-state with tournament selection. The parameters and primitives used in our GEP runs are shown in Table 2.

There are important differences between GP and GEP. Firstly, in GP, programs are represented using trees which can vary in size and shape, while GEP uses a fixed-size linear representation with two components: a head that can contain functions and terminals, and a tail that can only contain terminals. Secondly, in tree-based GP, the operators act on trees, modifying nodes and exchanging subtrees, while in GEP, the genetic operators are more similar to those used in GAs having only to

**Table 3**
Parameters used in the SIHC experiments.

| Mutation | Maximum number of mutations | Maximum number of fitness evaluations |
|---|---|---|
| Sub-tree | 50, 500, 1000, and 25,000 | 50,000 |
| Uniform | 25,000 | 50,000 |

comply with the constraints on the primitives allowed in the head and the tail. Finally, in GP, no transformation is required before the fitness of a program tree can be computed, while in the GEP the fixed-length representation is transformed into a tree structure first. This is done by picking primitives from the fixed-length string and inserting them into a tree following a breadth-first order. These differences between GP and GEP translate into radically different behaviours and performance of these two classes of EPAs.

### 5.3. SIHC systems

We used a Stochastic Iterated Hill Climber similar to the one presented in [80] but with different mutation operators: sub-tree mutation, which is the same type of mutation used in the GP runs, and a mutation operator, which we will call *uniform mutation*, where each node in the tree is mutated with a certain probability. A node to be mutated is replaced with: (a) one of its children, (b) a random node of the same arity, or (c) a randomly generated tree which includes also the tree originally rooted at the mutated node as a subtree.

SIHC starts by creating a random program tree using the same procedure and primitives as for the GP and GEP experiments. SIHC then mutates this program until a fitter one is found. This replaces the initial program and the mutation process is resumed. When a maximum number of allowed mutations is reached the individual is set aside, a new random individual is created and the process is repeated. SIHC terminates when a solution has been found or when a maximum number of fitness evaluations is reached. Table 3 shows the parameters used for the SIHC experiments.

### 5.4. ANN and bin packing heuristics

The ANN we used to exercise our training algorithm was a fully connected feed-forward network with 3 layers and 7 hidden neurons. The activation function was a symmetric sigmoid. The algorithm used to train it was iRPROP [81]. The initial weights and biases for the network were randomly and uniformly chosen in the range $[-0.1, +0.1]$.

For the case of bin packing, we used two well-known human-designed heuristics: First Fit Decreasing (FFD) and Best Fit Decreasing (BFD). Both work by first sorting the items by non-increasing size. Then, FFD places each item in the first bin where it can fit, while BFD places items where they fit most tightly. We used two different bin sizes: 100 and 150.[9]

## 6. Results

### 6.1. Not all similarity measures are equal for EPA performance modelling

Eq. (5) was derived considering that in many GP problems the fitness function, $f$, is effectively a distance. However, having extended the interpretation of the model in Section 3.3, it is clear that we are free to choose a similarity measure, $d$, which does not coincide with $f$. It, therefore, makes sense to compare the quality of the models obtained with different $d$ measures.

Table 4 shows how the 5-fold cross-validation RSE of the models varies across three GP systems for different $d$ functions. For symbolic regression, RSEs were computed for the BRF and NBRF performance measures, while we used the success rate in Boolean induction. The quality of the models depends on $d$. For the case of rational problems, when using BRF the closeness measure $\sum_i |t_i - p_i|$ was best overall, while when performance was evaluated using NBRF, $\sum_i \ln(1 + |t_i - p_i|)$ was best with $\sum_i |t_i - p_i|$ a close second. For the case of Boolean induction problems, $(\mathbf{p} \cdot \mathbf{t})^2$ provided the lowest RSE values. Similar results were obtained with all program induction systems described in Section 5. So, in the rest of the paper, we will use the optimal similarity measure identified above for each class of problems and performance measure, across all algorithms.

### 6.2. Performance models of EPAs

Table 5 presents an accuracy comparison (in terms of RSE) for the performance models of the GP, GEP and SIHC systems presented in Section 5. The lowest RSE values are associated with the generational systems with roulette-wheel selection and the BRF measure. However, in virtually all cases RSEs are small for both training and validation sets and always well below 1 (i.e., even the worst models are able to predict performance much better than the mean).

[9] When the size of the bins is 150 both heuristics gave the same performance so in the experiments we only present the results for FFD.

**Table 4**
Quality of the models of GP (with crossover rate of 90% and no mutation) for different closeness measures (BRF = best of run fitness, NBRF = normalised BRF).

| Configuration | | | Rational functions | | | | Boolean functions | |
|---|---|---|---|---|---|---|---|---|
| Type | Selection | $d$ | BRF | | Normalised BRF | | Success rate | |
| | | | $\lvert\mathbb{S}\rvert$ | RSE | $\lvert\mathbb{S}\rvert$ | RSE | $\lvert\mathbb{S}\rvert$ | RSE |
| Generational | Roulette | $\sqrt{\frac{1}{n}\sum_i (t_i - p_i)^2}$ | 26 | 0.0292 | 251 | 0.5261 | 399 | 0.4629 |
| | | $\sum_i \lvert t_i - p_i\rvert$ | 55 | **0.0215** | 131 | 0.4980 | 1 | 1.0102 |
| | | $(\mathbf{t}\cdot\mathbf{p})^2$ | 5 | 0.1197 | 48 | 0.6251 | 126 | **0.3007** |
| | | $(\mathbf{t}\cdot\mathbf{p})^3$ | 1 | 1.0862 | 1 | 1.0010 | 168 | 0.3736 |
| | | $\exp(-\lVert\mathbf{t}-\mathbf{p}\rVert^2)$ | 1 | 1.0012 | 390 | 0.5780 | 399 | 0.8045 |
| | | $\sum_i \ln(1+\lvert t_i - p_i\rvert)$ | 399 | 0.5621 | 126 | **0.4895** | 1 | 1.0102 |
| Generational | Tournament | $\sqrt{\frac{1}{n}\sum_i (t_i - p_i)^2}$ | 12 | 0.3290 | 227 | 0.4679 | 394 | 0.5788 |
| | | $\sum_i \lvert t_i - p_i\rvert$ | 340 | **0.1577** | 153 | 0.3658 | 1 | 1.0087 |
| | | $(\mathbf{t}\cdot\mathbf{p})^2$ | 6 | 0.5269 | 45 | 0.6078 | 118 | **0.4158** |
| | | $(\mathbf{t}\cdot\mathbf{p})^3$ | 7 | 0.8931 | 1 | 1.0027 | 150 | 0.4876 |
| | | $\exp(-\lVert\mathbf{t}-\mathbf{p}\rVert^2)$ | 3 | 1.0020 | 380 | 0.5408 | 399 | 0.8993 |
| | | $\sum_i \ln(1+\lvert t_i - p_i\rvert)$ | 67 | 0.3570 | 125 | **0.3593** | 1 | 1.0087 |
| Steady state | Tournament | $\sqrt{\frac{1}{n}\sum_i (t_i - p_i)^2}$ | 25 | 0.5168 | 310 | 0.4567 | 399 | 0.7303 |
| | | $\sum_i \lvert t_i - p_i\rvert$ | 110 | 0.3148 | 113 | 0.4554 | 1 | 1.0100 |
| | | $(\mathbf{t}\cdot\mathbf{p})^2$ | 2 | 0.9076 | 52 | 0.5640 | 104 | **0.6117** |
| | | $(\mathbf{t}\cdot\mathbf{p})^3$ | 5 | 1.2406 | 22 | 0.9988 | 128 | 0.6822 |
| | | $\exp(-\lVert\mathbf{t}-\mathbf{p}\rVert^2)$ | 1 | 1.0029 | 384 | 0.5954 | 399 | 0.9245 |
| | | $\sum_i \ln(1+\lvert t_i - p_i\rvert)$ | 100 | **0.2949** | 133 | **0.4524** | 1 | 1.0100 |

**Table 5**
Quality of the model (RSE) for different GP, GEP and SIHC systems and parameter settings. The models used the optimal closeness measures identified in Section 6.1.

| Configuration | | | | Rational functions | | | | | | Boolean functions | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Type | Selection | $p_{xo}$ | $p_m$ | Best of run fitness | | | Normalised BRF | | | Success rate | | |
| | | | | $\lvert\mathbb{S}\rvert$ | $T$ set | $V$ set | $\lvert\mathbb{S}\rvert$ | $T$ set | $V$ set | $\lvert\mathbb{S}\rvert$ | $T$ set | $V$ set |
| Generational | Roulette | 1.00 | 0.00 | 47 | 0.0123 | 0.0209 | 127 | 0.2304 | 0.5246 | 128 | 0.1640 | 0.2877 |
| | | 0.90 | 0.00 | 55 | 0.0062 | 0.0267 | 126 | 0.2251 | 0.5375 | 126 | 0.1510 | 0.2962 |
| | | 0.50 | 0.50 | 43 | 0.0193 | 0.0243 | 143 | 0.1832 | 0.4999 | 127 | 0.1505 | 0.2833 |
| | | 0.00 | 1.00 | 58 | 0.0179 | 0.0359 | 128 | 0.1986 | 0.4907 | 125 | 0.1712 | 0.3058 |
| | | GEP | | 180 | 0.0005 | 0.0101 | 118 | 0.2375 | 0.5212 | 129 | 0.1969 | 0.3745 |
| Generational | Tournament | 1.00 | 0.00 | 122 | 0.0065 | 0.2683 | 137 | 0.1612 | 0.4082 | 118 | 0.2216 | 0.4065 |
| | | 0.90 | 0.00 | 340 | 0.0003 | 0.5857 | 125 | 0.1794 | 0.4257 | 117 | 0.2530 | 0.3941 |
| | | 0.50 | 0.50 | 18 | 0.1503 | 0.8291 | 160 | 0.1382 | 0.4130 | 116 | 0.2413 | 0.4010 |
| | | 0.00 | 1.00 | 112 | 0.0107 | 0.4009 | 167 | 0.1352 | 0.4291 | 121 | 0.2760 | 0.4686 |
| | | GEP | | 99 | 0.0048 | 0.1270 | 129 | 0.1877 | 0.4477 | 124 | 0.2424 | 0.4501 |
| Steady state | Tournament | 1.00 | 0.00 | 106 | 0.0193 | 0.6224 | 131 | 0.2067 | 0.5778 | 116 | 0.3501 | 0.5401 |
| | | 0.90 | 0.00 | 110 | 0.0182 | 0.4300 | 133 | 0.2092 | 0.5634 | 104 | 0.3531 | 0.5820 |
| | | 0.50 | 0.50 | 100 | 0.0181 | 0.4168 | 130 | 0.2286 | 0.5967 | 114 | 0.3735 | 0.6379 |
| | | 0.00 | 1.00 | 112 | 0.0138 | 0.3549 | 132 | 0.2331 | 0.6367 | 109 | 0.4159 | 0.6336 |
| | | GEP | | 14 | 0.1787 | 0.5608 | 126 | 0.1845 | 0.4243 | 88 | 0.3966 | 0.5512 |
| Sys. | Mut. | Max Mut. | | $\lvert\mathbb{S}\rvert$ | $T$ set | $V$ set | $\lvert\mathbb{S}\rvert$ | $T$ set | $V$ set | $\lvert\mathbb{S}\rvert$ | $T$ set | $V$ set |
| SIHC | Sub-tree | 50 | | 170 | 0.0054 | 0.1674 | 114 | 0.2091 | 0.4349 | 118 | 0.2294 | 0.4045 |
| | | 500 | | 150 | 0.0046 | 0.4079 | 113 | 0.1934 | 0.4540 | 121 | 0.2109 | 0.3989 |
| | | 1000 | | 220 | 0.0026 | 0.8974 | 93 | 0.2361 | 0.4378 | 122 | 0.2066 | 0.3787 |
| | | 25,000 | | 193 | 0.0041 | 0.2579 | 84 | 0.2510 | 0.4587 | 125 | 0.1786 | 0.3120 |
| | Unif. | 25,000 | | 93 | 0.0192 | 0.3437 | 130 | 0.2044 | 0.4890 | 118 | 0.2174 | 0.3641 |

RSE figures provide an objective indication of model quality. However, it may be difficult to appreciate the accuracy of our models from just such figures. To give a clearer illustration of this, Fig. 1 shows scatter plots of the actual performance of a GP system *vs.* the performance estimates obtained by the model in the training and validation sets for symbolic regression and Boolean induction. The solid diagonal line in each plot represents the behaviour of a perfect model. The fact that data form tight clouds around that line is a clear qualitative indication of the accuracy of the models. Other systems and parameter settings provided similar results.

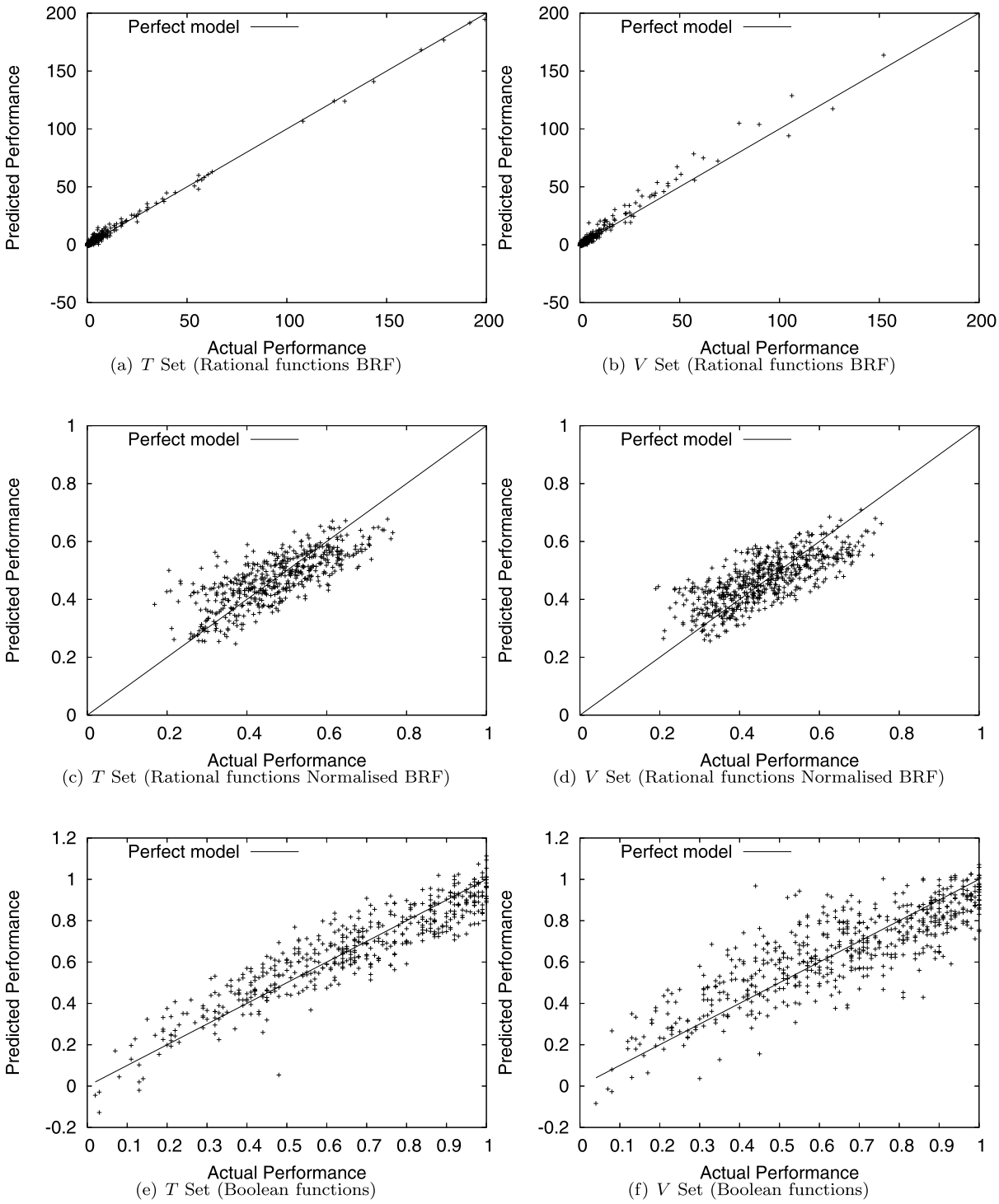



**Fig. 1.** Scatter plots of the performance measured *vs.* the performance obtained from the model for continuous regression problems using BRF (a & b) and normalised BRF (c & d) as performance measures, and Boolean induction problems (e & f), for both the training set and the validation set. The data refer to a GP system with 90% crossover rate, no mutation, and roulette-wheel selection.

### *6.3. Performance models of ANN training*

Table 6 shows the quality of the models of ANN training resulting from the use of different similarities measures. The table reports the RSE obtained using cross validation (column 2) as well as the RSE obtained on the training and validation

**Table 6**
Quality of the models of ANN training for different closeness measures.

| Closeness measure | $\lvert\mathbb{S}\rvert$ | RSE | $T$ RSE | $V$ RSE |
|---|---|---|---|---|
| $\sqrt{\frac{1}{n}\sum_i (t_i - p_i)^2}$ | 399 | 0.5166 | 0.0011 | 0.4846 |
| $\sum_i \lvert t_i - p_i\rvert$ | 7 | 0.9954 | 0.9411 | 1.0004 |
| $(\mathbf{t}\cdot\mathbf{p})^2$ | 134 | **0.3711** | 0.1814 | 0.3567 |
| $(\mathbf{t}\cdot\mathbf{p})^3$ | 189 | 0.4798 | 0.1220 | 0.4502 |
| $\exp(-\lVert\mathbf{t}-\mathbf{p}\rVert^2)$ | 399 | 0.8212 | 0.0016 | 0.7953 |
| $\sum_i \ln(1+\lvert t_i - p_i\rvert)$ | 7 | 0.9954 | 0.9411 | 1.0004 |

**Table 7**
Quality of the model for the different heuristics in the off-line bin packing problem.

| Name | Size of bins | $\lvert\mathbb{S}\rvert$ | Cross-val. RSE | $T$ RSE | $V$ RSE |
|---|---|---|---|---|---|
| FFD | 150 | 1315 | 0.0031 | 0.0002 | 0.0028 |
| FFD | 100 | 828 | 0.1405 | 0.0414 | 0.1413 |
| BFD | 100 | 855 | 0.1342 | 0.0377 | 0.1361 |

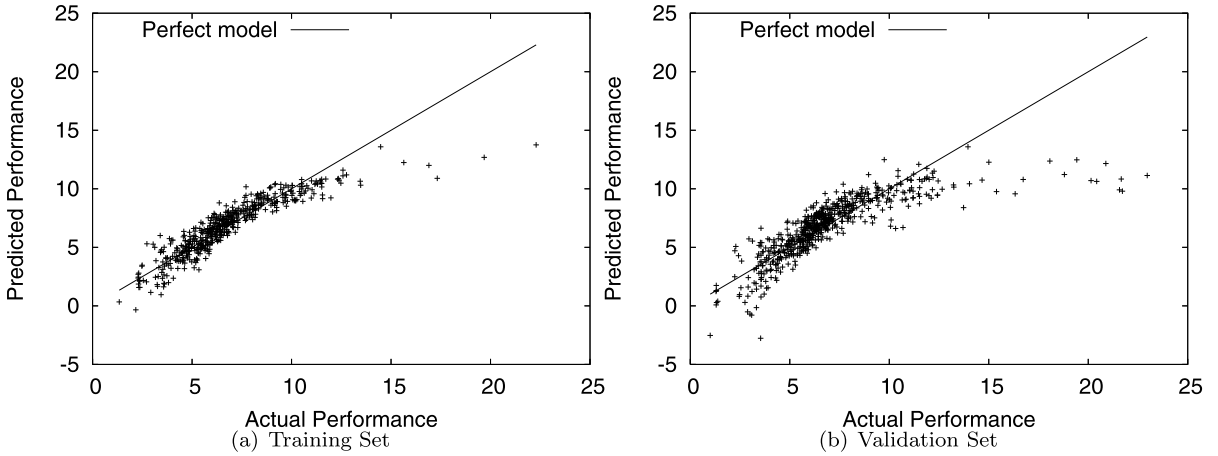



**Fig. 2.** Scatter plots of the actual performance (epochs) *vs.* the performance estimated by the model for ANN training problems.

sets (columns 3 and 4, respectively). The $d$ function with the best cross-validation RSE was $(\mathbf{t}\cdot\mathbf{p})^2$ which also corresponds to the lowest RSE on the validation set. RSE values suggest that our model produced good predictions.

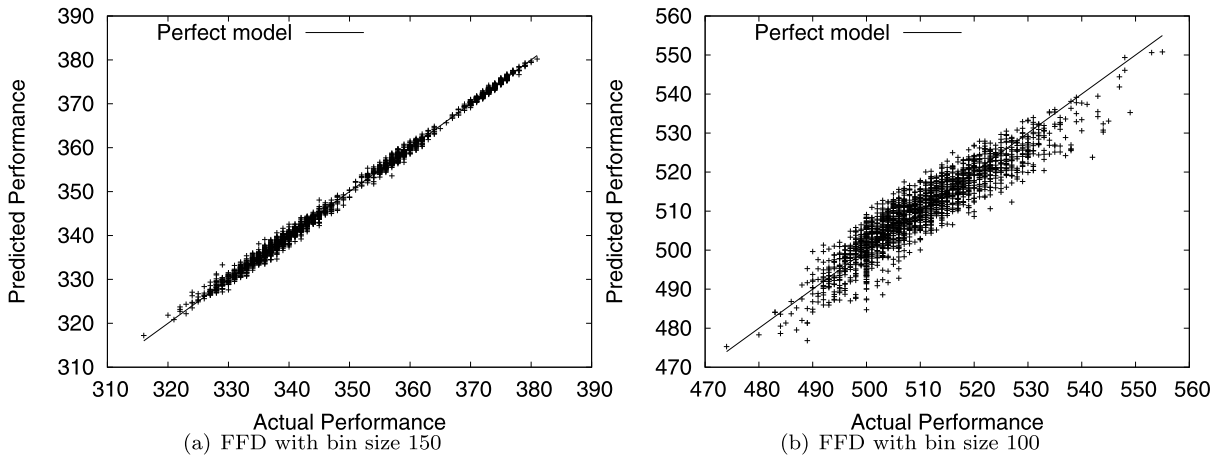
Fig. 2 reports a scatter plot of the actual performance *vs.* the performance estimated by the model (when $d = (\mathbf{t}\cdot\mathbf{p})^2$). This shows that the model was able to accurately predict actual performance for most problems. Only for problems where training took longer than about 15 epochs, the model significantly underestimated performance. The reason of this is that there are only very few problems requiring a high number of learning epochs in our benchmark set.[10]

### *6.4. Performance models of bin packing algorithms*

We tested how the quality of models obtained with different closeness measures varied also for bin packing. We found that the sum of absolute differences produced models with lowest RSE value. Thus, we adopted this for the experiments reported below.

Table 7 presents the RSE values for the FFD and BFD heuristics for two bin sizes. In all cases our models predicted very accurately the performance of the two bin-packers and generalised very well. The size of $\mathbb{S}$ was considerably bigger than for models of other systems, most likely because of the much larger number of degrees of freedom (100) associated with our bin packing problems. We show scatter plots of the actual performance *vs.* the estimated performance in the validation set for FFD in Fig. 3. Again, the data closely follow the perfect-model line.

[10] If accurate prediction of performance of rare cases (such as long runs) is important for a user, one can correspondingly bias the training set so as to ensure that these are over-represented and, thus, more accurately modelled.




(a) FFD with bin size 150 (b) FFD with bin size 100

**Fig. 3.** Scatter plots of the actual performance *vs.* the estimated performance on the validation set for the FFD off-line bin packing algorithm with two bin sizes.

## 7. Performance models and the algorithm selection problem

The process of solving a problem often starts with looking at the different procedures available for the job and deciding which one to apply. In automated problem solving this is known as the *algorithm selection problem* [25]. Here we consider the applicability of our models in the context of the algorithm selection problem. We are interested in this because techniques to solve the problem rely on some form of prediction of algorithm performance, and, thus, they bear some similarities with our approach. However, as we will highlight below, there are also important differences.

### *7.1. Algorithm selection problem*

Most approaches for solving the algorithm selection problem require the same set of ingredients [82]: (1) a large collection of problem instances of variable complexity; (2) a diverse set of algorithms, each having the best performance on some such instances; (3) a measure to evaluate the performance of algorithms; and (4) a set of features that describe the properties of the problems in the collection. These elements are necessary because most approaches use machine learning techniques to *predict* which algorithm from the collection will have the *best performance* starting from a description of the problem to be solved. Of course, for this to work, the features in (4) must be such that algorithms have similar performance on problems with similar features.

The methods used to solve the algorithm selection problem can be divided into two groups, depending on when the decision on the strategy to use to solve a problem is made. In *dynamic selection* the decision is made during the run of an algorithm, while in *static selection* the decision is taken before the search starts.

One algorithm that uses a prediction model to guide the search process dynamically is STAGE [83]. STAGE is effectively a hill climber with intelligent restarts. After each hill climber search, STAGE uses the points sampled to produce or update the prediction model (linear regression) of the heuristic. The heuristic then suggests a promising point where to restart the hill climber. STAGE was successfully tested on SAT problems. Note that the models constructed during a run of STAGE can only predict performance on the problem instance it is trying to solve. Following a similar idea, in [84] a prediction model (linear regression) was used to decide which path to follow in a search tree. The approach was tested on Knapsack problems and set partitioning problems.

In [85] the algorithm selection problem for SAT was modelled as a Markov decision process (MDP). The Davis–Putnam–Logemann–Loveland algorithm was enhanced with this model to decide, for each node of the search tree, which branching rule to use next. Similarly, in [86] a Bayesian model was created to predict the run time of an algorithm and dynamically use this prediction to decide when the search should be restarted. Also, as suggested in [87], when different recursive algorithms are available to solve a problem, one can modify them in such a way that they can dynamically call each other at each recursive call. Using a MDP to decide which function to call gave good results with sorting algorithms and with the order statistic selection problem.

Static-selection methodologies, instead, do not alter the course of an algorithm: they first decide which procedure to use and then wait until the chosen algorithm terminates. A substantial body of work falls in this category which attempts to predict the run-time of algorithms for matrix multiplication [88,89], sorting [90,88,89], solving partial differential equations [90] and signal processing [91]. The objective is choosing the algorithm's most efficient implementation based on factors such as the computer's architecture, the workload, and the instance size. Methodologies for predicting the run-time of SAT solvers based on linear regression [29] and a combination of linear regression and a mixture-of-experts [33–35] have

also been proposed. Techniques based on linear regression were also used in [26,27,32,31,30] to solve the auction winner determination problem while case-based reasoning has been used in constraint programming [92–94].

Algorithm portfolios [26–35] are a particularly important class of static algorithm selection techniques. An *algorithm portfolio* is a collection of algorithms that are run in parallel or in sequence to solve a particular problem. Given a new problem instance, performance models are used to rank algorithms. The top ranking algorithms are then executed, stopping as soon as a solution is found. Because in this way the algorithms used are good matches for the problem instance, the average performance of the portfolio over a set of instances is often better than the average performance of any algorithm in the portfolio. Performance models can also be used to identify sets of particularly hard problems for an algorithm [26,27, 32].

When attacking the algorithm selection problem, normally the features used to describe a problem within a performance model are obtained *by an expert* (perhaps via a careful analysis of the literature on a particular class of problems) and are *problem-specific*. In [95], however, such features were obtained automatically by doing preliminary runs of the algorithms in a collection on a problem instance. Solutions found at different times within runs of the algorithms were collected and then used to form a feature set. Following a similar approach, in [96] preliminary runs were used to measure the expected time that the algorithm would spend in processing a node in a search tree. Using Knuth's method for estimating the size of a search tree, this made it possible to predict the run-time of different algorithms.

### 7.2. Similarities and differences with our performance model

The algorithm selection problem is approached using some form of machine learning technique to predict the performance of a collection of algorithms (and then match problems with algorithms), which is very similar with what we do with our performance models. However, our methodology presents significant differences with respect to prior work on algorithm selection.

Firstly, we focus primarily on program induction and more specifically EPAs (although as we have seen our method extends naturally to other domains). This is an area neglected by prior work on algorithm selection. Secondly, we characterise problems using features that are radically different from those used in prior work. This has mainly focused on the use of problem-specific features that experts in the problem domain consider to be useful to measure the hardness of a problem. These features work well, but their selection require considerable domain knowledge (and effectively relies on previous attempts to characterise hardness). Also, the idea of computing features based on preliminary runs of the algorithms being modelled works well. However, this method produces models that lack generality having been derived for one specific problem instance. Instead, in Eq. (5) *our features simply measure how similar a problem is to a set of reference problems*, automatically identified via LARS and cross-validation. Therefore, the features are generic rather than problem-specific and the models of algorithms we obtain are applicable to whole classes of problems, not single instances. Thirdly, we do not just use performance models to predict performance: we also elicit important knowledge on the similarities and differences between algorithms from such models (as we will show in Section 8).

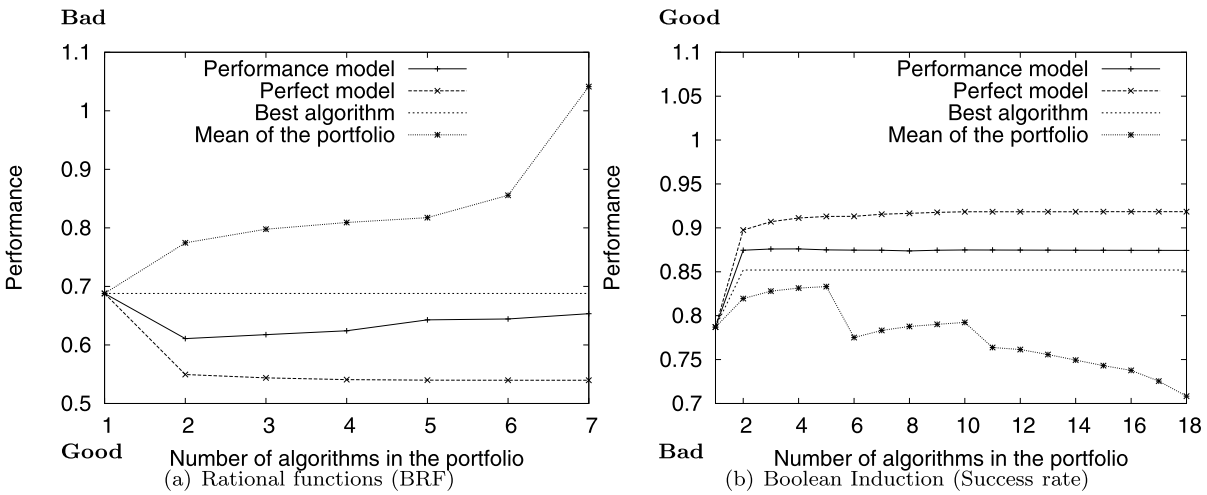
### 7.3. Program-induction portfolios

Despite some good attempts (see Section 2), so far researchers have had relatively little success in the practical characterisation of the difficulty of program induction. This has effectively prevented the extension of the work on portfolios to such a domain. The good results obtained with our models, however, suggest that they might allow such an extension. We, thus, decided to attempt to develop algorithm portfolios for symbolic regression and Boolean induction.

As suggested in [26], the algorithms forming a portfolio should behave differently on different problems. Also, each algorithm should beat all other algorithms in the portfolio on some problems. So, we decided to form portfolios using a subset of the 20 program induction algorithms considered in Section 6. To determine which algorithms to consider for insertion in the portfolio we looked at performance on the problems in the training set and considered only the algorithms that had best performance in at least one problem. This resulted in the exclusion of 13 algorithms for symbolic regression and 2 algorithms for Boolean induction.

To decide which of the remaining algorithms to include in the portfolio, we used a cross-validation technique equivalent to the one used in Section 3.4. We started by creating a portfolio having only the algorithm that resulted best in the biggest number of problems in the training set. Then we added the algorithm that was overall second on the training set. We used the predictions made in the cross-validation to decide which of the two algorithms should be used for each of the problems in the training set, we simulated running the predicted best algorithm on each problem, and averaged the resulting performance values to estimate the performance of the portfolio on the training set.[11] We then added the third, fourth, etc. best algorithm to the portfolio, repeating the phases above until all the algorithms under consideration were included. This whole procedure was repeated for each of the closeness measures, $d$, described in Table 4 plus the Sokal–Sneath similarity measure (which for steady-state GEP gave considerably better results than any other function). This allowed us to select both the portfolios and the $d$ functions with the lowest RSEs.

[11] Since we had already run all algorithms on all problems to create our training set, this phase amounted to a simple look up operation.




(a) Rational functions (BRF) (b) Boolean Induction (Success rate)

**Fig. 4.** Performance of the algorithm portfolio (on the training set), when different criteria are used to select algorithms. The "performance model" uses cross-validation on the training set to choose algorithms. The "perfect model" decides based on the performance measured in actual runs, and the "best model" is the algorithm having the best measured average performance across all problems in the training set.

Fig. 4 illustrates the performance of the algorithm portfolios constructed using this methodology. The "performance model" curves show the portfolio's performance when cross-validation is used in conjunction with our model to decide which algorithm from a portfolio to use for each problem in the training set. The "perfect model" curves show the portfolio's performance that would be obtained if a perfect model of performance was available. The "best algorithm" curves present the performance of a portfolio when we always select the algorithm with the best average performance across the training set. For reference we also provide "mean of the portfolio" curves obtained by averaging the performance of the algorithms composing a portfolio, which indicate the performance we should expect if the choice of algorithm within a portfolio was random.

Since we have ordered algorithms based on their performance in the training set, it is not surprising to see that as we add more and more algorithms to portfolios the average portfolio's performance represented by the "mean of the portfolio" curves decreases. Looking at the "performance model" curves, in all cases the portfolios based on our model have better performance than the best algorithm. As we increase the number of algorithms, the portfolio's performance rapidly peaks and then either remains stable (as for Boolean induction) or slightly decreases (as for symbolic regression).[12] This may be slightly surprising: one might have expected that the more algorithms are available in a portfolio the higher the chances of finding a good match for a problem and, thus, the better the performance. The reason why the "performance model" curves peak at some portfolio size is simply that the gains provided by a larger portfolio in terms of an increased ability to match algorithms to problem are offset by an increasing risk of making incorrect decisions due to there being more chances of picking a sub-optimal algorithm. So, one really needs to compare the "performance model" and "mean of the portfolio" curves. Such a comparison reveals that the improvement in selection ability provided by our models is very significant irrespective of the size of the portfolio and, in fact, increases as the selection problem becomes harder.

We tested the portfolios identified via cross-validation on the validation set to see if they generalised well. As shown in Fig. 5, the portfolio performance obtained by choosing algorithms using our models is only second to that achievable by a perfect model, which, of course, we cannot expect to ever match. However, for both symbolic regression and Boolean induction problems, the pairwise differences in performance between selecting algorithms using our performance models and using the best algorithm in the portfolio are statistically significant, the one-sided, two-sample Kolmogorov–Smirnov test reporting $p$ values $< 0.01$.

## 8. Eliciting knowledge from performance models

### *8.1. Comparing algorithms and parameter settings*

When considering different algorithms to solve a problem it is important to understand the similarities and differences in the behaviour of such algorithms. It is reasonable to attempt to infer some such similarities and differences via a comparison between performance models.

[12] The best performance for symbolic regression of rational functions was obtained when the portfolio is composed by just two algorithms (the TinyGP system with 100% mutation and the SIHC with 25,000 maximum mutations and sub-tree mutation) while, for Boolean induction, the best portfolio included four algorithms (the three TinyGP systems for which $p_{xo} + p_m = 100\%$ and the steady state GEP system).

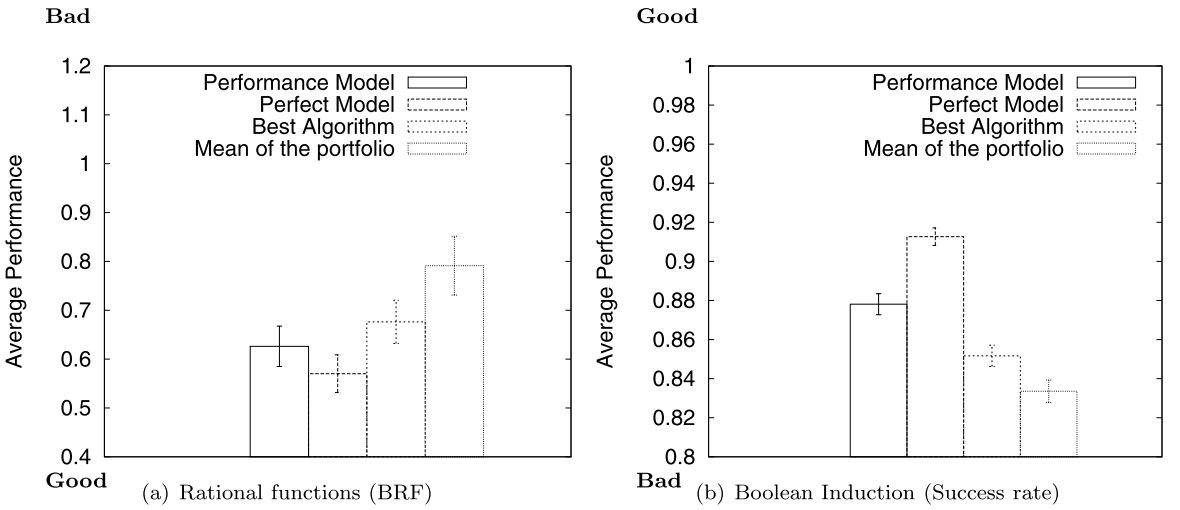



**Fig. 5.** Mean performance of our algorithm portfolios on the validation set. Error bars represent the standard error of the mean.

We start by noting that Eq. (5) represents a hyperplane as clarified by rewriting it in the normal form $(-1, a_{\mathbf{p}_1}, \ldots, a_{\mathbf{p}_{|\mathbb{S}|}}) \cdot ((P(\mathbf{t}), d(\mathbf{p}_1, \mathbf{t}), \ldots, d(\mathbf{p}_{|\mathbb{S}|}, \mathbf{t})) - \mathbf{x}_0) = 0$, where $\cdot$ is the scalar product, $\mathbf{p}_1, \ldots, \mathbf{p}_{|\mathbb{S}|}$ are the elements of $\mathbb{S}$ and $\mathbf{x}_0 = (a_0, 0, \ldots, 0)$ is a point on the hyperplane which depends only on the term $a_0$. Then, one could measure the similarity between an algorithm characterised by the vector $\mathbf{n}' = (-1, a'_{\mathbf{p}_1}, \ldots, a'_{\mathbf{p}_{|\mathbb{S}|}})$ and one represented by the vector $\mathbf{n}'' = (-1, a''_{\mathbf{p}_1}, \ldots, a''_{\mathbf{p}_{|\mathbb{S}|}})$ by simply computing the angle between them, $\alpha = \arccos(\frac{\mathbf{n}' \cdot \mathbf{n}''}{\|\mathbf{n}'\| \|\mathbf{n}''\|})$.

Unfortunately, since the sets $\mathbb{S}$ are independently chosen in different models, in principle the $i$-th coefficient of one model's hyperplane might be associated to a problem, while the $i$-th coefficient in another model's hyperplane might be associated to a different problem. To circumvent this problem we used the following slightly more sophisticated procedure. Let $\mathbb{S}'$ and $\mathbb{S}''$ be the sets of reference vectors associated with two performance models we want to compare. The models also include two corresponding sets of coefficients $a'_{\mathbf{p}_1}, \ldots, a'_{\mathbf{p}_{|\mathbb{S}'|}}$ and $a''_{\mathbf{p}_1}, \ldots, a''_{\mathbf{p}_{|\mathbb{S}''|}}$, respectively. We construct two new sets of coefficients: one, which we will call $b'_{\mathbf{p}_1}, \ldots, b'_{\mathbf{p}_{|\mathbb{S}''|}}$, is obtained by re-running linear regression on the training set associated with the first model but using the reference vectors $\mathbb{S}''$ (this is why the subscripts range from 1 to $|\mathbb{S}''|$); the second, $b''_{\mathbf{p}_1}, \ldots, b''_{\mathbf{p}_{|\mathbb{S}'|}}$, is obtained symmetrically. We then define the vectors $\mathbf{a}' = (-1, a'_{\mathbf{p}_1}, \ldots, a'_{\mathbf{p}_{|\mathbb{S}'|}})$, $\mathbf{a}'' = (-1, a''_{\mathbf{p}_1}, \ldots, a''_{\mathbf{p}_{|\mathbb{S}''|}})$, $\mathbf{b}' = (-1, b'_{\mathbf{p}_1}, \ldots, b'_{\mathbf{p}_{|\mathbb{S}''|}})$ and $\mathbf{b}'' = (-1, b''_{\mathbf{p}_1}, \ldots, b''_{\mathbf{p}_{|\mathbb{S}'|}})$. While $\mathbf{a}'$ and $\mathbf{a}''$ are not comparable, and so are $\mathbf{b}'$ and $\mathbf{b}''$, we can compare $\mathbf{a}'$ with $\mathbf{b}''$ and $\mathbf{a}''$ with $\mathbf{b}'$. Thus, we define the *angle (or dissimilarity) between two algorithms* as $\alpha = \frac{1}{2}[\arccos(\frac{\mathbf{a}' \cdot \mathbf{b}''}{\|\mathbf{a}'\| \|\mathbf{b}''\|}) + \arccos(\frac{\mathbf{a}'' \cdot \mathbf{b}'}{\|\mathbf{a}''\| \|\mathbf{b}'\|})]$.

If the angle between two algorithms is small, the algorithms can reasonably be expected to produce similar performance across all problems. If one algorithm succeeds on a problem, the other will likely succeed and *vice versa*. So, one might decide to favour the faster algorithm. If the angle between two algorithms is big, then we can expect that at least on some problems the performance of the two algorithms differs. Upon failure to solve a problem with one algorithm, one could then have some hope to solve it with the other.

### 8.2. Toward automated taxonomies

In the presence of more than two algorithms, we can build a matrix collecting the angles between all pairs of algorithms under consideration and infer useful information on their mutual relationships. However, when considering many algorithms, this comparison matrix is very large and manually finding interesting patterns in it may be difficult. Here we propose a simple automated procedure which can aid and complement such a manual analysis. To exemplify the approach, we will focus on the 20 GP, GEP and SIHC systems presented in Section 5.

We start by feeding the pair-wise comparison matrix into a *clustering algorithm* to group systems based on the similarity of their performance. More specifically, we adapted the hierarchical clustering algorithm of [97] to create the clusters. The algorithm works as follows. Firstly, we chose the following similarity measure for pairs of clusters: $s(\mathcal{X}, \mathcal{Y}) = \frac{\sum_{i \in \mathcal{X}} \sum_{j \in \mathcal{Y}} \mathcal{M}(i,j)}{|\mathcal{X}||\mathcal{Y}|}$, where $\mathcal{M}$ is the matrix containing the average similarity between all pairs of algorithms under study, $|\cdot|$ denotes the number of elements in a cluster and $\mathcal{X}$ and $\mathcal{Y}$ are clusters. More specifically, $\mathcal{M} = \frac{1}{3}(\mathcal{M}_{BRF} + \mathcal{M}_{NBRF} + \mathcal{M}_B)$ where the matrices $\mathcal{M}_{BRF}$, $\mathcal{M}_{NBRF}$ and $\mathcal{M}_B$ were obtained by performing pair-wise comparisons between our 20 program induction systems for each of our three different performance measures (using the best $d$ functions associated to each). Then, we performed

the following three steps: (1) each system was assigned to a separate cluster; (2) a new cluster was created by merging the two closest clusters based on $s(\mathcal{X}, \mathcal{Y})$, thereby reducing the number of clusters by one; (3) we repeated step (2) until there was only one cluster left.

This resulted in a cluster hierarchy with 20 levels. To simplify interpretation, we decided to focus only its 8 topmost clusters. To visualise such clusters we treated them as nodes in a fully-connected graph and we used the graph-drawing package `neato`, which is part of the `GraphViz` library, to obtain and draw a graph layout where pairs of nodes corresponding to clusters with high similarity were placed closer to each other than clusters corresponding to systems that were dissimilar performance-wise.

The strategy used by `neato` to position the nodes of a graph is to interpret them as physical bodies connected by springs (the edges in the graph). The user can set both the springs' stiffness and their rest length. To produce a layout, the (virtual) physical system is initialised in some suboptimal configuration, which is then iteratively modified until the system relaxes into a state of minimal energy. In this work, we associated to each edge a length proportional to $s(\mathcal{X}, \mathcal{Y})$, so that the nodes would be pushed apart when producing the graph's layout proportionally to their dissimilarity. We also set the stiffness of springs using the formula $1/(0.01 + s(\mathcal{X}, \mathcal{Y}))$.

Fig. 6 shows the output produced by `neato`. The edges between *clusters* in this figure are represented using dashed lines, while edges connecting *systems* to their mother cluster are drawn with solid lines. We can think of this diagram as a *taxonomy* of the systems under study. To distinguish between different forms of selection, reproduction and type of mutation we used different symbols, as indicated at the bottom left of the figure.

From the figure we can see how the steady state GP systems are grouped in a cluster (left of the figure). The generational GP systems with tournament selection are arranged in another cluster (bottom left). At the top, we can find the cluster containing all the SIHCs with sub-tree mutation. The generational GP systems with roulette-wheel selection are grouped according to whether crossover or mutation is the dominant operator. More specifically, at the bottom of the figure we find a cluster containing the generational GP systems with no mutation. Just above it is a cluster with the generational GP systems with 50% and 100% mutation. Finally, somehow surprisingly, each of the GEP systems was placed in a separate cluster (middle and right of the figure) indicating that these systems are very different performance-wise.

Overall, our taxonomy suggests that the type of selection and reproduction used have a bigger impact on performance than crossover and mutation rates. This is evident from the clusters formed by the steady state GP systems, the generational GP systems with tournament selection, and the generational GP systems with roulette-wheel selection. The taxonomy also suggests that the reproduction strategy is very important in determining the behaviour of GEP systems. For SIHCs, the taxonomy indicates that the type of mutation used is more important than the maximum number of mutations. Surprisingly, the taxonomy also suggests that the SIHC with uniform mutation is very similar to generational GP systems with tournament selection, which is something one could hardly guess by looking at structural similarities between these algorithms.[13]

Given that systems were grouped based on performance similarity, it is reasonable to expect that if a particular system consistently fails to solve a problem, it will be more efficient to try one or more alternative systems *from a different cluster*, rather than finely optimise the parameters of the first system. This should be done to further improve performance once a satisfactory system is found. For the same reasons, perhaps in algorithm portfolios one should not just pick the best $n$ algorithms, but also look at how independent the performance of such algorithms is as portfolios with a good coverage of the performance space might be expected to generalise better.

### 8.3. What knowledge can we extract from measuring performance empirically?

In this section, we compare what we have learnt from analysing our performance models with what users of program induction systems might be able to learn by using traditional approaches. These typically consist of computing some performance statistics on sets of test problems with the systems and parameter settings under comparison over a number of independent runs.

We followed this approach to construct in Table 8, which reports the performance of the GP, GEP and SIHC systems and parameter settings considered in this paper on the rational-function and the Boolean-function testbeds. Statistics were collected by running each system on the 1100 different problems in the training and validation sets for each problem class. Performance was estimated by averaging results of 100 independent runs. This required a total of 4,400,000 runs.

As can be seen from the table, SIHC with sub-tree mutation and a maximum of 25,000 mutations between restarts has the best performance on the rational functions problems irrespective of whether we used the BRF or the NBRF measures. TinyGP with 100% crossover has the best performance on Boolean induction problems. Also, there are large performance differences between roulette-wheel selection and tournament selection and between generational and steady state systems. These are statistically significant. For the generational GP system in the rational problems the mean and standard deviation of the performance measure decrease as the mutation rate increases, suggesting that there might be differences in behaviour between the high-mutation and high-crossover search modes. However, all differences in performance observed when crossover and mutation rates are varied are not statistically significant.

[13] Whether these similarities and differences are present only in the classes of problems considered here and their origin is something that needs to be explored in future research, possibly using alternative means.

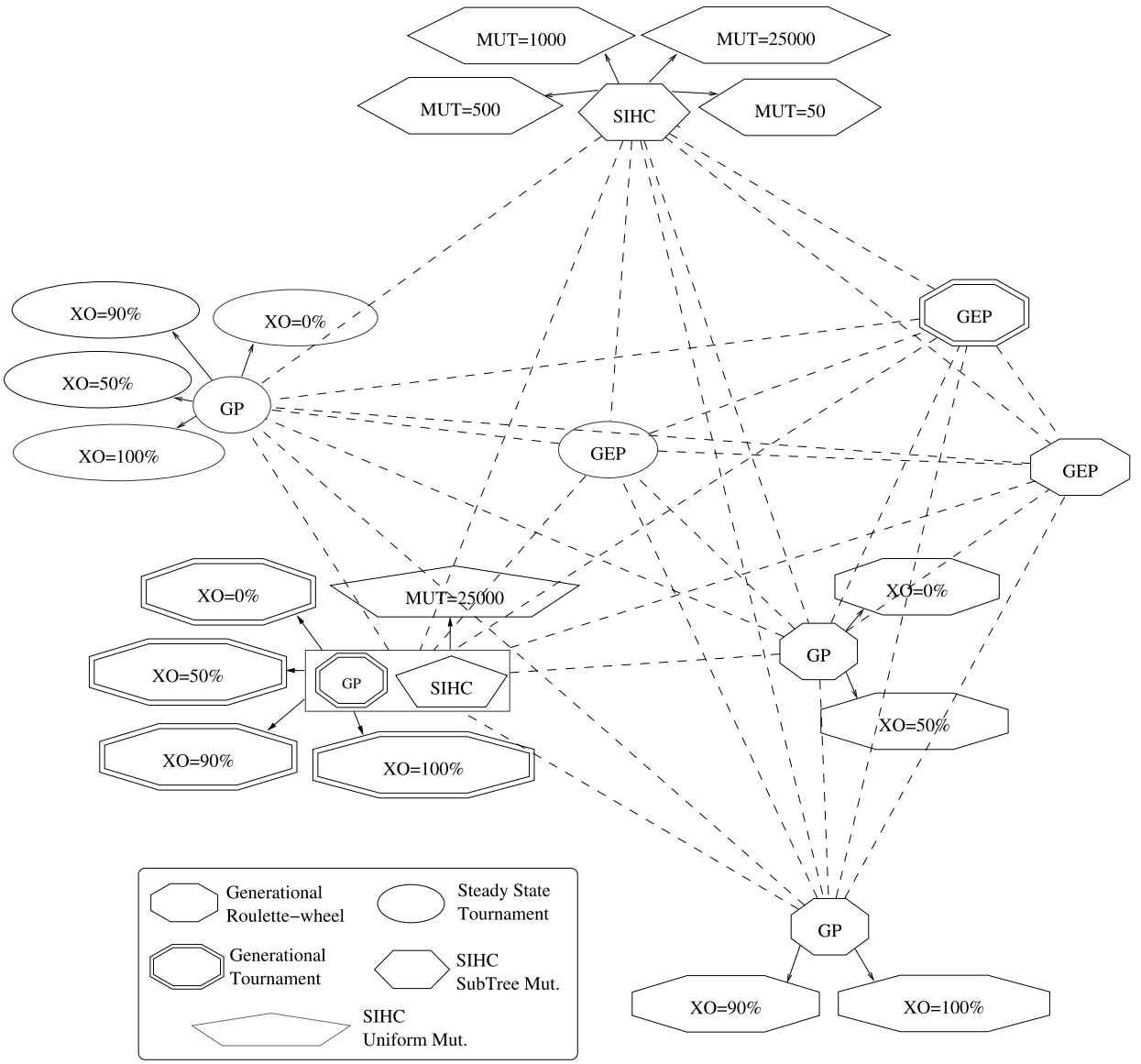



**Fig. 6.** Taxonomy of GP, GEP and SIHC systems with different parameter settings based on performance models.

In other words, like the analysis based on our performance models, the data suggest that for the systems and problems studied, changing the selection mechanism has a bigger effect than varying the crossover and mutation rates. However, there are a variety of other phenomena that we were able to capture (see Section 8.2) that simple performance statistics cannot reveal.

This does not mean, of course, that the information provided by our models cannot be corroborated empirically. It can, but this may require targeted empirical analyses. To illustrate this, in Table 9 we report the average Pearson's correlation coefficients obtained by comparing the performance results of pairs of program-induction algorithms on the same problems and with the same performance measures used in Table 8. Careful inspection of the table confirms *all* the relationships highlighted by our taxonomy. This includes the unexpected finding that generational systems with tournament selection are more similar to steady state systems with tournament selection than to generational systems with roulette-wheel selection and that SIHC with uniform mutation is similar to generational GP systems with tournament selection and quite different from the SIHC with sub-tree mutation. Why cannot this be inferred from Table 8? Simple: if one system does well on a subset of problems and not so well on another subset while another does the opposite, means and standard deviations of performance may not be able tell such systems apart.

**Table 8**
Standard experimental results with the GP, GEP, and SIHC systems under study.

| Configuration | | | | Rational functions | | | | Boolean functions | |
|---|---|---|---|---|---|---|---|---|---|
| Type | Selection | $p_{xo}$ | $p_m$ | Best of run fitness | | Normalised BRF | | Success rate | |
| | | | | Mean | Std. Dev. | Mean | Std. Dev. | Mean | Std. Dev. |
| Generational | Roulette | 1.00 | 0.00 | 6.5635 | 27.4912 | 0.4658 | 0.1146 | 0.6554 | 0.2374 |
| | | 0.90 | 0.00 | 6.6153 | 28.0177 | 0.4652 | 0.1150 | 0.6575 | 0.2386 |
| | | 0.50 | 0.50 | 4.7502 | 17.6580 | 0.4516 | 0.1176 | 0.6693 | 0.2326 |
| | | 0.00 | 1.00 | 4.2718 | 15.3276 | 0.4462 | 0.1189 | 0.6891 | 0.2237 |
| | | GEP | | 6.4979 | 20.5852 | 0.4816 | 0.1109 | 0.4869 | 0.3573 |
| Generational | Tournament | 1.00 | 0.00 | 2.5828 | 5.8401 | 0.4003 | 0.1139 | 0.8136 | 0.1671 |
| | | 0.90 | 0.00 | 2.5341 | 5.7355 | 0.3977 | 0.1141 | 0.8094 | 0.1720 |
| | | 0.50 | 0.50 | 2.3552 | 5.4806 | 0.3916 | 0.1140 | 0.8192 | 0.1618 |
| | | 0.00 | 1.00 | 2.2864 | 5.3267 | 0.3878 | 0.1126 | 0.8327 | 0.1504 |
| | | GEP | | 4.1644 | 10.7753 | 0.4515 | 0.1128 | 0.4983 | 0.3735 |
| Steady state | Tournament | 1.00 | 0.00 | 0.8576 | 1.7970 | 0.2535 | 0.0860 | **0.8518** | 0.1333 |
| | | 0.90 | 0.00 | 0.8720 | 1.7886 | 0.2535 | 0.0857 | 0.8416 | 0.1375 |
| | | 0.50 | 0.50 | 0.8682 | 1.8666 | 0.2511 | 0.0817 | 0.8455 | 0.1329 |
| | | 0.00 | 1.00 | 0.8856 | 1.9348 | 0.2494 | 0.0800 | 0.8437 | 0.1320 |
| | | GEP | | 2.1178 | 4.5666 | 0.3580 | 0.1101 | 0.7894 | 0.2445 |
| Sys. | Mut. | Max Mut. | | Mean | Std. Dev. | Mean | Std. Dev. | Mean | Std. Dev. |
| SIHC | Sub-tree | 50 | | 1.2440 | 2.5780 | 0.2430 | 0.0789 | 0.4000 | 0.2185 |
| | | 500 | | 1.1943 | 2.4902 | 0.2419 | 0.0785 | 0.4143 | 0.2265 |
| | | 1000 | | 1.0773 | 2.4326 | 0.2328 | 0.0748 | 0.4298 | 0.2305 |
| | | 25,000 | | **0.6816** | 1.2105 | **0.2021** | 0.0686 | 0.5326 | 0.2460 |
| | Unif. | 25,000 | | 1.4295 | 3.2838 | 0.2978 | 0.0970 | 0.7370 | 0.2317 |

## 9. Conclusions

We presented a set of techniques to build efficient and accurate models of the performance of problem solvers. We modelled three versions of GP with multiple parameter settings, three versions of GEP, two versions of SIHC (one with multiple parameters settings), one ANN learning algorithm, and two bin-packing heuristics. These algorithms were applied to the following problems: symbolic regression of rational functions, Boolean induction, and off-line bin packing.

Many applications are possible for our models. They can be used to determine what is the best algorithm for a problem, as shown in Section 7 where we obtained algorithm portfolios for EPAs which had significantly better average performance than the best overall algorithm in the each portfolio. As we showed in Section 8, they can also be used to reveal similarities and differences between algorithms across whole problem classes and to build highly-informative algorithm taxonomies. Our taxonomy of EPAs, for example, provided numerous new findings, including that the EPAs studied are little sensitive to the choice of genetic operator rates, while reproduction and selection strategies influence performance significantly. Only some of this information can readily be provided by standard empirical analyses, although once interesting relationships have been identified through the use of our models and taxonomies, it is important to mine empirical results (or perform ad-hoc runs) to corroborate such relationships, as we did with Table 9.

A difference between our models and other approaches to modelling EAs is that our models can simply be used to accurately predict the performance of algorithms on *unseen problems* from the class from which the training set was drawn. Other techniques, such as the fitness distance correlation, *fdc*, and the negative slope coefficient, *nsc*, can only predict if a particular problem is hard or easy, but not precisely how hard or easy (see Section 2). Also, our approach allows the user to *choose the performance measure* they want to model, while *fdc* and *nsc* don't.

The execution of our models involves an extremely low computational load. Of course, their instantiation requires running a system, possibly multiple times, on a suitably large training set of problems. However, the cost of this is similar to that required to compare the performance of different algorithms empirically. The difference here is that once reliable models are constructed, they can be used over and over again to test performance on new problems, while empirical testing requires effectively re-running all systems on each new problem to see how they behave.

The main difference between our models and approaches used to solve in the algorithm selection problem is that our models do not require the manual selection of sets of features to describe problems, while in other approaches these features are typically defined by an expert and are problem-specific. Instead, our models use the concept of closeness between the problem for which performance is being estimated and some reference problems previously automatically selected (i.e., the set $\mathbb{S}$). This makes it easy to apply our approach to different classes of problems.

An important similarity between our models and those used in algorithm selection techniques is that they are linear functions of their features. One might wonder why such simple models work so well. Firstly, in both types of models the features used are related to the hardness of the problem. In the case of our models, the $d$ functions measure the similarity (e.g., the square of the dot product) or dissimilarity (e.g., the sum of absolute differences) between the input problem and a

**Table 9**
Pearson's correlation coefficient between the performance results obtained in 1100 symbolic regression problems and 1100 Boolean induction problems averaged across three performance measures (BRF, NBRF and success rate).

| Configuration | | | Generational Roulette | | | | | Generational Tournament | | | | | Steady state Tournament | | | | | SIHC | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | $p_{xo}$ | | | | GEP | $p_{xo}$ | | | | GEP | $p_{xo}$ | | | | GEP | Sub-tree Mut. | | | | Unif. |
| | | | 1.00 | 0.90 | 0.50 | 0.00 | | 1.00 | 0.90 | 0.50 | 0.00 | | 1.00 | 0.90 | 0.50 | 0.00 | | 50 | 500 | 1000 | 25,000 | 25,000 |
| Type | Selection | $p_{xo}$ | | | | | | | | | | | | | | | | | | | | |
| Generational | Roulette | 1.00 | 1.00 | 0.99 | 0.98 | 0.98 | 0.86 | 0.86 | 0.87 | 0.86 | 0.86 | 0.83 | 0.76 | 0.77 | 0.77 | 0.78 | 0.74 | 0.77 | 0.78 | 0.81 | 0.70 | 0.83 |
| | | 0.90 | 0.99 | 1.00 | 0.98 | 0.98 | 0.86 | 0.86 | 0.87 | 0.86 | 0.86 | 0.83 | 0.76 | 0.77 | 0.77 | 0.78 | 0.74 | 0.77 | 0.78 | 0.81 | 0.70 | 0.83 |
| | | 0.50 | 0.98 | 0.98 | 1.00 | 0.99 | 0.87 | 0.89 | 0.89 | 0.89 | 0.89 | 0.85 | 0.80 | 0.80 | 0.80 | 0.81 | 0.77 | 0.80 | 0.81 | 0.84 | 0.73 | 0.87 |
| | | 0.00 | 0.98 | 0.98 | 0.99 | 1.00 | 0.86 | 0.89 | 0.89 | 0.89 | 0.89 | 0.84 | 0.80 | 0.81 | 0.81 | 0.82 | 0.77 | 0.80 | 0.81 | 0.84 | 0.73 | 0.87 |
| | | GEP | 0.86 | 0.86 | 0.87 | 0.86 | 1.00 | 0.78 | 0.78 | 0.77 | 0.77 | 0.97 | 0.69 | 0.69 | 0.69 | 0.70 | 0.86 | 0.71 | 0.72 | 0.74 | 0.64 | 0.75 |
| Type | Selection | $p_{xo}$ | | | | | | | | | | | | | | | | | | | | |
| Generational | Tournament | 1.00 | 0.86 | 0.86 | 0.89 | 0.89 | 0.78 | 1.00 | 0.98 | 0.97 | 0.97 | 0.82 | 0.92 | 0.92 | 0.91 | 0.91 | 0.83 | 0.81 | 0.81 | 0.80 | 0.80 | 0.94 |
| | | 0.90 | 0.87 | 0.87 | 0.89 | 0.89 | 0.78 | 0.98 | 1.00 | 0.97 | 0.97 | 0.82 | 0.92 | 0.92 | 0.92 | 0.91 | 0.84 | 0.81 | 0.82 | 0.80 | 0.80 | 0.94 |
| | | 0.50 | 0.86 | 0.86 | 0.89 | 0.89 | 0.77 | 0.97 | 0.97 | 1.00 | 0.98 | 0.82 | 0.93 | 0.93 | 0.93 | 0.93 | 0.84 | 0.81 | 0.82 | 0.80 | 0.80 | 0.95 |
| | | 0.00 | 0.86 | 0.86 | 0.89 | 0.89 | 0.77 | 0.97 | 0.97 | 0.98 | 1.00 | 0.82 | 0.93 | 0.93 | 0.94 | 0.93 | 0.85 | 0.81 | 0.81 | 0.80 | 0.79 | 0.95 |
| | | GEP | 0.83 | 0.83 | 0.85 | 0.84 | 0.97 | 0.82 | 0.82 | 0.82 | 0.82 | 1.00 | 0.74 | 0.74 | 0.74 | 0.74 | 0.90 | 0.73 | 0.75 | 0.75 | 0.68 | 0.79 |
| Type | Selection | $p_{xo}$ | | | | | | | | | | | | | | | | | | | | |
| Steady state | Tournament | 1.00 | 0.76 | 0.76 | 0.80 | 0.80 | 0.69 | 0.92 | 0.92 | 0.93 | 0.93 | 0.74 | 1.00 | 0.97 | 0.97 | 0.96 | 0.80 | 0.80 | 0.80 | 0.78 | 0.81 | 0.94 |
| | | 0.90 | 0.77 | 0.77 | 0.80 | 0.81 | 0.69 | 0.92 | 0.92 | 0.93 | 0.93 | 0.74 | 0.97 | 1.00 | 0.97 | 0.96 | 0.79 | 0.80 | 0.80 | 0.79 | 0.81 | 0.94 |
| | | 0.50 | 0.77 | 0.77 | 0.80 | 0.81 | 0.69 | 0.91 | 0.92 | 0.93 | 0.94 | 0.74 | 0.97 | 0.97 | 1.00 | 0.97 | 0.80 | 0.78 | 0.78 | 0.77 | 0.78 | 0.93 |
| | | 0.00 | 0.78 | 0.78 | 0.81 | 0.82 | 0.70 | 0.91 | 0.91 | 0.93 | 0.93 | 0.74 | 0.96 | 0.96 | 0.97 | 1.00 | 0.80 | 0.77 | 0.77 | 0.76 | 0.76 | 0.92 |
| | | GEP | 0.74 | 0.74 | 0.77 | 0.77 | 0.86 | 0.83 | 0.84 | 0.84 | 0.85 | 0.90 | 0.80 | 0.79 | 0.80 | 0.80 | 1.00 | 0.70 | 0.71 | 0.70 | 0.68 | 0.82 |
| Sys. | Mut. | Max Mut. | | | | | | | | | | | | | | | | | | | | |
| SIHC | Sub-tree | 50 | 0.77 | 0.77 | 0.80 | 0.80 | 0.71 | 0.81 | 0.81 | 0.81 | 0.81 | 0.73 | 0.80 | 0.80 | 0.78 | 0.77 | 0.70 | 1.00 | 0.96 | 0.96 | 0.95 | 0.84 |
| | | 500 | 0.78 | 0.78 | 0.81 | 0.81 | 0.72 | 0.81 | 0.82 | 0.82 | 0.81 | 0.75 | 0.80 | 0.80 | 0.78 | 0.77 | 0.71 | 0.96 | 1.00 | 0.96 | 0.93 | 0.84 |
| | | 1000 | 0.81 | 0.81 | 0.84 | 0.84 | 0.74 | 0.80 | 0.80 | 0.80 | 0.80 | 0.75 | 0.78 | 0.79 | 0.77 | 0.76 | 0.70 | 0.96 | 0.96 | 1.00 | 0.92 | 0.84 |
| | | 25,000 | 0.70 | 0.70 | 0.73 | 0.73 | 0.64 | 0.80 | 0.80 | 0.80 | 0.79 | 0.68 | 0.81 | 0.81 | 0.78 | 0.76 | 0.68 | 0.95 | 0.93 | 0.92 | 1.00 | 0.83 |
| | Unif. | 25,000 | 0.83 | 0.83 | 0.87 | 0.87 | 0.75 | 0.94 | 0.94 | 0.95 | 0.95 | 0.79 | 0.94 | 0.94 | 0.93 | 0.92 | 0.82 | 0.84 | 0.84 | 0.84 | 0.83 | 1.00 |

set of reference problems. So, by properly setting the magnitude and sign of the coefficients in Eq. (5), linear regression can create a scheme by which similarity with a difficult reference problem leads to reducing the performance estimate and *vice versa*. Secondly, while our (and other) models are linear in their features, such features are typically non-linear functions of the degrees of freedom in the problem's representation. The same architecture is used in some types of multi-layer perceptrons and radial-basis neural networks, which are powerful function approximators. So, it is not entirely surprising that our models can fit performance functions well.

Finally, we would like to briefly discuss possible future research avenues. Our approach is generally able to make accurate predictions on unseen problems from the class from which the training set was drawn. However, as we have seen in the case of ANN learning in the presence of rare problems requiring long training times, if new problems are not sufficiently similar to any of the problems in the training set, model predictions can significantly deviate from actual performance. In this paper we have not studied the problem of outliers in depth. By their own nature, outliers are rare and, thus, obtaining statistically meaningful results on model outliers would require an enormous computational effort. We expect outliers to be generated via two mechanisms: either a problem is very different (as assessed by the similarity measure $d$ used to build a model) from all the problems in the reference set $\mathbb{S}$ and/or a problem falls in an region of the performance function which presents rapid changes (or discontinuities as in a phase transition) which cannot be well modelled with the simple kernel provided by the $d$ function. In future work we will investigate the possibility of detecting such cases to inform the user that the model might make incorrect predictions and/or to take counter measures.

Sizing the population is a major step in all population-based algorithms. So, in future research we also intend to apply our models to look at how population sizes influence performance. Also, as shown in Section 6.1, some closeness measures produce better models than others. It is possible that there exist even better measures than the ones we settled for in this paper. In future research we want to use GP itself to obtain even more predictive closeness measures. Furthermore, in this paper we used angles to measure the similarity between the models of different algorithms. In the future, we want to explore different ways of measuring such a similarity (e.g., via distances) since there is hope these may reveal even finer details. Also, introducing regularisation terms is a standard technique to transform ill-posed problems into well-posed ones. In future research we will explore the benefits of adding such terms (e.g., that the sum $\sum_{\mathbf{p}} a_{\mathbf{p}}^2$ be minimised) to least squares. Finally, we want to explore whether there are some performance measures and problem classes for which the approach is particularly suitable or unsuitable and why.

## Acknowledgements

We would like to thank the editor, associate editor and reviewers for their fair and useful comments and ideas. The paper has been considerably strengthened thanks to their feedback.

The first author acknowledges support from the National Council of Science and Technology (CONACyT) of Mexico to pursue graduate studies at the University of Essex.

## References

[1] J.H. Holland, Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, USA, 1975.
[2] D.E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison–Wesley, 1989.
[3] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, 2nd edition, Springer, Berlin, 1994.
[4] M. Mitchell, An Introduction to Genetic Algorithms, MIT Press, Cambridge, MA, 1996.
[5] T. Bäck, D.B. Fogel, Z. Michalewicz (Eds.), Evolutionary Computation 1: Basic Algorithms and Operators, Institute of Physics Publishing, 2000.
[6] J.R. Koza, Genetic Programming: On the Programming of Computers by Natural Selection, MIT Press, Cambridge, MA, USA, 1992.
[7] D.B. Fogel (Ed.), Evolutionary Computation. The Fossil Record. Selected Readings on the History of Evolutionary Computation, IEEE Press, 1998.
[8] A.E. Nix, M.D. Vose, Modeling genetic algorithms with Markov chains, Annals of Mathematics and Artificial Intelligence 5 (1992) 79–88.
[9] M.D. Vose, The Simple Genetic Algorithm: Foundations and Theory, MIT Press, Cambridge, MA, 1999.
[10] T.E. Davis, J.C. Principe, A Markov chain framework for the simple genetic algorithm, Evolutionary Computation 1 (3) (1993) 269–288.
[11] G. Rudolph, Convergence analysis of canonical genetic algorithm, IEEE Transactions on Neural Networks 5 (1) (1994) 96–101.
[12] C.R. Stephens, H. Waelbroeck, Schemata evolution and building blocks, Evolutionary Computation 7 (2) (1999) 109–124.
[13] J. He, X. Yao, Drift analysis and average time complexity of evolutionary algorithms, Artificial Intelligence 127 (1) (2001) 57–85.
[14] R. Poli, Exact schema theory for genetic programming and variable-length genetic algorithms with one-point crossover, Genetic Programming and Evolvable Machines 2 (2) (2001) 123–163.
[15] W.B. Langdon, R. Poli, Foundations of Genetic Programming, Springer, 2002.
[16] J. He, X. Yao, Towards an analytic framework for analysing the computation time of evolutionary algorithms, Artificial Intelligence 145 (1–2) (2003) 59–97.
[17] R. Poli, N.F. McPhee, General schema theory for genetic programming with subtree-swapping crossover: Part II, Evolutionary Computation 11 (2) (2003) 169–206.
[18] R. Poli, N.F. McPhee, J.E. Rowe, Exact schema theory and Markov chain models for genetic programming and variable-length genetic algorithms with homologous crossover, Genetic Programming and Evolvable Machines 5 (1) (2004) 31–70.
[19] R. Poli, W.B. Langdon, N.F. McPhee, A field guide to genetic programming, published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk, 2008 (with contributions by J.R. Koza).
[20] J.F. Miller, P. Thomson, Cartesian genetic programming, in: R. Poli, W. Banzhaf, W.B. Langdon, J.F. Miller, P. Nordin, T.C. Fogarty (Eds.), Proceedings of the Third European Conference on Genetic Programming (EuroGP-2000), in: LNCS, vol. 1802, Springer-Verlag, Edinburgh, 2000, pp. 121–132.
[21] M. O'Neill, C. Ryan, Grammatical evolution, IEEE Transactions on Evolutionary Computation 5 (4) (2001) 349–358.
[22] C. Ferreira, Gene expression programming: A new adaptive algorithm for solving problems, Complex Systems 13 (2) (2001) 87–129.
[23] R. Poli, C.R. Stephens, The building block basis for genetic programming and variable-length genetic algorithms, International Journal of Computational Intelligence Research 1 (2) (2005) 183–197, invited paper.

[24] R. Poli, Y. Borenstein, T. Jansen, Editorial for the special issue on: Evolutionary algorithms – bridging theory and practice, Evolutionary Computation 15 (4) (2007) iii–v.
[25] J.R. Rice, The algorithm selection problem, Advances in Computers 15 (1976) 65–118.
[26] K. Leyton-Brown, E. Nudelman, Y. Shoham, Empirical hardness models: Methodology and a case study on combinatorial auctions, J. ACM 56 (4) (2009) 1–52.
[27] K. Leyton-Brown, E. Nudelman, Y. Shoham, Empirical hardness models for combinatorial auctions, in: P. Cramton, Y. Shoham, R. Steinberg (Eds.), Combinatorial Auctions, MIT Press, 2006, pp. 479–504 (Ch. 19).
[28] F. Hutter, Y. Hamadi, H.H. Hoos, K. Leyton-Brown, Performance prediction and automated tuning of randomized and parametric algorithms, in: F. Benhamou (Ed.), CP, in: Lecture Notes in Computer Science, vol. 4204, Springer, 2006, pp. 213–228.
[29] E. Nudelman, K. Leyton-Brown, H.H. Hoos, A. Devkar, Y. Shoham, Understanding random SAT: Beyond the clauses-to-variables ratio, in: Wallace [100], pp. 438–452.
[30] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, Y. Shoham, Boosting as a metaphor for algorithm design, in: F. Rossi (Ed.), CP, in: Lecture Notes in Computer Science, vol. 2833, Springer, 2003, pp. 899–903.
[31] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, Y. Shoham, A portfolio approach to algorithm selection, in: G. Gottlob, T. Walsh (Eds.), IJCAI, Morgan Kaufmann, 2003, pp. 1542–1543.
[32] K. Leyton-Brown, E. Nudelman, Y. Shoham, Learning the empirical hardness of optimization problems: The case of combinatorial auctions, in: P.V. Hentenryck (Ed.), CP, in: Lecture Notes in Computer Science, vol. 2470, Springer, 2002, pp. 556–572.
[33] L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown, SATzilla: portfolio-based algorithm selection for SAT, Journal of Artificial Intelligence Research 32 (2008) 565–606.
[34] L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown, SATzilla-07: The design and analysis of an algorithm portfolio for SAT, in: Bessiere [101], pp. 712–727.
[35] L. Xu, H.H. Hoos, K. Leyton-Brown, Hierarchical hardness models for SAT, in: Bessiere [101], pp. 696–711.
[36] K. Deb, D.E. Goldberg, Analyzing deception in trap functions, in: L.D. Whitley (Ed.), Foundations of Genetic Algorithms Workshop, vol. 2, Morgan Kaufmann, 1993, pp. 93–108.
[37] M. Mitchell, S. Forrest, J.H. Holland, The royal road for genetic algorithms: fitness landscapes and ga performance, in: F.J. Varela, P. Bourgine (Eds.), Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life, The MIT Press, 1992, pp. 245–254.
[38] S.J. Wright, The roles of mutation, inbreeding, crossbreeding and selection in evolution, in: D.F. Jones (Ed.), Proceedings of the Sixth International Congress on Genetics, vol. 1, 1932, pp. 356–366.
[39] J. Horn, D.E. Goldberg, Genetic algorithm difficulty and the modality of the fitness landscapes, in: L.D. Whitley, M.D. Vose (Eds.), Foundations of Genetic Algorithms Workshop, vol. 3, Morgan Kaufmann, 1995, pp. 243–269.
[40] S.A. Kauffman, S. Johnsen, Coevolution to the edge of chaos: Coupled fitness landscapes, poised states, and coevolutionary avalanches, Journal of Theoretical Biology 149 (4) (1991) 467–505.
[41] T. Jones, S. Forrest, Fitness distance correlation as a measure of problem difficulty for genetic algorithms, in: L.J. Eshelman (Ed.), ICGA, Morgan Kaufmann, 1995, pp. 184–192.
[42] M. Clergue, P. Collard, M. Tomassini, L. Vanneschi, Fitness distance correlation and problem difficulty for genetic programming, in: W.B. Langdon, E. Cantú-Paz, K.E. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M.A. Potter, A.C. Schultz, J.F. Miller, E.K. Burke, N. Jonoska (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002), Morgan Kaufmann, New York, USA, July 2002, pp. 724–732.
[43] L. Vanneschi, M. Tomassini, P. Collard, M. Clergue, Fitness distance correlation in structural mutation genetic programming, in: C. Ryan, T. Soule, M. Keijzer, E.P.K. Tsang, R. Poli, E. Costa (Eds.), EuroGP, in: Lecture Notes in Computer Science, vol. 2610, Springer, 2003, pp. 455–464.
[44] L. Vanneschi, M. Tomassini, M. Clergue, P. Collard, Difficulty of unimodal and multimodal landscapes in genetic programming, in: E. Cantú-Paz, J.A. Foster, K. Deb, L. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R.K. Standish, G. Kendall, S.W. Wilson, M. Harman, J. Wegener, D. Dasgupta, M.A. Potter, A.C. Schultz, K.A. Dowsland, N. Jonoska, J.F. Miller (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2003), in: Lecture Notes in Computer Science, vol. 2723, Springer, Chicago, IL, USA, July 2003, pp. 1788–1799.
[45] M. Tomassini, L. Vanneschi, P. Collard, M. Clergue, A study of fitness distance correlation as a difficulty measure in genetic programming, Evolutionary Computation 13 (2) (2005) 213–239.
[46] L. Vanneschi, M. Clergue, P. Collard, M. Tomassini, S. Vérel, Fitness clouds and problem hardness in genetic programming, in: K. Deb, R. Poli, W. Banzhaf, H.-G. Beyer, E.K. Burke, P.J. Darwen, D. Dasgupta, D. Floreano, J.A. Foster, M. Harman, O. Holland, P.L. Lanzi, L. Spector, A. Tettamanzi, D. Thierens, A.M. Tyrrell (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2004), in: Lecture Notes in Computer Science, vol. 3103, Springer, Seattle, WA, USA, June 2004, pp. 690–701.
[47] L. Vanneschi, M. Tomassini, P. Collard, M. Clergue, A survey of problem difficulty in genetic programming, in: S. Bandini, S. Manzoni (Eds.), Advances in Artificial Intelligence: Proceedings of the Ninth Congress of the Italian Association for Artificial Intelligence (AI*IA-2005), in: Lecture Notes in Computer Science, vol. 3673, Springer, Milan, Italy, September 2005, pp. 66–77.
[48] L. Vanneschi, M. Tomassini, P. Collard, S. Vérel, Negative slope coefficient: A measure to characterize genetic programming fitness landscapes, in: P. Collet, M. Tomassini, M. Ebner, S. Gustafson, A. Ekárt (Eds.), EuroGP, in: Lecture Notes in Computer Science, vol. 3905, Springer, 2006, pp. 178–189.
[49] R. Poli, L. Vanneschi, Fitness-proportional negative slope coefficient as a hardness measure for genetic algorithms, in: Lipson [98], pp. 1335–1342.
[50] S. Droste, T. Jansen, I. Wegener, A rigorous complexity analysis of the $(1+1)$ evolutionary algorithm for separable functions with Boolean inputs, Evolutionary Computation 6 (2) (1998) 185–196.
[51] S. Droste, T. Jansen, I. Wegener, On the analysis of the $(1+1)$ evolutionary algorithm, Theoretical Computer Science 276 (1–2) (2002) 51–81.
[52] I. Wegener, On the expected runtime and the success probability of evolutionary algorithms, in: U. Brandes, D. Wagner (Eds.), WG, in: Lecture Notes in Computer Science, vol. 1928, Springer, 2000, pp. 1–10.
[53] T. Jansen, K.A.D. Jong, I. Wegener, On the choice of the offspring population size in evolutionary algorithms, Evolutionary Computation 13 (4) (2005) 413–440.
[54] C. Witt, Runtime analysis of the (mu + 1) ea on simple pseudo-boolean functions, Evolutionary Computation 14 (1) (2006) 65–86.
[55] T. Jansen, I. Wegener, The analysis of evolutionary algorithms – a proof that crossover really can help, Algorithmica 34 (1) (2002) 47–66.
[56] T. Jansen, I. Wegener, Real royal road functions – where crossover provably is essential, Discrete Applied Mathematics 149 (1–3) (2005) 111–125.
[57] O. Giel, I. Wegener, Evolutionary algorithms and the maximum matching problem, in: H. Alt, M. Habib (Eds.), STACS, in: Lecture Notes in Computer Science, vol. 2607, Springer, 2003, pp. 415–426.
[58] F. Neumann, I. Wegener, Randomized local search, evolutionary algorithms, and the minimum spanning tree problem, Theoretical Computer Science 378 (1) (2007) 32–40.
[59] J. Scharnow, K. Tinnefeld, I. Wegener, The analysis of evolutionary algorithms on sorting and shortest paths problems, Journal of Mathematical Modelling and Algorithms 3 (4) (2004) 349–366.
[60] S. Baswana, S. Biswas, B. Doerr, T. Friedrich, P.P. Kurur, F. Neumann, Computing single source shortest paths using single-objective fitness, in: Foundations of Genetic Algorithms Workshop, vol. 10, ACM, New York, NY, USA, 2009, pp. 59–66.
[61] T. Storch, How randomized search heuristics find maximum cliques in planar graphs, in: M. Cattolico (Ed.), Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2006), ACM, New York, NY, USA, 2006, pp. 567–574.

[62] J. Reichel, M. Skutella, Evolutionary algorithms and matroid optimization problems, in: Lipson [98], pp. 947–954.
[63] D.H. Wolpert, W.G. Macready, No free lunch theorems for optimization, IEEE Transactions on Evolutionary Computation 1 (1) (1997) 67–82.
[64] T.M. English, Optimization is easy and learning is hard in the typical function, in: A. Zalzala, C. Fonseca, J.-H. Kim, A. Smith (Eds.), Proceedings of the Congress on Evolutionary Computation (CEC-2000), IEEE Press, July 2000, pp. 924–931.
[65] T.M. English, Practical implications of new results in conservation of optimizer performance, in: M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J.J.M. Guervós, H.-P. Schwefel (Eds.), Proceedings of the Sixth International Conference on Parallel Problem Solving from Nature (PPSN VI), in: Lecture Notes in Computer Science, vol. 1917, Springer, Paris, France, September 2000, pp. 69–78.
[66] C. Schumacher, M.D. Vose, L.D. Whitley, The no free lunch and problem description length, in: L. Spector, E.D. Goodman, A. Wu, W.B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M.H. Garzon, E. Burke (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), Morgan Kaufmann, San Francisco, CA, USA, 2001, pp. 565–570.
[67] C. Igel, M. Toussaint, On classes of functions for which no free lunch results hold, Information Processing Letters 86 (6) (2003) 317–321.
[68] R. Poli, M. Graff, N.F. McPhee, Free lunches for function and program induction, in: Foundations of Genetic Algorithms Workshop, vol. 10, ACM, New York, NY, USA, 2009, pp. 183–194.
[69] R. Poli, M. Graff, There is a free lunch for hyper-heuristics, genetic programming and computer scientists, in: L. Vanneschi, S. Gustafson, A. Moraglio, I.D. Falco, M. Ebner (Eds.), EUROGP-2009, in: Lecture Notes in Computer Science, vol. 5481, Springer, 2009, pp. 195–207.
[70] R. Poli, M. Graff, Free lunches for neural network search, in: F. Rothlauf (Ed.), GECCO, ACM, 2009, pp. 1291–1298.
[71] T. Bäck, Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms, Oxford University Press, New York, 1996.
[72] G. Rudolph, Convergence of evolutionary algorithms in general search spaces, in: Proceedings of IEEE Conference on International Conference on Evolutionary Computation (ICEC-1996), IEEE Press, Nagoya, Japan, May 1996, pp. 50–54.
[73] R. Poli, W.B. Langdon, M. Clerc, C.R. Stephens, Continuous optimisation theory made easy? Finite-element models of evolutionary strategies, genetic algorithms and particle swarm optimizers, in: C.R. Stephens, M. Toussaint, L.D. Whitley, P.F. Stadler (Eds.), Foundations of Genetic Algorithms Workshop, vol. 9, in: Lecture Notes in Computer Science, vol. 4436, Springer, Mexico City, Mexico, 2007, pp. 165–193.
[74] Y. Borenstein, R. Poli, Information landscapes, in: H.-G. Beyer, U.-M. O'Reilly (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2005), ACM, Washington DC, USA, June 2005, pp. 1515–1522.
[75] J. Hadamard, Sur les problèmes aux dérivées partielles et leur signification physique, Princeton University Bulletin (1902) 49–52.
[76] B. Efron, T. Hastie, I. Johnstone, R. Tibshirani, Least angle regression, Annals of Statistics 32 (2) (2004) 407–499.
[77] E. Alpaydin, Introduction to Machine Learning, MIT Press, Cambridge, MA, USA, 2004.
[78] R. Poli, J. Woodward, E.K. Burke, A histogram-matching approach to the evolution of bin-packing strategies, in: IEEE Congress on Evolutionary Computation, IEEE, 2007, pp. 3500–3507.
[79] R. Poli, TinyGP, see Genetic and Evolutionary Computation Conference (GECCO-2004) competition at http://cswww.essex.ac.uk/staff/sml/gecco/TinyGP.html, June 2004.
[80] U.-M. O'Reilly, F. Oppacher, Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing, in: Y. Davidor, H.-P. Schwefel, R. Manner (Eds.), Proceedings of the Third International Conference on Parallel Problem Solving from Nature (PPSN VI), in: Lecture Notes in Computer Science, vol. 866, Springer-Verlag, Jerusalem, 1994, pp. 397–406.
[81] C. Igel, M. Hüsken, Empirical evaluation of the improved Rprop learning algorithms, Neurocomputing 50 (2003) 105–123.
[82] K.A. Smith-Miles, Cross-disciplinary perspectives on meta-learning for algorithm selection, ACM Comput. Surv. 41 (1) (2008) 1–25.
[83] J.A. Boyan, A.W. Moore, Learning evaluation functions for global optimization and boolean satisfiability, in: Proceeding of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference (AAAI-1998, IAAI-1998), AAAI Press, Madison, WI, USA, 1998, pp. 3–10.
[84] O. Telelis, P. Stamatopoulos, Combinatorial optimization through statistical instance-based learning, in: Proceedings of the 13th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2001), IEEE Computer Society, Dallas, TX, USA, 2001, p. 203.
[85] M.G. Lagoudakis, M.L. Littman, Learning to select branching rules in the dpll procedure for satisfiability, Electronic Notes in Discrete Mathematics 9 (2001) 344–359.
[86] E. Horvitz, Y. Ruan, C.P. Gomes, H.A. Kautz, B. Selman, D.M. Chickering, A Bayesian approach to tackling hard computational problems, in: J.S. Breese, D. Koller (Eds.), UAI, Morgan Kaufmann, 2001, pp. 235–244.
[87] M.G. Lagoudakis, M.L. Littman, Algorithm selection using reinforcement learning, in: Langley [99], pp. 511–518.
[88] R. Vuduc, J. Demmel, J. Bilmes, Statistical models for automatic performance tuning, in: V.N. Alexandrov, J. Dongarra, B.A. Juliano, R.S. Renner, C.J.K. Tan (Eds.), International Conference on Computational Science, vol. 1, in: Lecture Notes in Computer Science, vol. 2073, Springer, 2001, pp. 117–126.
[89] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N.M. Amato, L. Rauchwerger, A framework for adaptive algorithm selection in STAPL, in: K. Pingali, K.A. Yelick, A.S. Grimshaw (Eds.), PPOPP, ACM, 2005, pp. 277–288.
[90] E.A. Brewer, High-level optimization via automated statistical modeling, in: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP-1995), ACM Press, Santa Barbara, CA, USA, 1995, pp. 80–91.
[91] B. Singer, M.M. Veloso, Learning to predict performance from formula modeling and training data, in: Langley [99], pp. 887–894.
[92] C. Gebruers, B. Hnich, D.G. Bridge, E.C. Freuder, Using CBR to select solution strategies in constraint programming, in: H. Muñoz-Avila, F. Ricci (Eds.), ICCBR, in: Lecture Notes in Computer Science, vol. 3620, Springer, 2005, pp. 222–236.
[93] C. Gebruers, A. Guerri, B. Hnich, M. Milano, Making choices using structure at the instance level within a case based reasoning framework, in: J.-C. Régin, M. Rueher (Eds.), CPAIOR, in: Lecture Notes in Computer Science, vol. 3011, Springer, 2004, pp. 380–386.
[94] C. Gebruers, A. Guerri, Machine learning for portfolio selection using structure at the instance level, in: Wallace [100], p. 794.
[95] T. Carchrae, J.C. Beck, Applying machine learning to low-knowledge control of optimization algorithms, Computational Intelligence 21 (4) (2005) 372–387.
[96] L. Lobjois, M. Lemaître, Branch and bound algorithm selection by performance prediction, in: Proceeding of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference (AAAI-1998, IAAI-1998), AAAI Press, Madison, WI, USA, 1998, pp. 353–358.
[97] S. Johnson, Hierarchical clustering schemes, Psychometrika 32 (3) (1967) 241–254.
[98] H. Lipson (Ed.), Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2007), ACM, London, England, UK, July 2007.
[99] P. Langley (Ed.), Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29–July 2, 2000, Morgan Kaufmann, 2000.
[100] M. Wallace (Ed.), Principles and Practice of Constraint Programming – CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27–October 1, 2004, Proceedings, Lecture Notes in Computer Science, vol. 3258, Springer, 2004.
[101] C. Bessiere (Ed.), Principles and Practice of Constraint Programming – CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23–27, 2007, Proceedings, Lecture Notes in Computer Science, vol. 4741, Springer, 2007.