Models of the Performance of Evolutionary Program Induction Algorithms

Mario Graff-Guerrero

A thesis submitted for the degree of Doctor of Philosophy in Computer Science
Department of Computing Science and Electronic Engineering
University of Essex
October 2010
# Contents

1 Introduction 1
   1.1 Motivations ................................................. 2
   1.2 Objectives ............................................... 4
   1.3 Achievements .............................................. 5
   1.4 List of Publications ...................................... 6
   1.5 Thesis Outline ............................................ 7

2 Literature Survey 9
   2.1 Search Algorithms ......................................... 9
       2.1.1 Local Search ........................................ 10
       2.1.2 Evolutionary Algorithms .......................... 11
       2.1.3 Evolutionary Program Induction Algorithms .... 13
   2.2 No Free Lunch Theorems .................................. 16
   2.3 Problem Hardness .......................................... 19
   2.4 Information Landscape .................................... 21
   2.5 Summary .................................................. 23

3 Free Lunches for Function and Program Induction 24
   3.1 Literature Review ......................................... 25
   3.2 The Geometry of Function and Program Induction .... 27
   3.3 No Free Lunch for Program and Function Induction . 32
   3.4 Free Lunches for Function and Program Induction .. 36
   3.5 Extensions ................................................. 43
   3.6 Implications of the No-free Lunch for Evolutionary Algorithms 45
   3.7 Summary .................................................. 47

4 Modelling Evolutionary Program-induction Algorithm Performance 49
   4.1 Information Landscape for Search in Program Spaces? 50
   4.2 A Sparse Representation of Fitness ........................ 53
   4.3 From Sparse Fitnesses to Sparse Performance Landscapes . 54
   4.4 From Sparse Performance Landscapes to Sparse Problem Landscapes . 55
   4.5 Model Identification ...................................... 58
   4.6 Model Optimisation ....................................... 59
   4.7 Summary .................................................. 61

5 Testing the Performance Models 63
   5.1 Test Problems and their Representation in our Models .... 63
       5.1.1 EPA’s Problems ................................... 63
       5.1.2 Artificial Neural Network Problems ............... 65
       5.1.3 One-dimension Off-line Bin Packing ............... 65
   5.2 Systems and Parameter Settings ............................ 67
       5.2.1 GP Systems ......................................... 67
       5.2.2 GEP Systems ...................................... 68
       5.2.3 CGP Systems ...................................... 68
       5.2.4 SIHC Systems ...................................... 69
       5.2.5 ANN and Bin Packing Heuristics .................... 70
   5.3 Results .................................................. 71
5.3.1 Not all Similarity Measures are Equal .................................................. 71
5.3.2 Performance of Performance Models of EPAs ....................................... 73
5.3.3 Artificial Neural Networks Results ......................................................... 76
5.3.4 Off-line Bin Packing Results ................................................................. 77
5.4 Summary .................................................................................................... 78

6 Performance Models and Algorithm Selection Problem ................................ 80
6.1 Algorithm Selection Problem ..................................................................... 80
6.2 Related Work ............................................................................................ 82
6.2.1 Dynamic Selection .................................................................................. 82
6.2.2 Static Selection ....................................................................................... 83
6.3 Similarities and Differences with our Model ............................................ 84
6.4 Program-induction Portfolios .................................................................... 85
6.5 Summary .................................................................................................... 88

7 Eliciting Knowledge from Performance Models ........................................... 89
7.1 Comparing Algorithms and Parameter Settings ......................................... 89
7.1.1 Using Angles to Measure Similarity ....................................................... 90
7.1.2 Using Distance Functions to Measure Similarity .................................... 91
7.1.3 Using Similarity Measures .................................................................... 91
7.2 Taxonomies for Program-induction Algorithms ......................................... 92
7.3 Creating EPA’s Taxonomies ....................................................................... 94
7.4 Taxonomies ............................................................................................... 96
7.5 Taxonomies for EPAs with Different Population Sizes ............................. 99
7.6 What Knowledge can we Extract from Measuring Performance Empirically? 103
7.7 Summary .................................................................................................... 104

8 Performance Models using Hardness Measures ........................................... 106
8.1 Hardness Measure for Boolean Functions ............................................... 106
8.2 Hardness Measure for Continuous Functions .......................................... 107
8.3 Modelling the Performance of EPAs using Hardness Measures ............... 110
8.4 Model Identification .................................................................................. 110
8.4.1 Model Optimisation ............................................................................... 111
8.5 Test Problems, Parameter Settings and Model Initialisation ................. 111
8.6 Results ....................................................................................................... 112
8.6.1 Comparison between Performance Models and Hardness Models .......... 112
8.7 Hardness Models and Algorithm Selection Problem ............................... 114
8.8 Summary .................................................................................................... 115

9 Conclusions ................................................................................................ 116
9.1 Achievements ............................................................................................ 116
9.2 Limitations ............................................................................................... 118
9.3 Future Work .............................................................................................. 118
9.3.1 Performance Model ............................................................................... 118
9.3.2 Hardness Model .................................................................................... 120
9.3.3 Algorithm Selection ............................................................................... 120
9.3.4 Algorithm Taxonomies ......................................................................... 120
List of Figures

2.1 Evolutionary Algorithm .......................................................... 12
2.2 The function $f(x) = \frac{0.9(x+2.5)}{x}$ represented as a program-tree. ......................... 15
2.3 A sample search space (top left), a problem (i.e., an assignment of fitness to the elements of the search space) (top right), and two sample permutations of the fitness function (bottom). .......................... 17
2.4 A fitness function seen as a vector and its permutations. ........................................... 18
3.1 Symbolic regression problem. Circles represent the values of $t$ that are used to form $t$ ....... 27
3.2 In symbolic regression, fitness (to be minimised) is the distance between the target behaviour $t$ and the behaviour exhibited by each program, $p_1, p_2$, etc. .......................... 29
3.3 If $g$ measures the squared difference between two numbers, a valid fitness function requires the spheres centred on each program behaviour and with radius given by their corresponding fitness to intersect at one point: the target vector $t$. .................................................. 30
3.4 If $g$ measures the absolute difference between two numbers, then a valid fitness function requires diamonds centred on each of the program behaviours and with edge-length given by $\sqrt{2}$ times the corresponding fitness to intersect in one segment. Any target vector $t$ on that segment satisfies the constraints. .............................................................. 30
3.5 Geometric interpretation of program induction fitness functions which are functions of the sum of errors or fitness contributions over a set of fitness cases as in Equation 3.2 .............. 31
3.6 Three programs $p_1, p_2, p_3$ whose behaviours $p_1, p_2, p_3$ over a test set of $n = 2$ fitness cases are the corners of an equilateral triangle in $R^2$. A fitness function $f$ induced by a target vector $t_1$ assigns fitnesses $f_1, f_2$ and $f_3$ to such programs (top left). Then, because of geometric symmetries, any permutation of such fitness assignments to the three programs is also induced by a target vector (see $t_2, \ldots, t_6$ in top right and remaining panels). .................. 34
3.7 Illustration of Theorem 2. Let $p_1$ and $p_2$ be two program behaviours: (a) as the target behaviour $t$ is varied, the vectors $g(p_1, t)$ and $g(p_2, t)$ produce two curves in $R^n$; (b) for a given $t$ the fitness of the programs $p_1$ and $p_2$ is determined by the length of the projection of the vectors $g(p_1, t)$ and $g(p_2, t)$ onto the vector $1$ via the function $h$; (c) each fitness value corresponds to a hyperplane: (d) Theorem 2 requires that for each set of hyperplanes it is possible to find a target behaviour $t$ such that $g(p_1, t)$ belongs to the hyperplane originally associated with $g(p_1, t)$ and $g(p_2, t)$ while $g(p_2, t)$ belongs to the hyperplane originally associated with $g(p_1, t)$. ................................................................. 35
3.8 Certain assignments of fitnesses to program behaviours are incompatible, in the sense that their fitness cannot represent the error between the functionality of the programs and the target functionality. Thus, there cannot be a point where all spheres meet. In these cases one cannot find a target vector $t$ that could generate such a fitness function. ........................................... 37
3.9 A search space and a city-block-type symbolic regression fitness function (a), which, however, cannot be permuted and still remain a valid symbolic regression fitness function (b). .................. 38
3.10 A search space and a valid Euclidean-type symbolic regression fitness function (a), which, however, cannot be permuted and still remain a valid symbolic regression fitness function (b). ............. 39
3.11 If two programs have identical behaviour, they must have identical fitness or there cannot be any intersection between the spheres centred on them. ........................................... 40
3.12 The programs $p_1$ and $p_2$ have fitness such that $f_1 + f_2$ is smaller than their distance. Thus, by the triangular inequality there cannot be any intersection between the spheres centred on them. ........................................... 41
3.13 The program $p_3$ has a fitness greater than $f_1 + 2f_2$. The figure depicts the case when the fitnesses corresponding to $p_1$ and $p_3$ are swapped. Clearly, the new fitness of $p_1$ (dotted line) contains the circle centred on $p_2$, and hence there is not a point $t$ where these circles can met. 42
3.14 Graphical interpretation of fitnesses and distances for symbolic regression fitness functions with constant terms. ................................................................. 44

4.1 Representation of terms $d(p_i, t)$ of Equation (4.4) and the behaviours of the program of $S$. .......................... 54
4.2 Representation of the case where neither target behaviour $t$ nor behaviour $p_1$ are included in the search space. The arrows represent the similarity between behaviours. ........................................... 55
4.3 General case of Equation (4.5) where all the members of $S$ (i.e. $S = \{t, p_1, \ldots, p_4\}$) are not included in the search space instead they belong to the set of all problems. Each of the circles in the rectangle represent a search space, for example, the circle at the top represents the search space of problem $p_1$. ................................................................. 57

5.1 Scatter plots of the performance measured vs the performance obtained from the model for continuous regression problems using BRF (a & b) and normalized 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. ................................................................. 74
5.2 Scatter plots of the actual performance (epochs) vs the performance estimated by the model for ANN training problems. ................................................................. 77
5.3 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 and using the square of the dot products as $d$. ................................................................. 79

6.1 Algorithm Selection ................................................................. 81
6.2 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. ................................................................. 87
6.3 Ratio between the portfolio using as predictor the “performance model” and the mean of the portfolio. ................................................................. 88

7.1 Example dendrogram: hollow circles represent class members, solid ones represent clusters. ......................... 95
7.2 Taxonomy of 21 different program-induction algorithms created initialising the performance models on the rational problems. ................................................................. 97
7.3 Taxonomy of 21 different program-induction algorithms created initialising the performance models on the Boolean induction problems. ................................................................. 99
7.4 Taxonomy of different program-induction algorithms with different population size and created initialising the performance models on the rational problems. ................................................................. 101
7.5 Taxonomy of different program-induction algorithms with different population size and created initialising the performance models on the Boolean induction problems. ................................................................. 102

8.1 Schematic representation of all the input patterns for two Boolean variables and the distance between input pattern $(0, 0)$ and all other input patterns ................................................................. 108
8.2 Graphical representation of the sets $C_i(j_1)$. ................................................................. 109
List of Tables

2.1 Fitness function .......................................................... 22
2.2 Information Landscape .................................................. 22

3.1 Performance of two deterministic search algorithms in the search space generated by the
primitive set \{a, b, +\}, the trees are restricted to 3 nodes at most and the performance is the
number of evaluations needed to find the optimum .................................. 25

4.1 Boolean induction problems of 3 inputs .................................. 58

5.1 Parameters used in the GP experiments ............................... 67
5.2 Parameters used in the GEP experiments ............................. 69
5.3 Parameters used in the CGP experiments ............................. 69
5.4 Parameters used in the SIHC experiments ............................. 70
5.5 Distances or Similarities functions for rational problems .............. 71
5.6 Distances or Similarities for Boolean problems ........................ 72
5.7 Quality of the model for different closeness measures (GP with 90\% crossover) .... 72
5.8 Mean RSE for the different closeness measures ....................... 73
5.9 Quality of the model (RSE) for different GP, GEP, CGP and SIHC systems and parameter
settings .................................................................................. 75
5.10 Quality of the models of ANN training for different closeness measures ............. 77
5.11 Quality of the model for the different heuristics and similarity measures in the off-line bin
packing problem ................................................................. 78

6.1 Performance of our algorithm portfolios on the validation set ............. 88

7.1 Standard experimental results with the GP, GEP, and SIHC systems under study .......... 105

8.1 Quality of the hardness model (RSE) for different EPAs and parameter settings .......... 113
Acknowledgements

This work could not be possible without the invaluable help of many people. First of all, I would like to thank my supervisor Riccardo Poli for his support, feedback and comments made to this dissertation, the papers published together and in general the research done during my time at Essex.

It was also very important the support I received from home. I am grateful to: my parents, brothers, parents-in-law, brothers-in-law, and my friends Eric and Jose. I would like to thank Sergio and Nena for their immeasurable support. I will never forget all the special occasions we spent together. During my time at the University, I made some good friends and I wish to thank them too. In particular, I would like to mention Fernando and Karla with whom; we spent a lot time together.

Specially, I am most grateful to my beloved wife Erika. First of all, I would not have enrolled at the University without her help, support and encouragement. During our time here, she continued being my support and, I am sure, I could not have finished my PhD, the way I did, without her help and support. Most important, I would like to thank my wife for making this time a happy one.
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 of evolutionary algorithms. However, these models are often criticised for being only applicable to simplistic problems or algorithms with unrealistic parameters.

Here, 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 some simple and practical models 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, Cartesian Genetic Programming and Stochastic Iterated Hill Climbers.

In all cases our models are able to accurately predict the performance of each algorithm on unseen problems. This allows, for example, the use of our models to solve the algorithm selection problem (i.e., the problem of deciding which is the best algorithm to solve a problem) for program induction.

Besides performing accurate predictions, we show that our models can help in the analysis and comparison of different algorithms and/or algorithms with different parameters setting. This process, too, can be automated. We illustrate this via the automatic construction of a taxonomy for the stochastic program-induction algorithms considered in this study.

Although our approach was initially aimed at modelling evolutionary program induction algorithms, it is in fact very general and, in principle, can be used to predict the performance of non-evolutionary learning algorithms and problem solvers. To illustrate this, we modelled one well-known training algorithm for artificial neural networks and two common heuristics of the off-line bin packing problem with very encouraging results.
A great variety of problems can be solved by formulating them in terms of the exploration of a search space. A search space is a set containing possible solutions of a problem. Of course, some of these solutions would solve the problem very poorly; however, among them there may be one or more that are optimal or near optimal, that is, that solve the problem to an acceptable degree.

Once the search space is defined, it is the job of a search algorithm to look for a suitable solution in this set. Normally, given the size of typical search spaces, it is not possible to test all the elements of the space in order to find a solution, and thus, a search algorithm tries to find a suitable solution by testing as few candidate solutions as possible. Over the last decades, researchers have proposed a variety of search algorithms each one using a different procedure to sample the search space.

Sometimes, search algorithms are tailored to solve a particular class of problems. In this case, these algorithms are very efficient at solving that particular class of problems, but they solve other classes of problems inefficiently. In some cases, one can prove that a specialised search algorithm is optimal (with respect to a particular performance function) for solving a class of problems and, therefore, there does not exist an algorithm that performs better for that class.

Generally, the behaviour of specialised search algorithms is characterised using traditional computational complexity techniques. That is, these algorithms are analysed doing a run-time analysis, such as estimating the time needed to solve the easiest problem, the average problem or the hardest problem. Such analyses provide a detailed picture of the performance of an algorithm and, therefore, one can rely on this to decide which algorithm to use. Unfortunately, performing a run-time analysis is a complex task that requires a detailed knowledge of the algorithm and the problem being analysed. Therefore, for most algorithms and classes of problems carrying out such an analysis is often impractical.

This analysis is even more complicated when the algorithms under study are designed to work on a variety of classes of problems. This is because one would need to study the complexity of the algorithm on each of the classes it is expected to work with. Unfortunately, generally the analysis of an algorithm solving
a particular problem cannot easily be translated into an analysis of that algorithm on other problem. As a result, one needs to use other techniques to measure the performance of these types of algorithms.

A search algorithm designed to work with a variety of classes of problems has the objective of finding an acceptable solution for a diversity of classes of problems. Often these algorithms have a natural inspiration. For example, they may attempt to reproduce some elements of natural evolution, immune systems or the annealing process, which can all somehow be seen as specialised problem solvers in nature.

Among the natural inspired algorithms, we find the Evolutionary Algorithms (EAs): a class of algorithms that has seen a considerable popularity in recent years. EAs are relatively simple forms of search and optimisation \([36, 18, 73, 77, 5, 6]\). Their invention dates back many decades \([42, 30, 115, 101, 59]\) (see also \([29]\)). As a result, one might imagine that, by now, we should have a full theoretical understanding of their operations, as well as, a rich set of theoretically-sound guidelines for their parametrisation and customisation. Unfortunately, the situation is quite different.

### 1.1 Motivations

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 \([79, 134, 19, 108, 106, 105, 107, 120, 121, 119, 39, 88, 72, 63, 40, 94, 95, 96]\). An important reason for this delay is that each algorithm, representation, set of genetic operators and, in some cases, fitness/objective 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 Evolutionary Program-induction Algorithms (EPAs), including Genetic Programming (GP) \([59, 60, 63, 92]\), Cartesian GP (CGP) \([74, 75]\), Gene Expression Programming (GEP) \([28]\), Grammatical Evolution (GE) \([84, 83]\), Evolutionary Programming (EP) \([30]\), among others. These are focused on techniques that aim at the automatic evolution of computer programs. Since computer programs are complex entities of variable length and structure, our theoretical understanding of these algorithms has been even slower to develop than for other EAs. The main reason is the objective difficulty of modelling stochastic searchers in infinite spaces 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 a solid theoretical foundations for GP and related EPAs (e.g., see \([63, 95, 97]\) and the recent review in \([92]\)) and the establishment of a forum where EPA theoreticians and practitioners can meet and discuss (the “Genetic Programming Theory and Practice” workshop series \([104, 87, 147, 103]\)), there is a growing gap between EPA theory and practice.

Often theoretical studies and models of EAs and EPAs are criticised for not being easily applicable to realistic situations (see, for example, the discussion in \([90]\)). One of the reasons for this is that producing a
A comprehensive theory for complex adaptive systems such as EAs is objectively a very hard and slow process, as mentioned earlier, while EA technology develops at an unrelentingly fast pace. Another reason is that, sometimes, theoreticians appear to focus on approaches and problems that are too distant from practice. So, despite the proven effectiveness of EAs (see for example [92]), there is a growing 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.

One consequence of the inapplicability of the theories of EAs and EPAs to model algorithms in realistic situations is that these theories do not enable an analysis of the similarities and differences between different search algorithms (or the same algorithm with different parameters). As a consequence, one is left with the traditional approach of computing, for the algorithm under study, some performance statistics on sets of test problems. With this information, one may know whether the performance of two algorithms is statistical different and conclude that, for these test problems, the algorithms exhibit a difference in performance. An equivalent procedure is required to understand the effects of the parameters (e.g., mutation rate, size of the population among others) on the algorithm’s performance. That is, one modifies the parameters following some strategy and computes for each configuration the performance of the algorithm. Clearly, this approach provides a picture of the effects of the parameters on performance; however, the process is slow and can only reveal certain aspects of the performance. We believe that this situation is unsatisfactory and that there is an urgent need for practical models of algorithm performance. These will complement the traditional empirical approach and, in some cases, may highlight similarities and differences that empirical studies cannot easily reveal. Furthermore, the availability of models that can be used to compare algorithms may push the development towards new search algorithms that differ completely from current algorithms. Moreover, such an analysis may reveal that algorithms that seem completely different in fact behave very similarly.

To sum up this thesis has been motivated by the following:

- There is a gap between theory and algorithms. This is the result of the complexity of modelling new algorithms and the rapid development of new techniques.

- Traditionally, theoretical studies are focused on simplified versions of algorithms or on simplistic problems. This makes them unsuitable to study standard algorithms with commonly used parameters working on realistic problems.

- The previous two problems are intensified for the case of EPAs were infinite and dynamic search spaces increase the difficulty of producing theoretical models.

- Current theories of EAs and EPAs do not provide information about the behaviour of a real algorithms when solving realistic problems.
• Current theories do not provide information to decide which algorithm and/or set of parameters are best to solve a problem.

• Contemporary theories of EAs and EPAs do not help in the analysis of the similarities and differences of algorithms on realistic situations, forcing researchers to purely rely on empirical studies and statistical tests.

1.2 Objectives

This thesis attempts to rectify the situations depicted above by proposing practical models of EPAs. The models, by design, do not capture all the characteristics of an algorithm nor model the algorithm exactly (something that typically can only be done on artificially simple systems and problems). Instead, these models focus on what matters the most to practitioners, the performance of EAs in realistic problems, accepting the fact that, in practice, modelling performance can only be done with a certain degree of approximation.

We need models that can easily be applied to new types of algorithms, besides the ones they were originally developed for. Ideally we would like, for example, to be able to produce a performance model for the new evolutionary algorithm as quickly as possible and, therefore, reduce the time gap between the appearance of a new algorithm and its model.

We need models that 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 is the best algorithm to solve a problem or a class of problems? We believe that an approximate answer to these questions is better than the current situation: since no alternative model of EPA performance is available, we simply can get no answers except by direct empirical experimentation.

We need models that not only are able to make accurate predictions of performance but also facilitate the comparison of the behaviour of different algorithms.

In summary the objectives of this thesis are:

• Create accurate models for the performance of EPAs that can be applied to standard algorithms on realistic problems.

• The models must be suitable for the comparison of algorithms and the construction of taxonomies.

• The models must be able to help in the decision of which algorithm and/or set of parameters are the most suitable to solve a particular problem.
1.3 Achievements

Performance Models

The core contribution of this thesis is the modelling technique to predict the performance of evolutionary algorithms, namely the performance models. These models produce accurate predictions for all the problems, algorithms and performance measures tested. Furthermore, these models only require a few ingredients in order to instantiate them. These are: a training set, a set of reference problems\(^1\) and a measure of closeness between problems.

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

Algorithms Selection Problem

Our empirical models are somehow related to techniques used to solve the algorithm selection problem [102] (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 [69, 68, 46, 81, 65, 66, 67, 146, 145, 144], i.e., collections of algorithms that are run in parallel or in sequence to solve a particular problem. As we will show, the methodologies presented here are complementary to (but competitive with) other approaches. Indeed, as a further corroboration of the practicality and accuracy of our models, we will demonstrate their use in the creation of effective portfolios of program induction algorithms: an area where no algorithm selection technique had been tested before.

Given the similarities of the performance models and the models used in the algorithm selection problem, we develop a different model for the performance of EPAs following an approach similar to the ones used in the algorithm selection problem. That is, we create a model based on a function that is related to the hardness of the problems. We call these models hardness models.

As we will see, the algorithm portfolio created with our performance models has a better average performance than all the algorithms composing the portfolio. Furthermore, performance is further improved when one carefully selects, out of the two modelling techniques presented in the thesis, the one to use to model each of the algorithm in the portfolio.

Eliciting Knowledge from Performance Models

Our models can also be used beyond the pure prediction of performance. For example, they enable us to measure the similarity (using a distance function), in relation to performance, between two search algorithms. These comparisons provide information about the relationship between different

\(^1\)The set of reference problems, as we will see, can be obtained from the training set.
search algorithms (or the same algorithm but with different parameters), by making a comparison of all possible pairs of algorithms studied here. However, when the number of algorithms is increased, drawing conclusions by looking at these comparisons is problematic.

In order to help in the analysis of these comparisons, we propose an automatic procedure to build meaningful and informative taxonomies for evolutionary and stochastic program-induction algorithms. As we will see, the taxonomies obtained highlight surprising differences and similarities of the search algorithms included in them.

1.4 List of Publications

This dissertation is based on a number of publications. Only, the model presented in Chapter 8 is unpublished work. The list of publications is presented below.

Journal Articles


Proceedings


1.5 Thesis Outline

Chapter 2 presents the related work. It starts by describing traditional search algorithms, such as A* and simulated annealing. This is followed by a description of work related to the performance of search algorithms. Topics covered include: the No Free Lunch (NFL) theorems; hardness measures, such as the fitness distance correlation and the negative slope coefficient; and the information landscape which was a source of inspiration of our proposed modelling techniques.

Chapter 3 analyses the applicability of the NFL theorems to program induction algorithms. It will be shown that under mild conditions there is a free lunch for function and program induction. This chapter starts by describing related work. This is followed by a description of the geometry of function and program induction. Based on this geometry, it is shown under which circumstances there is a NFL for program-induction algorithms. We will see that it is practically impossible to comply with the requirements needed by the NFL and, therefore, we have a free-lunch for program-induction algorithms. We conclude the chapter by highlighting the implications of the NFL theorems to EAs.

Chapter 4 introduces our main modelling technique. It starts describing the information landscape and shows that the information landscape cannot be applied to program-induction algorithms. This is why we abandoned the idea of using information landscapes to model EPAs and came up with a new model, the performance models. We will see that the performance model is a general modelling technique. In principle, it can model any search algorithms or problem solver as long as there is a suitable function to measure similarity between problems.

Chapter 5 shows the quality of our performance models through a set of experiments. In this chapter, we model different types of program-induction algorithms, an artificial neural network learning algorithm and two human designed heuristics for the problem of off-line bin packing. There, we will see that the performance model is able to produce models with good accuracy for all the problems and algorithms tested.

Chapter 6 shows that the performance model can be used to solve the algorithm selection problem. To demonstrate this, we will build an algorithm portfolio that relies on the performance model to decide which algorithm to use to solve the problem presented to the portfolio. We will see that an algorithm portfolio based on our performance model obtains better average performance than any of the algorithms composing the portfolio.
Chapter 7 shows that the performance models can be used to analyse the differences and similarities between search algorithms. In this chapter, we will describe a procedure to compute the similarity between two algorithms based on the performance models. Furthermore, we will use this procedure to compare algorithms by creating a meaningful and informative taxonomy of EPAs. We will see that this taxonomy highlights interesting similarities and differences between algorithms.

Chapter 8 presents the hardness models. A hardness model is a modelling technique specialised on the two classes of test problems of EPAs used in this thesis, namely symbolic regression of rational functions and Boolean induction problems. We will start the chapter by describing a hardness measure for Boolean induction problems. We will see that this hardness measure is not suitable for continuous functions, and, thus, we will modify it to address this limitation. With this enhancement, we are able to produce hardness models for all the program-induction algorithms and classes of problems tested in this thesis. Through a set of experiments, we will study the quality of these hardness models. We will also perform a comparison between the performance model, introduced previously, and the hardness model.

Chapter 9 presents the conclusions of this thesis and suggests a number of possible avenues for future research.
Chapter 2

Literature Survey

In this chapter we present a survey of works that are related to studying the performance of search algorithms, in particular EAs. After, reviewing different search algorithms (see Section 2.1), Section 2.2 presents the No-Free Lunch theorems that govern the performance of all non-resampling search algorithms. This is followed by consideration of works that are at the opposite end of the generality spectrum; in other words, works that are focused on studying a specific search algorithms working on a particular problem (Section 2.3). Section 2.4 presents the work most related to the present research, which is a model to predict the performance of GAs, namely Information Landscape.

2.1 Search Algorithms

Search algorithms can be classified according to the amount of information they use about the search space. *Uninformed* search algorithms are those that only know when they have found a solution. Among these algorithms we can find traditional AI search algorithms like: breath-first search, depth-first search, bidirectional search and iterative-deepening search, just to mention a few. On the other hand, *informed* search algorithms have an evaluation function $f(x)$ (called *fitness function* in the EA community or *heuristic function* in the AI community), which is used to guide the search. This function $f(x)$ estimates the quality of the solution $x$. For simplicity, let us suppose that fitness function $f(x)$ decreases when the quality of $x$ improves and it is zero when a perfect solution has been found. Then, we can see that an uninformed search algorithm is equivalent to an informed search algorithm having a binary fitness function. That is, $f(x) = 0$ if and only if $x$ is a solution and $f(x) = 1$ otherwise.

In the category of informed search algorithms we can find traditional AI algorithms like A*, best-first search, all evolutionary algorithms like ES, GA, GP, GEP, CGP, particle swarm optimisation, and ant optimisation, among others.

Although this thesis is focused on EAs, as previously mentioned, our survey of search algorithms starts
with A*. This is an informed search algorithm that uses an evaluation function \( f(x) \). \( f(x) \) is defined as \( g(x) + h(x) \), where \( g(x) \) is the cost of going from the element where the search started to the current element \( x \), and \( h(x) \) estimates the cost of the cheapest path from \( x \) to the goal element. It was proved that A* is optimal when \( h(x) \) never overestimates the actual value of the path. Even more important is that, if \( h(x) \) is consistent, then A* is also optimally efficient for a given function \( h \). That is, among all the optimal search algorithms, A* is the one that samples the smallest number of elements before finding the optimal solution.

According to [110], one of the principal drawbacks of A* is the amount of memory it requires. This is because it needs to store all the information to reconstruct every path. Although some modifications to the original procedure of A* have been proposed to overcome this flaw (e.g., memory-bounded A*, simplified memory-bounded A*, among others), these generally come with the price of a more complicated implementation and an increase in the time complexity.

However, in a great variety of situations, it is not necessary to know the path traversed to reach a solution and hence it is not necessary to store this information. Instead, the only information needed is that related to the current state of the search. This is the information about the element, from the search space, being analysed. In this class of algorithms, we find local search algorithms which have the additional characteristic of exploring the search space based on a neighbourhood structure.

### 2.1.1 Local Search

Local search algorithms are related to traditional optimisation methods based on gradient, e.g., Newton-Raphson’s method for estimating the zeros of a real value function or in the derivative of a function if one is interested in optimisation. However, local search algorithms do not require the derivative of the objective function to be computed in order to guide the search. In fact, this is one of the strengths of local search algorithms, because there are a great variety of problems (e.g. program induction, and classification), where it is not possible to compute the derivative of the function being optimised. In fact, in some situations it is not even possible to represent the function in close form: instead it is represented as a procedure.

The basic idea behind local search algorithms is as follows: the algorithm starts from a random point in the search space and from this point it iteratively explores its neighbour elements. After it has explored some of its neighbours, it decides to move to one of them, and then the process starts again from this new point. Finally, the algorithm terminates when a maximum number of evaluations is reached or a termination criterion is satisfied (which usually is that a solution has been found). One of the first steps before applying a local search algorithm is to define a neighbourhood structure into the search space. Frequently, the neighbourhood structure is implicitly defined by the encoding used to represent candidates’ solutions.

---

1. A path to element \( x \) is an ordered list containing the minimum number of elements to reach element \( x \) from the element where the search started.
2. A search algorithms is optimal if it finds the solution with the lowest cost.
3. \( h(x) \) is consistent if for every element \( x' \) reachable from \( x \), the estimated cost of reaching the goal from \( x \) is no greater than the estimated cost of reaching the goal from \( x' \).
For example, for the boolean satisfiability problem, the neighbours of an assignment $x$ might be a set of assignments that differ from $x$ by only one variable.

One of the simplest local search algorithms is *Hill-climbing*. As all local search algorithms, it starts from a random position in the search space and then systematically tests its neighbours until it finds a better element. Then, the current individual is set aside and the process is repeated with this new better element. The procedure terminates when a maximum number of evaluations is reached, the solution has been found or it is not possible to find a neighbour better than the current element.

The behaviour of hill-climbing depends on the strategy used to explore the neighbourhood. For instance in *first-choice* hill-climbing the neighbours are randomly sampled and it moves to the first better neighbour it finds while *stochastic* hill-climbing selects the next position from a random sample of the uphill moves. What these strategies have in common is that they do not test all the neighbours before making a decision. A hill-climbing algorithm that needs to test all the possible neighbours before moving is *Steepest ascent*, which then moves to the best neighbour.

This characteristic, moving only upwards, is one of the strengths of hill-climbing algorithms, but it is also the reason behind one of their main criticisms: they tend to be trapped in local optima. This behaviour can be clarified with a simple example. Let us suppose that the algorithm is climbing a hill containing a local optimum, then it can only continue upwards until it reaches that local optima, and then, it must stop because it is not possible to find a better neighbour.

Different algorithms have been proposed to overcome this problem. One of them is *Random-restart hill climbing*, which essentially is a standard hill-climbing searcher, with the additional characteristic that it is restarted from a different point in the search space when it is believed to be stuck in a local optimum. Of course, Random-restart hill climbing has not been the only technique proposed to escape from local optima. In the quest to find better search algorithms, scientist have looked at natural processes like evolution, social behaviour and physical phenomena, among others, for a source of inspiration for the development of efficient search algorithms.

For example, a local search algorithm inspired by the annealing process in metallurgy is *Simulated Annealing* [58]. This algorithm has the characteristic that it allows movements, with a decreasing probability, to elements that are worse than the current element. In more detail, at the beginning of the run, the probability of moving to an element less fit than the current position is high and as the search continues this probability decreases until the algorithm is only allowed to move to better elements. From this point on simulated annealing behaves like a hill-climbing algorithm.

### 2.1.2 Evolutionary Algorithms

One of the most prolific sources of inspiration of many popular and very powerful search algorithms is Darwinian evolution. Under this name, we can find algorithms such as: Evolution Strategies (ES) [100, 114],
Genetic Algorithms (GA) [42], Evolutionary Programming (EP) [30], Genetic Programming (GP) [59, 60, 63, 92] and Cartesian GP (CGP) [74, 75], among others.

All these algorithms mimic a simplistic version of the process of biological evolution (depicted in Figure 2.1) which consists in creating a population of individuals — where each individual represents a prospective solution of the problem being solved — and modifies this population using genetic operators such as: selection, mutation, recombination, and so on; until a termination criterion is met. At the end of the process the best individual (i.e. the fittest individual) found during the evolution is returned as the solution of the problem.

**Evolution Strategies**

Our review of EAs begins with one of the first EAs, namely *Evolution Strategies* (ES), which is a competitive alternative to traditional optimisation techniques. It was in this domain (i.e., optimisation) that it gained its popularity by finding better solutions than the ones obtained using standard optimisation techniques.

ES may be classified into two different versions: $(\mu/\rho, \lambda)$-ES and $(\mu/\rho + \lambda)$-ES. In $(\mu/\rho, \lambda)$-ES each new population is formed by using the individuals created using the genetic operators (i.e. offspring) while in
(µ/ρ + λ)-ES a new population is constructed using the parent population as well as the offspring. In both cases, µ represents the size of the population, ρ the number of parents involved in creating an offspring and λ the number of elements created at each generation.4

A characteristic that makes ES different from other EAs is that it can use a variable number of parents (i.e., ρ) to create an offspring. In traditional GA the number of parents is fixed depending on the genetic operator being used. For instance, crossover needs two parents to perform its function while mutation requires only one parent to create an offspring. Another difference is that besides representing a candidate solution, an individual in ES also encodes a set of coefficients that modify the course of the evolution by modifying the behaviour of the genetic operators; traditionally this set only governs mutation. Furthermore, this set of coefficients is evolved along with the solutions.

**Genetic Algorithms**

We continue this survey of EAs with Genetic Algorithms (GA). GA did not start as an optimisation technique, as other EAs did. Instead it began with the aim of studying the process of natural evolution by creating systems with similar behaviours to those found in natural environments [43]. However, nowadays, the GA is a general search algorithm that can be used to solve a great variety of problems.

The GA has been traditionally characterised as representing an individual using a fixed-length binary string. As a consequence, the genetic operators (mutation and recombination) were adapted to this binary representation. For example, mutating an individual requires to compute the complement of some randomly selected element of the binary string.

Although the characteristic of using fixed-length encoding has remained constant over time, the exclusive use of binary strings has been modified and, for instance, now we can find GAs using an array of numbers or literals. This has pressured the development of new types of mutation and crossover specifically designed to work with each one of these new representations.

### 2.1.3 Evolutionary Program Induction Algorithms

Let us turn our attention to other types of EAs: Evolutionary Program-induction Algorithms (EPAs). These evolutionary algorithms are focused on techniques that focus on the automatic evolution of computer programs. In this class of EAs we can find Genetic Programming (GP) [59, 60, 63, 92], Cartesian GP (CGP) [74, 75], Gene Expression Programming (GEP) [28], Grammatical Evolution (GE) [84, 83] and Evolutionary Programming (EP) [30], among others.

---

4We refer the reader to [9] for a complete introduction to the field of ES.
Evolutionary Programming

The description of EPAs starts with Evolutionary Programming (EP), which was the first EA to test the possibility of evolving computer programs. EP started as a tool for creating intelligent behaviour using evolution as the driving force. This objective clearly contrasts with the early use of GA and EP that were primarily employed as search and optimisation techniques.

EP began evolving finite state machines (FSM) to make predictions. It used the following procedure: each individual was a FSM that was fed with a set of symbols. For each input symbol received, it returns an output symbol. Then, these outputs were compared with a set of target symbols. The objective of the evolutionary process was to obtain a FSM whose outputs would be as equal as possible to the set of target symbols. In other words, one is using a FSM to learn a function $f(x)$ by providing examples of the form $(x, f(x))$ where $x$ is the input symbol and $f(x)$ represents the target symbol. Besides this application, EP has been used to tackle a great variety of problems like: predicting and classifying time series, pattern recognition, modelling systems and system identification, among other applications.

Genetic Programming

We start describing Genetic Programming (GP) by introducing a data structure that has been widely used in the community to represent programs. Traditional GP represents a program using a tree structure that can vary in size and shape, which dramatically contrasts with the fixed-length representation used by GA. This representation allows for the easy creation of syntactically correct new programs; however, this representation also comes with the price that without any restriction it induces an infinite search space.

Each tree in the search space is constructed by recursively composing primitives in two sets, namely the terminal set $T$ and the function set $F$. $T$ contains the inputs and constant that are necessary for the type of programs being evolved, while $F$ contains the functions. For example, in symbolic regression — which is the problem of finding a program (seen as a function that transforms some numerical inputs into one output) that fits a set of data points — $T$ might be $\{x, \mathbb{R}\}$ and $F$ might be formed by the arithmetic functions $+, -, \times, \div$. $\mathbb{R}$ represents a set of real value constants and $\div$ is generally redefined to avoid division by zero. Figure 2.2 presents an example of a program-tree found in a symbolic regression problem.

A more elaborate example that might help to demonstrate the use of $T$ and $F$ is the problem of creating a program to guide a robot to navigate in an environment. In this case $T$ might contain as inputs the images taken by the robot’s camera, its position read via a GPS device, a set of constants $\mathbb{R}$ and so on. In a nutshell, $T$ should contain the information received by the sensors of the robot. Instead, $F$ might be formed by a set of functions that can be used to analyse these inputs (like: the Hessian, arithmetic functions, transcendental functions, conditionals, loops, and so on), alongside functions which control the motion of the robot.

\footnote{In the community, it is customary to limit the tree’s depth or to set the maximum size of a tree, which makes this infinite search space into a finite one.}
The genetic operators used in GP are specialised to work with tree representation. For instance, traditionally, crossover works by randomly selecting a sub-tree from each parent and swapping these subtrees to form two offspring (this procedure is known as sub-tree crossover). Following an equivalent procedure, sub-tree mutation works by selecting a sub-tree and replacing it with a randomly generated tree.

An important characteristic that makes GP different from other EAs is the procedure used to evaluate the individuals. In GP each individual needs to be executed in order to evaluate its fitness and the fitness measures the degree of deviation of the behaviour encoded by the individual with respect a target behaviour (an equivalent procedure to the one used to evaluate the fitness of a FSM). We will talk more about this characteristic in Chapter 3.

**Gene Expression Programming**

Trees are not the only method used to codify programs. For instance, in *Gene Expression Programming* (GEP) a fixed-size representation (more specifically a fixed length string) is used to represent a program. This is very similar to the representation used in GAs; however, here, the strings are interpreted as having two components: a head and a tail. The head can contain functions and terminals, whereas the tail can only contain terminals.

The genetic operators are also affected with this change of encoding. While in tree-based GP, the operators work by modifying nodes and replacing subtrees, in GEP, they work modifying strings in a manner similar to the operations realised in GAs; however, in GEP the operators have to comply with the constraints on the primitives that can be present in the head and the tail. For example, a genetic operator must never insert a function in the tail of an offspring.

Another difference between GP and GEP is that, in the former, we do not need to transform the representation before the fitness of a program tree can be computed, while in the latter we need to transform the fixed-length representation into a tree structure first. This is done by picking primitives from the fixed-length string and inserting them in a tree following a breadth-first order. Once the tree has been obtained, the

![Figure 2.2: The function $f(x) = \frac{0.9(x+2.5)}{-2x}$ represented as a program-tree.](image)
process of evaluating the tree is equivalent to the one used in GP.

**Cartesian Genetic Programming**

Although GEP uses a fixed-length representation to codify programs, these are nothing more than a flat representation of a parse tree; however, as expected a parse tree is not the only codification that has been used to evolve programs. For example, in *Cartesian Genetic Programming* (CGP) a program is represented as an indexed graph.

In CGP the genotype is a list of integers that is then translated into a directed graph for its execution. As can be seen, this heuristic uses a completely different representation in comparison to other EPAs presented in this section.

A CGP program is defined by a list of integers that correspond to the program inputs, node input connections, functions and program outputs connections. In addition, it is necessary to build the graph the number of rows $n_r$, the number of columns $n_c$ and a level back parameter $\ell$ that limits the connectivity of previous outputs with current inputs.

More specifically, a program is represented as a graph in a grid of $n_r$ rows and $n_c$ columns. Each element of the grid has an associated output which might be connected to another element in the grid. The first elements of the chromosome indicate the inputs of the first element of the grid. After all the inputs of this element have been set, it indicates the function that will be applied to these inputs. This process continues until all the grids have an associated function. Finally the last elements of the chromosome indicate the outputs of the program.

### 2.2 No Free Lunch Theorems

We have devoted the previous section to describing a variety of search algorithms. We started with traditional search algorithms such as A*, then we moved to EAs and finally EPAs were presented. All these algorithms have advantages and disadvantages that at first sight might give the impression that among all possible search algorithms there is one that is the best; however, we know, from the No-Free lunch theorems, that this is not true and finding the best algorithm (i.e., the algorithm having the best performance over all possible problems) may not be possible. This section is devoted to the NFL theorems.

Informally speaking, the no-free-lunch theory (NFL) originally proposed by Wolpert and Macready [140] states that, when evaluated over all possible problems, all algorithms are equally good or bad irrespective of our evaluation criteria. In the last decade there have been a variety of results which have refined and specialised NFL (see [139] for a comprehensive recent review).

One such result states that if one selects a set of fitness functions that are *closed under permutation* (more on this below) then the expected performance of any search algorithm over that set of problems is
constant, i.e., it does not depend on the algorithm we choose [112] nor the chosen performance measure.

Next it is prudent to consider what is meant by a set of functions closed under permutation. A fitness function can be seen as an assignment of fitness values to the elements of the search space, as exemplified graphically in Figure 2.3(top). A permutation of a fitness function is simply a rearrangement of the fitness values originally allocated to the objects in the search space. Examples of permutations are shown at the bottom of Figure 2.3.

If we enumerate the elements of a search space according to some scheme, we can then represent a fitness function as a vector that stores the fitness associated with each point of the space. For example, if we enumerate the five points in Figure 2.3(top left) in a clockwise order, starting from the top, the fitness function in Figure 2.3(top right) can be represented as indicated in the leftmost vector in Figure 2.4. In this case, permuting a fitness function simply means shuffling the elements of the vector representing it. Some permutations of our sample fitness function are shown in Figure 2.4. A set of problems/fitness functions is closed under permutation, if for every function in the set all possible shuffles of that function are also in the set. If all the possible permutations (including the identity permutation) of the elements of the leftmost vector in Figure 2.4 were considered, then the resulting set of 120 fitness functions would be closed under permutation.
In formulae, [112] states that if $F$ is a set of fitness functions closed under permutation, we have that

$$\sum_{f \in F} P(f, a_1) = \sum_{f \in F} P(f, a_2)$$

(2.1)

for any pair of (non-resampling) search algorithms $a_1$ and $a_2$ and for any performance measure, $P$. $P(f, a_i)$ stands for the performance of search algorithm $a_i$ when the problem being solve induces a fitness function $f$.

Furthermore, [112] showed that the connection between closure and NFL is an “if and only if” one. That is, it is also the case that two arbitrary algorithms will have identical performance over a set of functions only if that set of functions is closed under permutation.

An even more general version of NFL was presented by Igel and Toussaint in [48]. When the set $F$ is obtained by permuting a particular function $f$, all of the functions in $F$ present the same histogram of fitness values. This histogram is obtained by applying $f$ to all possible elements of its domain and recording the co-domain values returned by the function. The set $F$ can be closed under permutation if and only if it includes all the functions which present a particular fitness histogram. Naturally, it is possible for a set $F$ to include functions with more than one fitness histogram. For the set to be closed under permutation it must include all functions with each such histogram. Igel and Toussaint [48] proved that NFL applies to a set if and only if all the functions with the same histogram are drawn from the set with identical probability, although different probabilities can be associated to different histograms. A similar result was proved by Streeter [122].

Before we conclude this brief history of NFL, we should point out that Igel and Toussaint [47] were also able to show that if one considers all possible sets of functions with a given domain and a given co-domain, in most conditions the sets that are closed under permutation represent a tiny fraction of the whole. This would suggest that NFL only rarely applies. While this is encouraging, it is not clear what kind of probability distribution characterises realistic problem sets (e.g., it might well be that in reality sets that are closed under permutation occur much more frequently than expected). So, for any given class of problems, it may be very difficult to know whether the class is within the tiny fraction of sets that are closed under permutation.

We should note that if the algorithms that are to be compared are known a priori, then it is possible to construct sets of functions on which NFL holds, yet which are not closed under permutation [138].

While NFL-type results are very important, as they limit what can or cannot be achieved by a search
algorithm performance-wise; in many practical situations they do not really help us estimate the performance of an algorithm on a specific problem or class of problems, unless the class of problems has special features, i.e., if it is closed under permutation then NFL applies \cite{113} and, therefore, a (trivial) prediction of performance is available. However, this trivial prediction does not provide relevant information from a practitioner’s point of view. This is because practitioners are interested in methods that can help determine which is the best algorithm to solve a particular problem. The next two sections are focused on works that have considered this important question.

2.3 Problem Hardness

One possible approach to investigating the behaviour of search algorithms or more specifically their performance is by understanding what makes a problem hard or easy for a particular EA. Problem-difficulty studies in EAs focused initially on the building-block hypothesis for Genetic Algorithms (GAs) \cite{36} and the related notion of deception \cite{20}. The approach involves constructing artificial fitness functions that, based on certain \textit{a priori} assumptions, would be easy or hard for GAs to solve. This has produced some useful results, but, because the work based is upon incomplete theories, some puzzling counter examples have also come to light \cite{78}.

One concept that underlies some more recent approaches, for example the study of the influence of multi-modality on problem difficulty \cite{44}, is the notion of \textit{fitness landscape}, originally proposed in \cite{143}. The fitness landscape metaphor can be helpful in understanding the difficulty of a problem. It is clear, for example, that a smooth landscape with a single optimum will be relatively easy to search for many algorithms (e.g., hill climbing or simulated annealing), while a very rugged landscape, with many local optima, may be more problematic.\footnote{While the metaphor works particularly well for hill-climbing, also approaches based on populations of individuals, like GAs or GP, have been shown to have problems with highly uncorrelated landscapes \cite{56}.}

The graphical visualisation of fitness landscapes, when possible, can give an indication about the difficulty of a problem for search algorithms like EAs. However, generally it is impossible to graph a fitness landscape given the huge sizes of typical search spaces and neighbourhoods. Thus, one really needs to condense useful information on fitness landscapes into one or a few numeric descriptors.

In \cite{54}, Jones introduced a heuristic called \textit{fitness distance correlation (fdc)}, as an algebraic indicator of problem difficulty for GAs. The study of \textit{fdc} has been extended to GP, for instance, in \cite{16, 131, 129, 130, 127, 126}. These studies show that \textit{fdc} is often quite a reliable indicator of problem hardness. However, it has some flaws, the most severe one being that the computation of \textit{fdc} requires the optimal solution (or solutions) to be known beforehand. This is obviously unrealistic and prevents one from using \textit{fdc} to estimate problem difficulty in practical applications.

One measure that does not require knowledge of landscape optima, the \textit{negative slope coefficient (nsc)},
was introduced and tested in [128, 127, 132, 133]. This is closely related to the notion of evolvability. The 
ns is based in the concept of a fitness cloud. A fitness cloud is a scatter plot of parent/offspring fitness 
pairs. The 
s uses the idea of first dividing the scatter plot into a certain number, \(n\), 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. More precisely,

\[
nsc = \sum_{i=1}^{n-1} \min(0, S_i)
\]

where \(S_i\) is the slope of the line connecting the histogram values of \(i\)-th pair of adjacent bins.

The 
s has been proved to be a reliable measure in a number of different benchmark problems in 
GP including, the multiplexer, the intertwined spiral problem, six instances of the royal tree problem, the 
artificial ant on the Santa Fe trail, the even-parity problem, one particular instance of symbolic regression, 
and a trap function [128, 127, 132, 133]. A slightly modified version of 
s called \textit{fitness proportional \ns} 
has also given good results in GAs, for the class of invertible functions of unitation [98].

There have been other hardness measures proposed in the literature such as: NK-model [55], epistasis 
[17] and bit-wise epistasis [31]. They, as well as other harness measures, have shown success in estimating 
the hardness of problems; however, they have been criticised because computing them exactly is as hard as 
searching the whole search space (see [50]). Recently, it has been shown that a general predictive measure 
like the ones just mentioned cannot be built unless \(P = NP\) (see [38]), i.e., unless the class of problems 
solvable in polynomial time is equivalent to the class of non-polynomial problems. At first, this result might 
be discouraging; however, it only states that there is not a single efficient hardness measure applicable for 
all problems and therefore it is still interesting to look for hardness measures designed for particular classes 
of problems.

While \textit{fdc} and \textit{nsc} (and other indicators of problem difficulty) have been shown to provide reasonably 
reliable measures of \textit{whether} a problem is hard or easy, they do not really give a direct estimation of \textit{how hard} or \textit{easy} a problem is in relation to any particular performance measure. That is, they cannot easily 
be used to estimate, for example, the success rate of an EA or the expected number of fitness evaluations 
required to solve a problem. To the best of our knowledge, no approach has ever been proposed that achieves 
this for EPAs. 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. Therefore it is useful to briefly review them below.

Precise bounds on the expected run time for the \((1+1)\) EA, i.e., an EA with a population of only 
one individual, can be obtained by using computational complexity techniques, but typically only for very 
specific classes of functions (e.g., see [21, 137, 51, 22, 52]). The situation is not much better in the case of EAs 
for continuous optimisation. There, we have a reasonably clear mathematical understanding of behaviour
and performance essentially only for Evolutionary Strategies (ES) [4], but only when they are applied to particularly simple functions (such as spheres and ridges). Markov chain models with continuous state spaces can be defined and general results have been obtained using them [109]. In principle, these cover most forms of continuous optimisation including for example, real-valued GAs. However, the complexity of the calculations involved makes them less than ideal for the analysis of the performance of continuous optimisers. The performance of continuous optimisers can also be studied alongside a related technique that allows discrete Markov chain models of continuous stochastic optimisers to be built, which can approximate them on arbitrary continuous problems to any precision [91]. This approach was tested on a (1+1)-ES, a bare bones particle swarm optimiser and a real-valued GA, which obtained accurate predictions of run times and success rates over a large number of generations.

While this is promising, it is not clear how to extend the approaches of [109] and [91] to EPAs, given that EPAs’ search is either in a space of discrete structures or in a hybrid discrete space with continuous subspaces (corresponding to real-valued numerical constants) embedded within it. While, in principle, it would be possible to apply Markov chains to predict the performance of EPAs (without real-valued constants) using or extending some recently developed models [96], as a matter of fact these models are immense, making their application to study any practical problem effectively impossible.

An approach that has particularly inspired the work presented in this dissertation is the work on information landscapes in [10], where the performance of a GA was approximately modelled with surprisingly reliable results. We discuss this in the next section.

2.4 Information Landscape

The performance of EAs where selection is based on comparing the fitness of different solutions (such as GAs with rank selection, truncation selection or tournament selection) really only depends on relative fitness values. The fitness function can, therefore, be re-represented using a comparison matrix \( M \) [10]. The matrix represents the outcomes of all the possible comparisons between pairs of solutions that the selection mechanism might need to perform. The selection mechanism can use the information stored in \( M \), instead of the fitness function, to decide which is the winner (loser) of the selection. The elements of the matrix are computed using the following equation:

\[
d(x_i, x_j) = \begin{cases} 
1 & \text{if } f(x_i) > f(x_j) \\
-1 & \text{if } f(x_i) < f(x_j) \\
0 & \text{otherwise}
\end{cases}
\] (2.2)

The value of the element \((i, j)\) in matrix \( M \) is \( d(x_i, x_j) \) where \( x_i \) and \( x_j \) represent the \( i \)-th and \( j \)-th element of the search space, respectively, and \( f \) is the fitness function. For example, Table 2.1 shows a
Table 2.1: Fitness function

<table>
<thead>
<tr>
<th>x</th>
<th>f(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>5</td>
</tr>
<tr>
<td>01</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 2.2: Information Landscape

<table>
<thead>
<tr>
<th>x</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

fitness function \( f \) of a search space formed by all the binary string of length 2 and Table 2.2 presents the corresponding comparison matrix \( M \) induced by \( f \).

\( M \) is skew-symmetric\(^7\) and its main diagonal is zero, then it can be represented more compactly using only its upper (lower) triangle (shaded in Table 2.2). Finally, we can represent \( M \) using a vector \( v = (v_1, v_2, \ldots) \), called information landscape, formed by reading the elements of the upper triangle one by one. In summary, the information landscape is a vector formed by the values of the upper triangle of the comparison matrix \( M \), which contains all the information the EA can ever need about the fitness function to perform its search.

As a result, the performance of a GA on a particular problem must be a function of the corresponding \( v \) vector only. Clearly, this function is expected to be non-linear, potentially quite complex, and practically impossible to derive from first principles for any algorithm of any complexity. However, [10] found that modelling the performance of a GA using a linear function of \( v \) actually provides a very good approximation. The performance function was assumed to be of the form

\[
P(v) \approx a_0 + \sum a_i v_i,
\]  

(2.3)

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.

The most interesting features of this approach are that, in principle, any comparison-based algorithm can be modelled in this manner, and that users are free to choose any performance measure they want to model. The big disadvantage of the technique is that it does not scale well with the size \( n \) of the search space. For example, for a search space of size \( n \), the linear model in Equation (2.3) requires the identification of \( \frac{n(n-1)}{2} + 1 \) coefficients. This implies that, in order to unequivocally identify the coefficients of the linear model, one needs a training set containing at least \( \frac{n(n-1)}{2} + 1 \) problem/performance pairs. Due to the stochasticity of GAs, the evaluation of the performance of a GA on a problem typically requires collecting statistics over multiple runs. Therefore, the construction of a training set suitable for the application of the method is practically impossible except for very small search spaces.

\(^7\)A matrix \( M \) is skew-symmetric if its transpose is \(-M\).
2.5 Summary

The works reviewed in this chapter studied the performance of EAs. After a brief survey of search algorithms, we presented the NFL theorems that are applicable to all search algorithms. However, as mentioned earlier, they provide only a trivial prediction of performance (and only when the class of problems is closed under permutation). This does not provide sufficient information in order to establish how suitable a particular algorithm is for a particular problem.

Other techniques, such as the fitness distance correlation, the negative slope coefficient, and other hardness measures, reviewed in Section 2.3, can only predict if a particular problem is hard or easy, but not precisely how hard or easy it is in relation to a particular algorithm. That is, since hardness measures are focused on studying problems, they cannot be used to compare the performance of different search algorithms working on particular problems. Furthermore, they do not provide information allowing the identification of the easiest or the hardest problem. This is because they are almost binary functions that only state when a particular problem is easy or hard, rather than providing a grades measure of difficulty.

In contrast the information landscape estimates the performance of EAs and thereby might be used as a guide to decide which algorithm is more appropriate to solve a particular problem. Unfortunately, the computational cost to instantiate this model makes its application impossible for any normal search space. In addition to this, this method requires the re-representing of a fitness function as an information landscape and in order to do so, one needs to compute the fitness of all the elements of the search space: something that is inappropriate for any practical problem. As a result of these limitations, we found that the information landscape is inapplicable to GP, CGP, GEP, GE and other program-induction algorithms when considered practically (more about this in Section 4.1).
Chapter 3

Free Lunches for Function and Program Induction

Informally speaking, the no-free-lunch (NFL) theory [140] states that, when evaluated over all possible problems, all algorithms are equally good or bad irrespective of the evaluation criteria. A simple interpretation of the NFL could be that it is impossible to find a search algorithm that provides the best performance when solving all possible problems.

Another way of interpreting the NFL might be to think of it as providing a (trivial) prediction of performance. That is, when the NFL is applicable for a class of problems $\mathcal{F}$ that is closed under permutation (c.u.p.), we know that the performance of a non-resampling search algorithm is defined by Equation (2.1). From the equation, it might be observed that the performance of a non-resampling search algorithm does not depend on the search algorithm. Instead, it only depends on the problems of the class $\mathcal{F}$. Therefore, we can re-write Eq. (2.1) as $\sum_{f \in \mathcal{F}} \frac{P(f, a)}{|\mathcal{F}|} = c_\mathcal{F}$ and thus, a non-resampling algorithm solving all the problems of class $\mathcal{F}$ will have a constant average performance $c_\mathcal{F}$.

One of the objectives of this thesis is to create models of performance for search algorithms that are specialised in the task of finding functions or programs that solve problems. As a result, it is natural to ask whether or not this prediction of performance is available for this class of problems. More specifically, this chapter will clarify in which situations the NFL applies to function and program induction search problems.

The implications of the NFL go beyond an awareness that a non-resampling search algorithm has a constant performance on sets that are c.u.p. The implications and applicability of the NFL have been clarified in several works, such as: [25, 26, 27, 113, 47]. However, here we will focus only on the implications and applicability of the NFL for function and program induction search spaces.

This chapter is organised as follows. We start by presenting previous work that has studied the applicability of the NFL in program search spaces (Section 3.1). Thereafter will be a description of the structure of function and program induction search spaces. This is, the key ingredient to understanding under which
Table 3.1: Performance of two deterministic search algorithms in the search space generated by the primitive set \( \{a, b, +\} \), the trees are restricted to 3 nodes at most and the performance is the number of evaluations needed to find the optimum.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Problem</th>
<th>Average performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>a b a+b a+a b+b</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>6 2 3 4 5</td>
<td>4</td>
</tr>
</tbody>
</table>

conditions the NFL is applicable (Section 3.2) to program search spaces. Section 3.3 formally presents the necessary conditions under which NFL holds for searchers in function and program spaces. Section 3.4 shows both graphically and analytically that, under very mild assumptions on the class of function and program induction problems, NFL does not apply. The applicability of NFL has important consequence for the development of new search algorithms. This will be analysed in Section 3.6.

### 3.1 Literature Review

Among the many extensions of NFL to a variety of domains, Woodward and Neil [142] have made some progress in assessing the applicability of NFL to the search spaces explored by tree-based GP. In particular, they argue that there is a free lunch in a search space whenever there is a non-uniform many-to-one genotype-phenotype mapping, and that the mapping from syntax to functionality in GP is one such mapping. The reason why NFL would not normally be applicable to search in program spaces is that there are many more programs than functionalities and not all functionalities are equally likely. As an example, Woodward [141] considered the search space generated by the primitive set \( \{a, b, +\} \) where \( a \) and \( b \) are terminals and + is the ordinary addition operation. Assuming only trees of up to 3 nodes are allowed, the search space contains the programs \( a, b, a+b, a+a, b+a \) and \( b+b \). However, irrespective of the set of test cases, the functionality of \( a+b \) is always identical to the functionality of \( b+a \). Therefore, if we interpret syntax trees as genotypes and functionalities as phenotypes, the GP genotype-phenotype mapping is many-to-one and non-uniform, which invalidates NFL. This is because an algorithm that always visited \( a+b \) or \( b+a \) last in its search would perform better in average than an algorithm that visited, say, \( a \) last if the adopted performance measure is the number of fitness evaluations required to find the optimum. For example, let \( A \) and \( B \) be two deterministic search algorithms that produce the same trace\(^1\) regardless of the problem. Specifically, let us set the trace of \( A \) to be \((a, b, a+b, a+a, b+b, b+a)\), \( B = (b+a, b, a+b, a+a, b+b, a)\) and focus only on the case where each problem have a perfect solution in the search space. Under these conditions there are only 5 different problems. Table 3.1 shows the performance of algorithms \( A \) and \( B \) on all the problems. As can be seen from the table, the average performance of \( A \) is better than the average performance of \( B \). This is because \( A \) samples \( b+a \) at last while \( B \) visits \( a \) at last.

\(^1\)The trace of a search algorithm is an ordered list containing the points sampled by algorithm where the first element of the list is the first point visited and so on.
Beyond this interesting counterexample, to show that, in general, not all functionalities are equally likely in program search spaces, Woodward and Neil [142] referred to Langdon’s results on the limiting distribution of functionality (as longer and longer programs are considered the proportion of programs with any particular functionality reaches a limit) [63] and to the universal distribution [57] (informally this states that there are many more programs with a simple functionality than programs with a complex one). However, there is formal proof of the former result only for the case of register based machines. Meanwhile, for spaces where programs are represented using syntax trees, there is only a semi-formal argument indicating that a limiting distribution should exist [63]. The latter result instead applies to Turing complete languages, i.e., to programs with memory and loops. Being Turing-complete GP is a real rarity (due to the complexities of avoiding non-terminating programs or programs with very long run times). Essentially, there is no formal proof of the conditions under which NFL holds or, conversely, where there can be a free lunch, in relation to real GP applications and mainstream forms of search over program spaces. Here, we want to understand whether a non-uniform many-to-one genotype-phenotype mapping is the only condition under which NFL breaks down when searching program spaces.

As we have seen in Chapter 2, in order to determine whether NFL applies or not to a search space, one needs to check whether the class of problems is c.u.p. This is an area of particular interest. As will be seen in the case of program search spaces, there are cases where a problem class is closed under permutation, and others where the problem class is not.

A case where the original formulation of NFL is guaranteed to break down even if the set of functions considered is closed under permutation is the case of infinite sets of functions (the original NFL and all its refinements are limited to finite sets of functions) as shown by Streeter [122]. In the case of infinite sets, the original NFL formulation does not hold because it relies on the assumption that all functions are equally likely. Naturally, with infinite sets it is impossible to sample a set uniformly at random. Streeter found that NFL is still applicable if the probability distribution over functions from the set is such that all functions that are permutations of a particular function have identical probability of being drawn from the set. This means that the probability of picking a particular function \( f \) from the set can only depend on the domain of \( f \) and the histogram of fitness values of \( f \) obtained when applying \( f \) to all possible elements of its domain. As Streeter highlighted, whilst there may be an argument for assuming that all functions have an equal probability of being the case in finite search spaces (this essentially amounts to saying that we have no information on the problem), there is no evidence to suggest that real-world problem distributions will behave (in relation to permutations of problems) as required in order for NFL to be applicable.

One may wonder if this result is applicable to tree-based GP, in the sense that tree-based GP is often said to explore the space of all possible recursive compositions of the primitives in its primitive set, which is, of course, an infinite space. However, we should be careful not to confuse the exploration of infinite search
spaces, which is in principle what GP does,\(^2\) with the application of it to an infinite set of fitness functions (or the probabilistic drawing of problems for EPAs from an infinite set).

### 3.2 The Geometry of Function and Program Induction

Before describing the geometry of function and program induction, let us start by explaining in more detail the problem of function and program induction. Program induction is the problem of finding a program that when executed exhibits a behaviour equivalent to a pre-defined target behaviour. For example, let us supposed that the problem is to find a sorting algorithm, then, in this case, the target behaviour is to sort any given array of numbers.

One of the first questions one might need to answer is how to specify the target behaviour. Here, we follow an approach commonly used in EPAs. That is, each target behaviour is described using a set of inputs \(I = \{x_1, x_2, \ldots, x_n\}\). For each input \(x\) in \(I\), we define a target output \(t(x)\). Clearly, if set \(I\) is ordered, one can then represent the target behaviour as a vector \(\mathbf{t} = (t(x_1), t(x_2), \ldots, t(x_n))\). Equally, one can represent the target behaviour of a program \(p\) as \(\mathbf{p} = (p(x_1), p(x_2), \ldots, p(x_n))\) where \(p(x)\) stands for the outputs or the side effects (or both) of program \(p\) when its input is \(x\).

For example, Figure 3.1 shows a typical case of a symbolic regression problem where the objective, as previously mentioned, is to find the structure together with any coefficients of a program implementing the function \(t(x)\) shown in the figure. In order to obtain \(\mathbf{t}\) from function \(t\), one needs to sample \(t\) a number of times (depicted as circles in the figure), lets say \(n\), and for each point \(x\), one needs to store the value of \(t(x)\) in a vector, such that, the first element of the vector is the value of \(t\) in the first point sampled and so on.

The use of inputs and target outputs to encode behaviours help to define the problem of program induction

---

\(^2\)In practice EPAs always search a finite search space, since there are always explicit (software) or implicit (computer hardware) limitations on the size of programs that can be considered.
as the problem of finding a program \( p \) such that \( t(x) = p(x) \, \forall x \in I \). Equally, we can define the problem of function induction (which is also known as symbolic regression) as the problem of finding a function, in this case \( p \), that transforms some numerical inputs \( x \) into output \( t(x) \). In this case inputs are either scalars or vectors representing program inputs. Similarly, the output is either a scalar (a typical case in GP) or a vector of program-outputs (a typical case in artificial neural networks).

Ideally, one would like to find a program \( p \) for each possible input \( x \in I \) that has a behaviour equivalent to the target behaviour, i.e., \( p = t \); however, in a variety of situations, this is not possible and one has to settle for finding a program exhibiting a similar behaviour to the target behaviour. Therefore, the question is how to assess the similarity between a target behaviour \( t \) and a behaviour \( p \).

To answer this question one only needs to look closely at the chosen representation of behaviours. Based on this representation, it is evident that a similarity measure between behaviours may have the following form: \( d : t \times p \rightarrow \mathbb{R} \). Without loss of generality, we assume that if behaviour \( p \) is more similar to \( t \) than behaviour \( \tilde{p} \) to \( t \), then \( d(p, t) < d(\tilde{p}, t) \), and that \( d(p, t) = 0 \) iff \( p(x) = t(x) \, \forall x \in I \). As a result, it is possible to treat the problem of function and program induction as the problem of finding a behaviour \( p \) such that the function \( d \) is minimal. Furthermore, given that \( t \) is fixed for a particular problem, in fact \( t \) defines the problem, then it follows that the fitness of behaviour \( p \) can be defined as

\[
 f(p) = d(p, t) \quad (3.1)
\]

where \( p \) represents the behaviour of program \( p \).

Please note that the NFL theorems and free-lunch theorems presented in this chapter only require a function \( d \) with the structure previously mentioned, unless otherwise noted. Nonetheless, it is easier to interpret the NFL and free-lunch results if we make further assumptions about the characteristics of \( d \), \( t \) and \( p \). As a result, in the following paragraphs, we start by describing the process, typically used in EPAs, to compute the fitness of a program.

It follows from the definition of \( t \) and \( p \) that the fitness of a program \( p \) is the result of evaluating the behaviour of \( p \) in a set of inputs \( I \) (which are also known as “fitness cases” in the EPAs literature or as “examples” in the machine learning literature). This is achieved by assessing the amount of deviation between one behaviour and a corresponding target behaviour, adding up the results of that assessment and then, optionally, performing some monotonic transformation of the sum [92]. That is

\[
 f(p) = h \left( \sum_{i=1}^{n} g(p(x_i), t(x_i)) \right) \quad (3.2)
\]

where \( f \) is the fitness function, \( g \) is a function which evaluates the degree to which the behaviour of \( p \) matches a target behaviour \( t \) on each fitness case and \( h \) is a monotonic transformation (more on this below).

In the case of scalar outputs, almost invariably the function \( g \) takes the form \( g(a, b) = |a - b|^k \) for \( k = 1 \)
Figure 3.2: In symbolic regression, fitness (to be minimised) is the distance between the target behaviour $t$ and the behaviour exhibited by each program, $p_1$, $p_2$, etc.

or $k = 2$. When $k = 1$, typically $h$ is the identity function, so Equation (3.2) computes the sum of absolute errors, or it is a scaling/normalisation function (e.g., fitness may be the mean absolute error). When $k = 2$, $h$ may be the square root function (in which case Equation (3.2) is the sum of squared errors), possibly with some normalisation (e.g., to produce a root mean squared error). If outputs and targets are vectors, very often $g(a, b) = ||a - b||^2$.

It is important to note that in most cases Equation (3.2), i.e., the fitness associated to a program $p$, can be interpreted as the distance between vector $t \in \mathbb{R}^n$ and vector $p \in \mathbb{R}^n$. This is the case, for example, if $f$ is the sum of absolute errors, in which case it corresponds to the city-block distance. Also, if $f$ is the root mean squared error, it is also a distance (being proportional to the Euclidean distance). In other cases, a simple transformation of $f$ is a distance. For example, when $f$ is the total sum of squared errors, $\sqrt{f}$ is a distance. Even in the case where $t$ and $p$ are not vectors, it is still possible for function $d$ to be a distance, albeit, in a different space. This is the case, for example, when the outputs are strings and $d$ might be the sum of the Damerau-Levenshtein distance\(^3\) applied to each element in $t$ and $p$ component wise.

Figure 3.2 shows a set of programs (represented as vectors in $\mathbb{R}^n$), their fitness and their relationship with the target vector $t$.\(^4\) Note that whenever we represent programs using their behaviour vector we are essentially focusing on the phenotype-to-fitness mapping, thereby complementing the analysis of Woodward and Neil [142] summarised in Section 3.1.

Let us consider a finite space of programs $\Omega = \{p_i\}_{i=1}^r$, such as, for example, the space of all possible programs with at most a certain size (or depth, if a syntax-tree representation for programs is used), where $r = |\Omega|$ is the cardinality of $\Omega$. In these conditions, a fitness function $f$ over $\Omega$ can be represented as a vector $f = (f_1, \ldots, f_r)$ where $f_i = f(p_i)$ or as:

$$f = (d(p_1, t), \ldots, d(p_r, t))$$  \hspace{1cm} (3.3)

if we defined $f(p)$ to be $d(p_r, t)$, as it was previously described.

---

\(^3\)The Damerau-Levenshtein distance is defined as the minimum number of operations required to transform one string into another.

\(^4\)In the rest of this thesis, we will represent vectors as points whenever possible to avoid cluttering the figures.
Figure 3.3: If \( g \) measures the squared difference between two numbers, a valid fitness function requires the spheres centred on each program behaviour and with radius given by their corresponding fitness to intersect at one point: the target vector \( t \).

Figure 3.4: If \( g \) measures the absolute difference between two numbers, then a valid fitness function requires diamonds centred on each of the program behaviours and with edge-length given by \( \sqrt{2} \) times the corresponding fitness to intersect in one segment. Any target vector \( t \) on that segment satisfies the constraints.

Note that if we know the fitness \( f \) of a program \( p \) and \( d \) is a distance metric, we know that the target behaviour \( t \) that generated that fitness must be on the surface of a sphere centred on \( p \) (the vector representation of \( p \)) and of radius \( f \). Thus, for every valid program induction fitness function, the target behaviour is at the intersection of the spheres centred on the behaviour of each program in the search space. Naturally, the notion of sphere and whether the intersection is a point or a whole set (segment) depends on the distance metric \( d \) which represents our fitness function (see Figures 3.3 and 3.4).

From Figures 3.3 and 3.4, it is evident that the application of Equation (3.2) to all programs \( p \) in the search space \( \Omega \) produces a system of equations which correspond to a set of geometric constraints on the target behaviour \( t \). In other words, we know that for any fitness \( f_i \) and target behaviour \( t \) the program \( p_i \) should belong to the set \( \{ x | d(x, t) = f_i \ \forall x \in \Omega \} \).
Another way to see these geometric constraints is by looking at the locus of all \( t \) that satisfy Equation (3.2) for all \( p \in \Omega \). To see this, we rewrite Equation (3.3) as

\[
\mathbf{f} = \left[ h(\mathbf{1} \cdot g(p_1, t)), \ldots, h(\mathbf{1} \cdot g(p_r, t)) \right]
\]

(3.4)

where \( \mathbf{1} \) is a vector whose components are all 1, \( \cdot \) represents scalar product and we extend the function \( g \) to act on its two arguments component wise, i.e., \( g(p_i, t) = (g(p_i(x_1), t(x_1)) \ldots g(p_i(x_n), t(x_n))) \). Therefore, every assignment of fitness to the programs in the search space produces a set of constraints on the ordered set \( t \) (irrespective of what exactly its components represent). In particular, if \( f_1, \ldots, f_r \) are the components of the vector \( \mathbf{f} \), then \( t \) must satisfy the system of equations

\[
\begin{align*}
  h^{-1}(f_1) &= \mathbf{1} \cdot g(p_1, t), \\
  \vdots \\
  h^{-1}(f_r) &= \mathbf{1} \cdot g(p_r, t).
\end{align*}
\]

(3.5)

This has a geometric interpretation, albeit not in terms of distances. Because \( \mathbf{1} \cdot g(p_i, t) \) represents the length of the projection of the vector \( g(p_i, t) \) along the vector \( \mathbf{1} \), Equation (3.5) effectively defines \( n \) parallel planes and requires \( t \) to be such that each \( g(p_i, t) \) belongs to one such plane. The situation is illustrated in Figure 3.5.

We should note at this point that whenever the size, \( n \), of the set of fitness cases is finite, and the
representation for each target behaviour \( t(x_i) \) is finite (as is the case for anything that can be represented in a digital computer), then the space of possible fitness functions of the form in Equation (3.2) or Equation (3.4) is finite even if the number of programs in the search space, \( r \), is infinite. This is because if at most \( k \) bits are necessary to represent each \( t(x_i) \), then \( n \times k \) bits are sufficient to represent the set \( t \). Thus, there can be at most \( 2^{nk} \) different fitness functions for program induction. In the particular case of symbolic regression fitness functions, since the target behaviours \( t(x_i) \) are simple real values, then \( k \) is typically either 32 or 64 depending on the chosen representation for floating point numbers. As a result, although the space of program induction fitness functions may be vast, that space is finite whenever the number of test examples is finite. Therefore, Streeter’s result [122] concerning the applicability or otherwise of NFL to infinite sets of fitness functions (see Section 3.1) cannot be seen to apply to the space of programs.

### 3.3 No Free Lunch for Program and Function Induction

As we have seen, program induction fitness functions take the vector form \( f \) in Equation (3.4). That is, the elements of a vector \( f \) representing a problem are not independent degrees of freedom: they are the result of applying function \( d \). An important question is whether or not imposing these constraints on the class of fitness functions has implications for the applicability of the no-free lunch theorem. As we will see below, the answer to this question is that yes, there are important consequences. In particular, we will show that under mild conditions the constraints imply that the set of GP fitness functions is not closed under permutation, and, therefore, NFL does not apply. Thus, the search for superior search algorithms is meaningful when dealing with program search spaces, for at least some performance measures.

As we indicated above, [112] showed that two arbitrary algorithms have identical performance (irrespective of the chosen performance measure) over a set of functions only if that set of functions is closed under permutation. In other words, using the notation introduced in the previous chapter, in order for NFL to apply to a set of functions \( \mathcal{F} \), for every \( f \in \mathcal{F} \) there must be a \( \tilde{f} \in \mathcal{F} \) such that \( \tilde{f}_i = f_{\sigma(i)} \) where \( \sigma \) is a permutation list (i.e., a permutation of the vector \((1, \cdots, r))\).\(^5\) The following theorem connects NFL’s permutations with the true degrees of freedom of any program induction fitness functions.

**Theorem 1.** Let \( \mathcal{F} = \{f_1, f_2, \ldots, f_m\} \) be a set of fitness functions of the form in Equation (3.3) (i.e., \((d(p_1, t), \ldots, d(p_r, t))\)) and let \( \mathcal{T} = \{t_1, t_2, \ldots, t_m\} \) be the set of target behaviours associated to the functions in \( \mathcal{F} \), with \( t_i \) being the vector generating \( f_i \) for all i. The set \( \mathcal{F} \) is closed under permutation (and NFL applies to it) if and only if for all target vectors \( t \in \mathcal{T} \) and for all permutations \( \sigma \) of \((1, \cdots, r)\) there exists a target vector \( \tilde{t} \in \mathcal{T} \) such that

\[
d(p_{\sigma(j)}, t) = d(p_j, \tilde{t})
\]

\(^5\)Since we consider the set of test case inputs \( I \) fixed, here we treat a function \( f \) and its vectorial representation \( \mathbf{f} \) as equivalent. So, we will sometime say that \( \mathbf{f} \) is a function. Also, we will often think of a set of functions \( \mathcal{F} \) as a set of vectors and we will write things such as \( \mathbf{f} \in \mathcal{F} \), instead of the more appropriate \( f \in \mathcal{F} \).
for all $j = 1, 2, \ldots, r$.

**Proof.** We proceed by reductio ab absurdum.

**IF:** Let us assume $\mathcal{F}$ is closed under permutation, but $\exists t \in \mathcal{T}$ and $\exists \sigma$ such that $\forall \tilde{t} \in \mathcal{T}, \exists j \in \{1, \ldots, r\}$ for which $d(p_{\sigma(j)}, t) \neq d(p_j, \tilde{t})$ (note the dependency of $j$ on $\tilde{t}$). Let $f$ be the fitness function associated to $t$. If we permute $f$ with $\sigma$ we obtain a fitness function $\tilde{f}$. There are two possibilities: either $\tilde{f}$ is a fitness function and so a target vector generating it exists, or it isn’t.

If $\tilde{f}$ is not a fitness function, then surely it cannot be a member of $\mathcal{F}$, but then this implies that it is possible to permute an element of the set and obtain an element outside it. So, $\mathcal{F}$ is not closed under permutation, which is a contradiction.

If, instead, $\tilde{f}$ is a fitness function, then let $\tilde{t}$ be a target vector associated to it. So, $f_j = d(p_j, \tilde{t})$. Because $f_{\sigma(j)} = \tilde{f}_j$ and $f_{\sigma(j)} = d(p_{\sigma(j)}, t)$, then $d(p_{\sigma(j)}, t) = d(p_j, \tilde{t})$ for all $j$. However, earlier we assumed that for our particular choice of $t$ and $\sigma$ for every element $\tilde{t} \in \mathcal{T}$ there is always a $j$ for which $d(p_{\sigma(j)}, t) \neq d(p_j, \tilde{t})$.

Since $\tilde{t}$ does not, it must be the case that $\tilde{t} \notin \mathcal{T}$. As a consequence $\tilde{f} \notin \mathcal{F}$. So, $\mathcal{F}$ is not closed under permutation, which is a contradiction.

**ONLY IF:** Let us assume that $\forall t \in \mathcal{T}$ and $\forall \sigma, \exists \tilde{t} \in \mathcal{T}$ such that $d(p_{\sigma(j)}, t) = d(p_j, \tilde{t})$ for all $j$, but that $\exists f \in \mathcal{F}$ and $\exists \sigma$ such that $f_j = f_{\sigma(j)}$ but $\tilde{f} \notin \mathcal{F}$. Let $t$ be the target vector associated to this particular $f$.

Because of our assumption, even when we consider the permutation $\sigma$ that generates $\tilde{f}$ from $f$, there must be some $\tilde{t} \in \mathcal{T}$ for which $d(p_{\sigma(j)}, t) = d(p_j, \tilde{t})$ for all $j$. Since $\tilde{t} \in \mathcal{T}$ it must have a function, $\tilde{f}$, associated to it and $\tilde{f} \in \mathcal{F}$. Furthermore, if we interpret the relationship $d(p_{\sigma(j)}, t) = d(p_j, \tilde{t})$ in terms of fitnesses, it is clear that $f_j = f_{\sigma(j)}$ for all $j$. As a consequence, $\tilde{f} \equiv \tilde{f}$ and so $\tilde{f} \in \mathcal{F}$, which is a contradiction.

For simplicity let us assume that we are dealing with symbolic regression problems and that function $d$ is the Euclidean distance. Then, from a geometrical point of view, the target vector $t \in \mathcal{T}$ associated to a function $f \in \mathcal{F}$ must be at the intersection of the spheres centred on programs $p_1, p_2, \ldots, p_r$ and having radii $f_1, f_2, \ldots, f_r$, respectively. Permuting the elements of $f$ via a permutation $\sigma$ to obtain a new fitness function corresponds to shuffling (according to $\sigma$) the radii of the spheres centred on $p_1, p_2, \ldots, p_r$. Note these centres remain fixed since they represent the behaviour of the programs in the search space. After the shuffling some spheres may have had their radius decreased, whilst others may have increased or remained unchanged. If any radius has changed then $t$ can no longer be the intersection of the spheres. However, we must be able to find a new intersection $\tilde{t} \in \mathcal{T}$, otherwise, the new fitness function we have generated is not a symbolic regression fitness function and, therefore, cannot be a member of $\mathcal{F}$. This would imply that the set is not closed under permutation.

One might wonder, at this point, how difficult it would be to find a set of fitness functions that satisfy Theorem 1. As shown in Figure 3.6, by making use of symmetries, it is easy to create an artificial program space and an associated set of symbolic regression problems which are closed under permutation. So, situations where NFL applies to symbolic regression are possible.
Figure 3.6: Three programs $p_1$, $p_2$, $p_3$ whose behaviours $p_1, p_2, p_3$ over a test set of $n = 2$ fitness cases are the corners of an equilateral triangle in $\mathbb{R}^2$. A fitness function $f$ induced by a target vector $t_1$ assigns fitnesses $f_1$, $f_2$ and $f_3$ to such programs (top left). Then, because of geometric symmetries, any permutation of such fitness assignments to the three programs is also induced by a target vector (see $t_2, \ldots, t_6$ in top right and remaining panels).

Equation (3.6) is a mathematical statement of the geometric requirements for $\tilde{t}$ to exist. The left-hand side represents the fitness that was originally associated to program $j$ and that, because of the permutation/shuffling, must now be assigned to program $\sigma(j)$. The right-hand side represents the constraint that $\tilde{t}$ must be on the surface of the new sphere. Because we require this condition to be verified for every $j$, we have a system of equations where $\tilde{t}$ is the unknown.

As we have seen in Section 3.2, a similar system of equations, namely Equation (3.5), is obtained if we consider fitness functions of the more general form in Equation (3.2). In the proof of Theorem 1 the fact that the function $d$ is a distance is never used. Nothing would prevent the function $d$ from having a form such as $h(1 \cdot g(p_i, t))$. It is then clear that we can generalise Theorem 1 to fitness functions of the form in Equations (3.2) and (3.4) obtaining as a result:

**Theorem 2.** Let $F = \{f_1, f_2, \ldots, f_m\}$ be a set of fitness functions of the form in Equation (3.4) (i.e, $[h(1 \cdot g(p_1, t)), \ldots, h(1 \cdot g(p_r, t))]$) and let $T = \{t_1, t_2, \ldots, t_m\}$ be the set of target behaviours associated to
Figure 3.7: Illustration of Theorem 2. Let \( p_1 \) and \( p_2 \) be two program behaviours: (a) as the target behaviour \( t \) is varied, the vectors \( g(p_1, t) \) and \( g(p_2, t) \) produce two curves in \( \mathbb{R}^n \); (b) for a given \( t \) the fitness of the programs \( p_1 \) and \( p_2 \) is determined by the length of the projection of the vectors \( g(p_1, t) \) and \( g(p_2, t) \) onto the vector \( 1 \) via the function \( h \); (c) each fitness value corresponds to a hyperplane; (d) Theorem 2 requires that for each set of hyperplanes it is possible to find a target behaviour \( \tilde{t} \) such that \( g(p_1, \tilde{t}) \) belongs to the hyperplane originally associated with \( g(p_2, t) \) while \( g(p_2, \tilde{t}) \) belongs to the hyperplane originally associated with \( g(p_1, t) \).

the functions in \( F \), with \( t_i \) being the behaviour generating \( f_i \) for all \( i \). The set \( F \) is closed under permutation (and NFL applies to it) if and only if for all target behaviours \( t \in T \) and for all permutations \( \sigma \) of \( (1, 2, \ldots, r) \) there exists a target behaviour \( \tilde{t} \in T \) such that

\[
1 \cdot g(p_{\sigma(j)}, t) = 1 \cdot g(p_j, \tilde{t}) \tag{3.7}
\]

for all \( j = 1, 2, \ldots, r \).

Proof. The proof is identical to the proof of Theorem 1 with the replacements \( f_i \leftrightarrow h^{-1}(f_i) \) and \( d(a, b) \leftrightarrow 1 \cdot g(a, b) \).

A geometric illustration of this theorem is provided in Figure 3.7.
We should note that Theorems 1 and 2 focus on the set of program behaviours. As a result, they tell us something about the nature of the phenotype-to-fitness function. Essentially they state under which conditions there can be a NFL for a searcher exploring program behaviours with no resampling. If instead a search algorithm explores the space of syntactic structures representing programs (e.g., sequences of machine code instructions or syntax trees), in the presence of symmetries (such as those highlighted by Woodward and Neil) then the searcher will produce resampling of program behaviours even if it has never resampled the same syntactic structure. Therefore, in the presence of a set of behaviours for which NFL holds, this would unavoidably give the “syntactic” searcher a lower average performance than an algorithm which never resampled behaviours.

Related to this, we should note that in constructing the example in Figure 3.6, we assumed program behaviours can be chosen arbitrarily. In reality program behaviours are the expression (e.g., execution) of a corresponding syntactic representation (which we would call a genotype in GP). Accordingly, one should really verify if NFL sets, such as the ones represented in Figure 3.6, would be compatible with the “syntactic”/“genotype” program space. Such sets do exist. For example, if \( \Omega \) includes the three programs

\[
p_1 = 0, \quad p_2 = (x + 1)/2 \quad \text{and} \quad p_3 = ((1 - \sqrt{3}) \times x + (1 + \sqrt{3}))/4
\]

and the fitness cases are \( x = -1 \) and \( x = +1 \), then we are in a situation similar to the one depicted in Figure 3.6. However, it is apparent how artificial this situation is. As a result, in general, finding a set of actual programs (as opposed to behaviours) and fitness cases for which NFL holds for symbolic regression, and, more generally, program/function induction, may be much harder than the figure may suggest.

### 3.4 Free Lunches for Function and Program Induction

Theorems 1 and 2 impose requirements which involve all possible permutations of fitness functions. Naturally, even if we focused on one specific permutation of \( f \), instead of all permutations, we still have the question of whether or not a vector \( \tilde{t} \) satisfying the system of equations (3.6) or equations (3.7) can exist in general.

In the case of \( \tilde{t} \) the problem is principally that we have \( r \) constraint equations \( \{j \in \{1, \ldots, r\}\} \), but only \( n \) variables \( (\tilde{t}_1, \ldots, \tilde{t}_n) \). Since training sets are generally smaller than search spaces, in general the problem of finding a vector \( \tilde{t} \) satisfying Equations (3.6) or Equations (3.7) should be expected to be over-constrained. We should therefore not be able to build a set of problems which is closed under permutation.

Informally speaking, this implies that, in general, there is a free lunch for search in program spaces. We will devote the rest of this section to illustrating this for a variety of practical situations.

Schumacher et al.’s [112] result — that two arbitrary algorithms have identical performance over a set of functions if and only if that set of functions is closed under permutation — was proven by designing a performance measure such that if NFL held over a set of functions \( \mathcal{F} \), which is not closed under permutation, it would then be possible to identify an algorithm whose average performance over \( \mathcal{F} \) is inferior to that of
Figure 3.8: Certain assignments of fitnesses to program behaviours are incompatible, in the sense that their fitness cannot represent the error between the functionality of the programs and the target functionality. Thus, there cannot be a point where all spheres meet. In these cases one cannot find a target vector \( t \) that could generate such a fitness function.

some other algorithm, leading to a contradiction. Another way to view this result is the following:

**Corollary 3.** If the set of functions \( \mathcal{F} \) is not closed under permutation, then there exists at least one performance measure \( M \) under which at least one algorithm has better (worse) than average performance at optimising functions from \( \mathcal{F} \).

As it was mention in Section 2.2, the applicability of the NFL is particular important for what we want to investigate in this thesis.

Whilst an analytical approach to finding conditions where NFL does not hold for program induction is possible, it is particularly easy to see what type of situations might lead to sets of program-induction fitness functions that are not closed under permutation by considering the geometric interpretation of such fitness functions. This is, of course, best illustrated when target vectors and program behaviours can be represented as 2-D points. As shown in Figure 3.8, whilst one can assign any fitness value to program behaviours in the search space, most assignments cannot be induced by a target vector. This is simply because the spheres centred on each program that have a radius equal to the fitness of the program may not intersect at one point. So, even if one started the construction of a set of functions from a function which is induced by a target vector, permuting such a function may easily produce something which violates some of the geometric constraints that symbolic regression fitness functions must satisfy.

Let us look at a simple symbolic regression example. Let us consider the function set \( \mathcal{F} = \{+, -\} \) and the terminal set \( T = \{x\} \) and let us restrict the search space \( \Omega \) to programs of up to depth 1. That is \( \Omega = \{x, (+ x x), (- x x)\} \) (where we expressed programs in standard Lisp prefix notation). Let the size of
Figure 3.9: A search space and a city-block-type symbolic regression fitness function (a), which, however, cannot be permuted and still remain a valid symbolic regression fitness function (b).

the training set be \( n = 2 \), with \( x_1 = -1 \) and \( x_2 = +1 \), and let us assume that the corresponding target values are \( t_1 = 0 \) and \( t_2 = 0 \) (e.g., the target function might be \( t(x) = 0 \)).

Following standard practice in GP let us assume that \( g(a, b) = |a - b| \) and let us evaluate the fitness of program \( p_1 = x \). Clearly \( f_1 = 2 \) since the function \( p_1 \) evaluates to -1 in -1 and to 1 in 1 (so \( p_1 = (-1, 1) \)) thereby producing two fitness contributions (errors) of 1 each. Let us now consider \( p_2 \). Because this is equivalent to the function \( 2x \), \( p_2 = (-2, 2) \) and we get double the errors found in the previous case, resulting in \( f_2 = 4 \). Finally, the fitness of \( p_3 \) is \( f_3 = 0 \) since \( p_3 = 0 \) everywhere and so \( p_3 = (0, 0) \). Thus, we get \( f = (2, 4, 0) \). Geometrically, the situation is as depicted in Figure 3.9(a).

Let us now consider the following permutation of this function: \( \tilde{f} = (0, 2, 4) \). Are there values of \( \tilde{t}_1 \) and \( \tilde{t}_2 \) that can induce this new function? Because this function requires \( f_1 = 0 \), it must be the case that \( \tilde{t}_1 = -1 \) and \( \tilde{t}_2 = +1 \), or else the error for \( p_1 \) would be bigger than 0. But do these values correctly induce the remaining entries in \( \tilde{f} \)? With \( p_2 \), i.e., \( 2x \), we see that these target values produce the correct result, \( \tilde{f}_2 = 2 \). However, when we test \( p_3 \) which is \( 0 \) everywhere, we find that \( \tilde{t} \) induces a value of fitness \( \tilde{f}_3 = 2 \) and not the required 4. The situation is depicted in Figure 3.9(b). Therefore, it is impossible to build a set of symbolic regression fitness functions that is closed under permutation and includes the function \( f = (2, 4, 0) \) induced by \( t = (0, 0) \).

As illustrated in Figure 3.10 the situation does not change if we consider a fitness function based on adding up squared errors instead of absolute errors.

A geometric arrangement similar to the one depicted in Figure 3.10 also represents the even simpler situation where the three programs \( p_1 \), \( p_2 \) and \( p_3 \) are constant functions, e.g., \( p_k = k \). If the target function \( t \) is also a constant function, the three programs will have fitnesses which cannot be permuted in all possible
ways and still produce program induction fitness functions.

These examples are useful since they show that a free lunch is possible for search in program spaces even if there is not many-to-one genotype-phenotype mapping. However, a geometric interpretation of the problem can lead to much more general results. For example, it is easy to see that if two programs have identical behaviour, they must have identical fitness or there cannot be any intersection between the spheres centred on them (see Figure 3.11). With this in mind, it is then simple to understand the following:

**Theorem 4.** Consider a search space which includes at least two programs \( p_1 \) and \( p_2 \) such that \( p_1 = p_2 \) (i.e., the two programs give identical outputs for all \( x \) in \( I \)). Let a set of program induction problems \( \mathcal{F} \) contain a fitness function \( f \) induced by a target vector \( t \) such that there exist a third program \( p_3 \) in the search space with fitness \( f(p_3) \neq f(p_1) = f(p_2) \). Then \( \mathcal{F} \) cannot be closed under permutation and NFL does not hold.

**Proof.** Let \( \alpha = f(p_1) = f(p_2) \) and \( \beta = f(p_3) \) with \( \alpha \neq \beta \). If \( \mathcal{F} \) was closed under permutation, there would have to be a permutation of \( f, \tilde{f} \), such that \( \tilde{f}(p_1) = \beta \) and \( \tilde{f}(p_2) = \alpha \). Let us assume that a vector \( \tilde{t} \) which induces the function \( \tilde{f} \) exists. From Equation (3.1) we have

\[
\beta = d(p_1, \tilde{t}) \quad \text{and} \quad \alpha = d(p_2, \tilde{t})
\]

However, because we know that \( p_1 = p_2 \), we have that \( d(p_1, \tilde{t}) = d(p_2, \tilde{t}) \) and, so, \( \beta = \alpha \), a contradiction. Hence, a target vector \( \tilde{t} \) with the required properties does not exist, \( \tilde{f} \) is not a program induction problem, and, so, \( \tilde{f} \not\in \mathcal{F} \). This then leads to conclude that \( \mathcal{F} \) cannot be closed under permutation.

This result investigates the case where the genotype-phenotype map is many-to-one, in the sense that while programs \( p_1 \) and \( p_2 \) are distinct, their behaviours \( p_1 \) and \( p_2 \) are identical. As a result, we are considering
Figure 3.11: If two programs have identical behaviour, they must have identical fitness or there cannot be any intersection between the spheres centred on them.

the same situation as Woodward and Neil [142] (see Section 3.1). However, we can now see that in the case of program induction fitness functions, non-uniformity in the mapping is not required for NFL to break down. NFL breaks down because it is impossible to create a set of problems that is closed under permutation. Note also that our result does not require that the two programs $p_1$ and $p_2$ be equivalent, as would be the case, for example, for the programs $a + b$ and $b + a$ which were used by Woodward [141] to create a counter-example to NFL in the case of program induction (see Section 3.1). It simply requires that two programs produce the same outputs to all fitness cases in the particular training set one has chosen.

Continuing with our geometric investigation of NFL, let us suppose that function $d$ is a metric, then it is clear that a necessary condition for the existence of a target vector $t$, which induces a particular fitness function, is that the triangular inequality be verified. More precisely:

**Lemma 5.** If a target vector $t$ induces a program induction fitness function $f$, then for every pair of program behaviours $p_1$ and $p_2$ the distance $d(p_1, p_2)$ between $p_1$ and $p_2$ (based on the same metric used to measure fitnesses) must not be greater than $f_1 + f_2$.

*Proof.* Because of the triangular inequality $d(p_1, p_2) ≤ d(p_1, t) + d(p_2, t) = f_1 + f_2$ for any valid program induction fitness function.

From this result, we can see that another common situation where there is incompatibility between an assignment of fitness to programs and the fitness representing a program induction problem is the case in which two programs have different behaviours (and so are represented by two distinct points in the particular space), but the sum of their fitnesses is smaller than their distance. The situation is illustrated in Figure 3.12. This leads to the following result:

**Theorem 6.** Given a set of program induction functions $F$, if there exists any function $f \in F$ and any four program behaviours $p_1, p_2, p_3, p_4$ in a search space such that $d(p_1, p_2) > f_3 + f_4$ then the set is not closed under permutation and NFL does not apply.

*Proof.* If $F$ is closed under permutation, then there must be a permutation of $f, \tilde{f}$, which assigns the values of
Figure 3.12: The programs $p_1$ and $p_2$ have fitness such that $f_1 + f_2$ is smaller than their distance. Thus, by the triangular inequality there cannot be any intersection between the spheres centred on them.

fitness originally associated to $p_3$ and $p_4$ to $p_1$ and $p_2$, respectively, instead. That is $\tilde{f}_1 = f_3$ and $\tilde{f}_2 = f_4$. As a result $d(p_1, p_2) > f_3 + f_4$ implies $d(p_1, p_2) > \tilde{f}_1 + \tilde{f}_2$. Thus, by Lemma 5, there cannot be a target vector $\tilde{t}$ which induces $\tilde{f}$. So, $\tilde{f}$ is not a symbolic regression fitness function and cannot therefore be a member of $\mathcal{F}$. So, not all permutations of $f$ are in $\mathcal{F}$, and the set is not closed under permutation.

It might be observed from Theorem 6 that it provides an upper bound for distances between program behaviours. In order to complement it, the following theorem provides a lower bound for distances based on the relation $|d(x, y) - d(x, z)| \leq d(y, z)$ which is known under the name of inverse triangular inequality:

**Theorem 7.** Let $f$ be a program induction fitness function. For every pair of program behaviours $p_1$ and $p_2$ and corresponding fitnesses $f_1$ and $f_2$, $|f_1 - f_2| \leq d(p_1, p_2)$.

**Proof.** Since $f$ is a program induction fitness function, there must be some target behaviour $t$ that induces it via the equation $f = d(p, t)$. Let us apply the inverse triangular inequality with $x = t$, $y = p_1$ and $z = p_2$. We obtain

$$|d(t, p_1) - d(t, p_2)| \leq d(p_1, p_2) \Rightarrow |f_1 - f_2| \leq d(p_1, p_2).$$

We have seen from Theorems 6 and 7 that the triangle inequality imposes some constraints onto the distances between program behaviours. From a different point of view, these limitations impose a set of constraints onto the fitness. This is shown in the following theorem.

**Theorem 8.** Given a set of program induction problems $\mathcal{F}$, if there exists any function $f \in \mathcal{F}$ and any three program behaviours $p_1, p_2, p_3$ in a search space such that

$$f_3 > f_1 + 2f_2,$$

then the set is not closed under permutation, hence the NFL does not hold.
Figure 3.13: The program \( p_3 \) has a fitness greater than \( f_1 + 2f_2 \). The figure depicts the case when the fitnesses corresponding to \( p_1 \) and \( p_3 \) are swapped. Clearly, the new fitness of \( p_1 \) (dotted line) contains the circle centred on \( p_2 \), and hence there is not a point \( t \) where these circles can meet.

**Proof.** Let us interpret \( f_3 = d(p_3, t) \) as the radius of a circle centred on \( p_3 \) and \( f_1 = d(p_1, t) \) as the radius of a circle centred on \( p_1 \). We want to understand what happens when these circles (and the corresponding radii) are swapped in relation to their ability to meet in one point with the circle centred on \( p_2 \) and of radius \( f_2 \).

Clearly the new circle centred on \( p_1 \) is going to be bigger after the swap, since its new radius, \( f_3 \) is bigger than the old one, \( f_1 \), by hypothesis. Point \( p_2 \) is \( d(p_1, p_2) \) away from \( p_1 \). The circle centred on it has not changed. So, it still has radius \( f_2 \). If \( f_3 > d(p_1, p_2) + f_2 \), the (new) circle centred on \( p_1 \) will entirely contain the circle centred on \( p_2 \), and so there is no point where all three circles can meet. This situations is presented in Figure 3.13.

By applying the triangular inequality we see that \( d(p_1, p_2) + f_2 \leq d(p_1, t) + d(t, p_2) + f_2 = f_1 + 2f_2 \). So, if \( f_3 > f_1 + 2f_2 \) (our hypothesis), there is not point where the three circles meet. As a result, a \( t \) cannot exist which induces the permutation of \( f \) corresponding to swapping the fitnesses of programs \( p_1 \) and \( p_3 \) of \( \Omega \). So, at least one permutation must be missing from the set \( \mathcal{F} \). Therefore, the set is not closed under permutation.

While this result may appear artificial, it is in fact one of the most powerful results developed here from a practitioner’s point of view. Let us consider an application.

Assume that \( p_3 \) is a program that is really bad (e.g., one that always returns a constant output irrespective of its inputs) in relation to the problems in \( \mathcal{F} \) and one is sure that this program cannot solve any of the problems in \( \mathcal{F} \) to any satisfactory degree. That is, \( f_3 \gg 0 \) for all \( f \in \mathcal{F} \). Let us further assume that \( p_1 \) and
\( p_2 \), instead, are reasonably close to being solutions to at least one problem (of our choice) from those in \( \mathcal{F} \). As a consequence, \( f_1 \) and \( f_2 \) will be relatively small with \( f_1 \ll f_3 \) and \( f_2 \ll f_3 \). It should be easy to find problems like these in practice.

In these conditions, Theorem 8 clearly applies. Therefore, in practice proving that a set of program induction problems is not closed under permutation amounts to finding a program that is so bad that it cannot solve any problem to any acceptable degree, and two other programs that come relatively close to solving one problem. If one can satisfy these simple conditions, than one can be certain that there is a superior algorithm for solving the problems in the set.

Theorems 6, 7 and 8 complement our investigation of the necessary conditions that lead to a set of program induction problems that are not closed under permutation. What makes them different from Theorem 4 is that they do not require finding two programs that have equivalent behaviour (i.e., \( p_1 = p_2 \)). In other words, the NFL breaks down not because there is a many-to-one mapping between genotypes and phenotypes, as happened in the counter-example presented by Woodward [141]. Instead, the NFL breaks down because of the geometric constraints imposed by metric \( d \). These theorems show under which circumstances there is a free-lunch for a searcher that always samples different program behaviours.

### 3.5 Extensions

So far, we have principally concentrated on a set of fitness functions which represent the cumulative error a program makes when evaluating a set of training cases. That is, we have been focused on the case where \( f(p) = d(p, t) \). Furthermore, we have studied in particular detail the problems where the fitness (i.e., function \( d \)) satisfies the axioms of distances. Many if not most EPAs use exactly these types of fitness functions. However, the ideas presented in the previous sections can easily be extended to other and more general situations.

One important instance is the case where the classical program induction fitness measure is combined with some other measure of performance, such as a parsimony pressure term, which is added to the fitness of programs and is modulated by the size of the programs. If this is the case the fitness of program \( p \) is obtained as

\[
f(p) = h \left( \sum_{i=1}^{n} g(p(x_i), t(x_i)), c(p) \right)
\]

where: \( c(p) \) is a function, which we will call a parsimony term, whose output depends solely on the structure or behaviour of program \( p \); the function \( h \) combines the error term with the parsimony term \( c(p) \); the other symbols have the same meaning as in Equation (3.2). Frequently this equation can be expressed using \( d \) as follows

\[
f(p) = d(p, t) + c(p),
\]
where $p$ stands for the behaviour of program $p$.

The addition of the offsets $c(p)$ complicates a geometrical interpretation of the requirements for a fitness function to be a program induction fitness function. Essentially, we have two circles around each $p$: an inner circle of radius $c(p)$ and a outer circle of radius $d(p, t) + c(p)$, which represents the fitness assigned to it. A fitness function can then be a program induction fitness function only if all of the outer circles associated with the different programs in the search space meet at point $t$. The situation is exemplified in Figure 3.14.

From an analytic point of view, however, things remain rather simple. All we need for a fitness function to be a valid program induction fitness function induced by a target vector $t$ and a penalty function $c(p)$ is that

$$f_j = d(p_j, t) + c_j$$

for $j = 1, \ldots, r$, where $c$ is a vector representing the parsimony coefficients associated to each program $p_j$, i.e., $c_j = c(p_j)$. It then seems reasonable to expect that a form of generalisation of Theorem 1 for this class of problems would be straightforward. We can also expect that many of the free-lunch results presented in the previous section could be extended to this case and that, in fact, NFL would be even less applicable due to the asymmetries introduced by the constant terms.

As a concrete example of one such extensions, we provide an extension of Theorem 4 which applies to the case where the fitness is composed by the distance between the behaviour of $p$ and the behaviour of $t$ plus some constant that depends on the program $p$.

**Theorem 9.** Consider a search space which includes at least two programs, $p_1$ and $p_2$, such that $p_1(x) = p_2(x)$ for all $x$ in $I$ and such that $c(p_1) = c(p_2)$. Let there be a set of symbolic regression problems with
parsimony terms, $F$, contain a fitness function $f$ induced by a target vector $t$ such that there exist a third program $p_3$ in the search space with fitness $f(p_3)$ such that $f(p_3) \neq f(p_1)$ and $f(p_3) \neq f(p_2)$. Then $F$ cannot be closed under permutation and NFL does not hold.

**Proof.** Given that $c(p_1) = c(p_2)$ and the fact that $p_1(x) = p_2(x)$ for all $x$ in $I$, then $f(p_1) = f(p_2)$. Let $\alpha = f(p_1)$ and $\beta = f(p_3)$ with $\alpha \neq \beta$. If $F$ was closed under permutation, there would have to be a permutation of $f$, $\hat{f}$, such that $\hat{f}(p_1) = \beta$ and $\hat{f}(p_2) = \alpha$. Let us assume that a vector $\tilde{t}$ which induces the function $\hat{f}$ exists. From Equation (3.9) we have

$$\beta = d(p_1, \tilde{t}) + c(p_1) \quad \text{and} \quad \alpha = d(p_2, \tilde{t}) + c(p_2).$$

However, because we $p_1(x) = p_2(x)$, it must be the case that $d(p_1, \tilde{t}) = d(p_2, \tilde{t})$. We also know that $c(p_1) = c(p_2)$ and so $\alpha = \beta$ which is a contradiction. Hence, a target vector $\tilde{t}$ with the required properties does not exist.

Naturally the constant $c(p)$ can represent the evaluation of any property of a program $p$ (not just size). We are also free to set $c(p) \equiv 0$ for all $p$, which shows that Theorem 4 is effectively a corollary of Theorem 9.

We conclude this section with an extension which links the ability to solve problems with free lunches in the most general conditions.

**Theorem 10.** Let $F$ be a set of program induction problems and let $p_1$ be a program in the search space $\Omega$ such that there exists a fitness function $f_1$ in $F$ for which $f_1(p_1) = 0$ (i.e., $p_1$ is a 100% correct solution). If there exists a program $p_2$ in $\Omega$ such that there is no fitness function $g$ in $F$ for which $g(p_2) = 0$, then $F$ cannot be closed under permutation and NFL does not hold.

**Proof.** If there is one program, $p_1$, that solves one problem, $f_1$, in the set of problems and one $p_2$ that solves none, then in at least one permutation of $f_1$ we will need to assign a fitness of zero to program $p_2$, but then this means it solves at least one problem in $F$. Contradiction.

### 3.6 Implications of the No-free Lunch for Evolutionary Algorithms

Until now, we have focused on clarifying under which condition the NFL theorems apply for search in program induction problems. Furthermore, we have shown that under mild conditions there is a free-lunch for search in these spaces. However, we have not described what consequences the applicability of the NFL would have on evolutionary algorithms.

Let us start by describing a typical situation that occurs when a new evolutionary search algorithm is presented to the community. Generally, given that it is almost impossible to perform a complete mathematical analysis of the performance of evolutionary algorithms, one is left with the option of testing the
performance of the new algorithm on a set of standard benchmarks. For example, in GP it is common to test every new variant of GP on symbolic regression problems, Boolean function induction, the Santa Fe trial, among other benchmarks (see [92]). This is carried out in order to show the strength of the new proposed search algorithm. That is, one is interested in obtaining a better performance than the one obtained by previous search algorithms for each problem tested.

In order to make further progress, we need to generalise and formalise our treatment a little. Let $P$ be a set containing the problems on which the new search algorithm, $a$, will be tested. That is, for each problem $x \in P$ one measures the performance of $a$ on $x$, i.e., $P_a(x)$. Furthermore, let $\Delta$ be a set containing algorithms designed to solve problems on set $P$ plus the new algorithm proposed $a$. In other words, besides containing algorithm $a$, $\Delta$ contains the algorithms with which $a$ will be compared. Then, ideally one would like that the new algorithm proposed, $a$, would be the algorithm with the best performance overall problems. In formulae,

$$a = \arg\max_{j \in \Delta} E[P_j(x)],$$

where the expectation $E$ is taken over all problems in the benchmark set, i.e., $\forall x \in P$.

We know from the NFL that if $P$ is closed under permutation then $E[P_j(a)] = c$ for any non-resampling search algorithm $j$. That is, we know that the performance is a constant and so it is irrelevant which algorithm solves any of the problems in $P$, given that overall they would have the same performance. However, we know from the theorems presented in this chapter that if $P$ is formed by program induction problems, then this set is not closed under permutation. As a result, there exists a search algorithm that is better than the average, at least for a particular performance measure $P$.

The inapplicability of NFL also has some important consequence for the algorithm selection problem, which is the problem of deciding which tool to choose to solve a problem out of a set of available tools. In other words, it is the problem of choosing an algorithm from $\Delta$ for each problem in $P$. The objective is to select for each problem, $x \in P$, the algorithm $j \in \Delta$ that will have the best performance, i.e., $\arg\max_{j \in \Delta} P_j(x)$. With the aim of obtaining a better average performance that the average performance of any algorithm in $\Delta$.

However, when the NFL applies to $P$ (i.e, $P$ is c.u.p.) and $\Delta$ contains only non-resampling algorithms then any algorithm-selection heuristic built with a some restrictions (more about these below) will have the same average performance that any algorithm in $\Delta$. Under these conditions, we know, from the NFL, that all algorithms in $\Delta$ will have a constant average performance. We will show that any algorithm-selection heuristic will also have a constant average performance equivalent to the one obtained by the algorithms in $\Delta$.

Let us assume that we have a function $P$ that receives an algorithm and a problem. It returns the performance $\tilde{P}$ of the algorithm on the problem. In other words, $P$ can be used as an algorithm-selection heuristic. That is, for each problem one chooses the algorithm having the better performance according to
In order to build $P$, we can use any of the problems in $\mathcal{P}$; however, each problem of $\mathcal{P}$ can only be used once. Furthermore, for sampling the search space, one must use a non-resampling procedure for any problem in $\mathcal{P}$. For instance, $P$ might be built using some of the problems in $\mathcal{P}$, for each problem tested we measure the performance of an algorithm in $\Delta$. With this information, we can use a machine learning technique to build $P$.\footnote{It is worth to mention that when the problem tested is not in $\mathcal{P}$ then we are free to use any procedure to sample the search space.}

The NFL theorems apply to any algorithm-selection heuristic built under these conditions. In order to analyse the average performance of an algorithm-selection heuristic, we need to remember that the function of performance $\tilde{P}$ used in the NFL theorems depends only on the sampling of the search space. That is, $\tilde{P}$ receives a list of elements where the first element corresponds to the fitness of the first element sampled by the search algorithms, the second element is the second element sampled and so on. Therefore, given that, in the construction of the algorithm-selection heuristic, was used non-resampling algorithms and each problem of $\mathcal{P}$ was tested once, then we can see for all the problems in $\mathcal{P}$ the algorithm-selection uses a non-resampling procedure to visit the elements of the search space. Thus it can be considered as another non-resampling search algorithm that is in fact governed by the NFL theorems.

We know that NFL does not hold for typical program/function induction problem sets, this implies that there is a difference in average performance among non-resampling search algorithms that is there are some algorithms that will have good/bad performance. So, in this scenario, it makes sense to try to create a model that predicts the performance of a particular algorithm on a particular problem. The objective is to build a model that can predict the performance of an algorithm without running the search algorithm on the problem of interest. With this prediction at hand we will show on Chapter 6 how it can be used to tackle the algorithm selection problem.

### 3.7 Summary

The work presented in this chapter studies under which conditions there is a free lunch for search on function and program induction problems. After a brief survey of related work, we began by describing the required conditions under which NFL holds for program spaces. We showed that it is extremely easy to find realistic situations in which a set of program induction problems is not closed under permutation, and hence there is a free lunch. Finally, we discussed some of the implications that the NLF has from the perspective of a researcher developing new evolutionary algorithms.

It is important to note that the fact that NFL results do not hold for these problem domains does not imply that the existing methods for searching program and function spaces, such as the many variants of EPAs, are average, better than average or worse than average. This may need to be proved by other means.
The non-applicability of NFL, however, means that it is worthwhile attempting to develop new and more powerful algorithms for function and program induction and for symbolic regression.
Chapter 4

Modelling Evolutionary
Program-induction Algorithm
Performance

Creating a model of the performance of EPAs is one of the main contributions of this thesis. Before explaining the characteristics of the model developed, let us start describing what a model of performance is according to this dissertation. Here, a model of performance is an equation that predicts the performance of an algorithm on a particular problem. The algorithms’ performance can be the fitness of the best individual found, the probability of finding a perfect solution or the time required to find a suitable solution, just to mention a few.

A model of performance depends on a number of factors such as: the algorithm being modelled, the kind of problems under study and the performance measure used. Moreover, looking closer at the algorithm, it is evident that the performance of the algorithm depends on the algorithms’ parameters. As a consequence, the model of performance also depends on these parameters. However, as we will see, the model proposed is able to produce accurate predictions for different of algorithms or the same algorithm with different parameters.

As previously seen, creating such a model is an important step in beginning to answer questions such as: How likely is it that a particular EPA will solve a problem of interest? What fitness should one expect to find for the solutions to a problem at the end of a run? and What is the best EPA for the job? among others. However, obtaining answers to these questions from exact models, as we have seen in Chapter 2, is extremely difficult, because generally, exact models can only be instantiated for simplified versions of EPAs using unrealistic parameters or working on simplistic problems.

In this chapter, we attempt to rectify this situation by proposing a practical model of EPAs. The model,\footnote{The model proposed belongs to the class of supervised learning (see [1]). That is, one needs to provide a set of example in order to instantiate the model (more about this in Section 4.5).}
by design, does not capture all the characteristics of an algorithm nor model the algorithms exactly. Instead, the model focuses on what matters the most to practitioners, the performance of EAs in realistic problems, accepting the fact that, in practice, modelling performance can only be achieved with a certain degree of approximation.

Although our approach was initially aimed at modelling EPAs, the generality of our modelling technique allows us to go beyond program induction by stochastic search, and to capture the characteristics of other forms of search and problem solving.

The rest of the chapter is organised as follows. After a brief review of the information landscape [10] (see Section 2.4) (which was a source of inspiration for the model proposed here), Section 4.1 analyses the possibility of creating a model for the performance of program induction algorithms using the information landscape. Unfortunately, we encountered several problems with the applicability of the information landscape to these spaces and, as a result, we abandoned this approach. In Section 4.2, we start developing our performance model. Once the performance model is introduced, we specialise on the case of program induction algorithms (Section 4.3). From this specialisation, we show that this performance model is general enough to model a variety of search algorithms and problem solvers working on different classes of problems (see Section 4.4). After describing our modelling technique, Sections 4.5 and 4.6 present the procedure used to instantiate it.

4.1 Information Landscape for Search in Program Spaces?

In Section 2.4, we described the work presented in [10] where the performance of an EA was modelled using a re-representation of the fitness function called information landscape. What is interesting about the information landscape is that, in principle, it can be used to model any comparison-based algorithm. Therefore, it is reasonable to ask whether or not it is possible to use the information landscape to produce a model of the performance of EPAs.

In order to answer this question, we need to translate into the information landscape the strategy employed in EPAs to compute the fitness of a program. In Section 3.2, we have seen that the fitness of any program \( p \) is the result of assessing by how much the behaviour \( p \) of program \( p \) deviates from a target behaviour \( t \), i.e., \( d(p, t) \). Without loss of generality, we assumed, in Section 3.2, that fitness \( f(p) = d(p, t)^2 \).

The information landscape is formed by the values of the upper triangle of a comparison matrix \( M \) (see Section 2.4). The elements of matrix \( M \) are computed using Equation (2.2). It might be observed from Equation (2.2) that the information landscape is simply the result of comparing the fitness of all possible pairs of elements in the search space. It is in this equation that the characteristics of EPAs need to be inserted. To do so, we only need to re-write Equation (2.2) using the fact that \( f(p) = d(p, t) \). With these

\[ f(p) = d(p, t)^2 \]
modifications, Equation (2.2) transforms into:

\[
M_{i,j} = \begin{cases} 
1 & \text{if } d(p_i, t) > d(p_j, t) \\
-1 & \text{if } d(p_i, t) = d(p_j, t) \\
0 & \text{otherwise}
\end{cases}
\]

where \( p_i \) represents the behaviour of the \( i \)-th program in the search space.

This re-writing of Equation (2.2) reveals the deeper structure of the information landscape for the case of EPAs. It might be observed from Equation (4.1) that the information landscape depends on the target behaviour \( t \) i.e. on the problem. It also relies on the closeness measure \( d \). This implies that given a target behaviour \( t \) (i.e. a problem) one could induce a different information landscapes only by changing the closeness measure \( d \). In other words, for the same problem, the search space can have different structures induced by a different \( d \).

An important consequence that results from measuring fitness using a closeness function or equivalently from expressing program behaviours as well as target behaviours using the same representation is that program spaces have the additional characteristic that the behaviour of a program, in the search space, can also be considered a target behaviour. One possible interpretation of this is to think of the set of possible solutions (search space) and the set of problems as both living in the same space.

Although the characteristic of having problems and solutions living in the same space does not restrict the applicability of the information landscape to all program search spaces, it imposes some restrictions for particular classes of program spaces. Before describing what sort of restrictions are imposed, we need to identify in which classes of program spaces these restrictions are present.

When problems and solutions live in the same domain, it is evident that if there are \( m \) different behaviours in the search space, there must be at least \( m \) different problems. In general, there could be more problems than different behaviours in the search space. This is the case, for example, if some limitations are imposed on the search space, such as restricting the maximum size of programs, or using a function and a terminal set that do not contain all the necessary components to build any program behaviour, and so on. However, we are interested in the classes of problems where the production of a search space that contains all possible behaviours is almost immediate. In other words, we are interested in search spaces that contain all the possible problems of a particular class. For example, in the Boolean function induction problems with \( x \) inputs, it is extremely easy to create a search space that contains all possible Boolean functions. For instance, in order to create such a search space, one only needs to create a terminal set containing all the possible inputs \( x \), a function set containing the NAND function, without limiting the size of the programs, and the search space is constructed by recursively composing primitives from the terminal and function set.

Before we proceed with our analysis, let us remember that in [10] a linear function of \( v \) was used to
model the performance of an EA (see Equation (2.3)) and that \( v \) has \( \frac{n \times (n-1)}{2} \) components where \( n \) is the size of the search space. Therefore, in order to instantiate Equation (2.3) one needs at least \( \frac{n \times (n-1)}{2} + 1 \) problem/performance pairs. That is, we need \( \frac{n \times (n-1)}{2} + 1 \) different information landscapes \( v \); however, for the class of problems where the search space contains all possible behaviours we have at most \( n \) different problems for a particular closeness measure \( d \). The problem is that \( d \) is kept constant for a particular EPAs. Hence, in this situation, we would only have at most \( n \) different vectors \( v \), thereby the \( a_i \) coefficients of Equation (2.3) could not unequivocally be identified. Things get even worse because these \( n \) different \( v \) vectors are only obtained in the optimistic case where all the programs in the search space exhibit a different behaviour, but, as we will see, this is not generally the case. Although this case limits the applicability of the information landscape, we will assume that there are possible workarounds in order to solve it. However; we found several other problems that do limit the applicability of information landscape to EPAs. We will discuss them below.

The first issue is the scalability of the 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. Thus, the required \( v \) vector would be infinitely dimensional, and as a result, an infinitely large training set would be needed to determine its elements. 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, as we discussed in Chapter 3. For example, within the search space generated by \( \{ x, y, \sqrt{}, +, \times \} \), the programs \( \sqrt{(\times x y)} \) and \( \sqrt{(\times y x)} \) are functionally indistinguishable.\(^3\) The situation is similar with the primitive set \( \{ x_1, x_2, x_3, \text{AND, NAND, OR, NOR} \} \), where, for example, \( \text{AND} x_1 x_2 \) is equivalent to \( \text{AND} x_2 x_1 \). 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. Therefore, the associated coefficients in the information landscape model of performance in Equation (2.3) cannot unequivocally be determined. As a result of all this, the problem of identifying the coefficients in Equation (2.3) is generally badly ill-posed.\(^4\)

\(^3\) We write programs in Lisp-like prefix notation.

\(^4\) While some workarounds are possible to improve the situation, these are computationally expensive and the problem always remains on the brink of being ill-posed.
If one combines this with the scalability issue mentioned above, it becomes clear that information landscapes are practically inapplicable to GP, CGP, GEP, GE and other program-induction algorithms. Therefore, we abandoned the idea of using the information landscape re-representation of fitness functions, and came up with a new re-representation. The new re-representation still shares the conceptual simplicity and appeal of the original idea, but does not suffer from ill-posedness. This is presented in the next sections.

4.2 A Sparse Representation of Fitness

Let \( \Omega \) be the search space and let \( f \) be a fitness function over \( \Omega \). We assume that \( \Omega \) is ordered (e.g., lexicographically). One can then represent the fitness function using a corresponding ordered set \( \mathcal{F}(\Omega) = \{ x | x = f(p) \ \forall p \in \Omega \} \). The first element of \( \mathcal{F}(\Omega) \) is the fitness of the first element in \( \Omega \), and so on. Clearly, \( \mathcal{F}(\Omega) \) is an exact representation of the fitness function. Therefore, taking inspiration from the work on information landscapes presented in Section 2.4, one could imagine estimating the performance of an EPA using the following linear model:

\[
P(f) \approx a_0 + \sum_{p \in \Omega} a_p f(p).
\]

Equation (4.2) could be transformed into a workable model if we accepted to only approximately represent the fitness function in the first place. That is, what if, instead of considering the fitness of all programs in the search space \( \Omega \) as a representation for \( f \), we considered the fitness of the programs in a subset \( S \subseteq \Omega \)?

We could then use the following model of performance

\[
P(f) \approx a_0 + \sum_{p \in S} a_p f(p).
\]

Clearly, the ordered set of fitness values \( \{ f(p) : p \in S \} \) is only a sparse representation of the fitness function and so, Equation (4.3) will necessarily be less accurate than Equation (4.2). However, since the latter is itself an estimate, it is reasonable to ask whether a careful choice of \( S \) might still produce reasonable results. The advantage of this approach is that the cardinality of \( S \) is under our control. Therefore, we can easily ensure that the problem of identifying the coefficients in Equation (4.3) is always well-posed and that there are not too many such coefficients. This in turn keeps the size of the training set under control. For these reasons here we will adopt Equation (4.3) as the real starting point for our models.

This is essentially the idea we will explore here. In the next sections, we will specialise this equation to the
Figure 4.1: Representation of terms $d(p_i, t)$ of Equation (4.4) and the behaviours of the program of $S$.

case of program-induction algorithms. This specialisation shows the generality of our modelling technique beyond program induction algorithms allows us to model other forms of search and even problem solvers in general.

### 4.3 From Sparse Fitnesses to Sparse Performance Landscapes

The procedure used to specialised Equation (4.3) to the case of program-induction algorithms only requires consideration of the method used in program-induction algorithms to measure the quality (i.e., fitness) of a program. That is, the fitness of a program $p$ is the result of measuring the similarity between the behaviour of $p$, namely $p$, and a target behaviour $t$, i.e., $d(p, t)$. In summary, we only need to replace $f(p)$ with $d(p(p), t)$\(^5\). As a result, in these conditions, Equation (4.3) transforms into

$$P(t) \approx a_0 + \sum_{p \in \mathcal{S}} a_p \cdot d(p(p), t), \tag{4.4}$$

where we used $P(t)$ instead of $P(f)$ since the fitness function is really determined by $t$.

Let us suppose that the behaviours of all the programs in the search space and the target behaviour can be drawn on a two dimensional canvas. Figure 4.1, shows a graphical representation of the terms $d(p(p), t)$ (represented by arrows). Each point represent either a behaviour of a program in the search space or the target behaviour $t$. The figure depicts the case in which problems and program behaviours live in the same space.

Figure 4.1 highlights some of the similarities and differences between our model and the information landscape model in Equation (2.3). In particular, let us imagine that $S = \Omega$ and $t$ is the functionality of at least one program in $\Omega$, i.e., there exists a program $\tilde{p} \in \Omega$ such that $f(\tilde{p}) = 0$, i.e., $t = p(\tilde{p})$. Then, effectively, the terms $d(p(p), t)$ in Equation (4.4) compare the functionality of each $p \in \Omega$ against the functionality of $\tilde{p}$ on the training set, which is something very similar to what the $v_i$ terms do in Equation (2.3). Let us

\(^5\)Note that we write $p(p)$ instead of the shorter version $p$ to make explicit that $p$ is the behaviour of program $p$. 

Figure 4.2: Representation of the case where neither target behaviour $t$ nor behaviour $p_1$ are included in the search space. The arrows represent the similarity between behaviours.

remember that terms $v_i$ are computed using Equation (2.2), hence they represent a discrete comparison between two elements of the search space.

One difference is, of course, that the $v_i$ compare every element of a search space to every other element of that search space. Here, instead, we compare the functionality of all programs in the search space against the functionality of one particular program, so the number of terms is not quadratic, but linear. Furthermore, in practice we will choose $S$ such that $|S| \ll |\Omega|$, making the approach viable even for huge search spaces. A further difference between Equations (4.4) and (2.3) is that while the values $v_i$ are discrete, the terms $d(p(p), t)$ are real-valued. Therefore, each one provides more information. In a sense, we could think of $d(p(p), t)$ as a sparse fuzzy version of the notion of information landscape.

4.4 From Sparse Performance Landscapes to Sparse Problem Landscapes

Another difference between Equations (4.4) and (2.3) is that, in principle, the search space may not contain a program with exactly the target functionality $t$, i.e., it may be the case that $\forall p \in \Omega : t \neq p(p)$. As a result, we are allowed to go outside $\Omega$ to pick our second term of comparison, $t$. This suggests that there is really no need to limit also our first term of comparison $p$ to be the functionality of a program actually in $\Omega$. This behaviour is depicted in Figure 4.2 where target functionality $t$ and element $p_1$ are not included in the search space (represented as a circle). Program behaviours $p_2, p_3$ and $p_4$ are still in the search space (inside the circle) to indicate that one is allowed to select any element to form $S$ regardless of its membership of the search space or not.
In formulae, we extend the model in Equation (4.4) to

\[ P(t) \approx a_0 + \sum_{p \in S} a_p \cdot d(p, t) \]  

(4.5)

where \( S \) should be interpreted as a subset of the set of all possible behaviours, rather than a subset of the search space. For example, \( S \) could be a subset of all possible Boolean functions of 4 inputs. In this case the terms \( d(p, t) \) effectively define a problem landscape, i.e., they represent the degree of similarity between any two problems.

Let us remember that in the procedure followed to arrive at Equation (4.5), we assumed \( d(p, t) \) to be the fitness of program \( p \) but, as Equation (4.5) shows, there are not restrictions in the type of function \( d \). That is, we can use as \( d \) any function with the form \( d : p \times t \to \mathbb{R} \). In general, \( d \) can be any function even though this function is not a fitness function. Furthermore, in Section 3.2, we defined \( p \) as \((p(x_1), p(x_2), \ldots, p(x_n))\) where \( x_i \) corresponds to the inputs where program \( p \) is tested. However, given that function \( d \) can receive two elements of \( S \) that are not even included in the search space, it follows that our previous definition of \( p \) is no longer needed. Therefore, we can freely decide the structure of \( p \). The only requirement is that \( t \) has the same structure as \( p \) and that there must exist a function \( d \) defined for the space where \( p \) and \( t \) come from. This freedom in the representation of \( p \) and \( t \) allows us to interpret them as two problems of any kind. We will see that these characteristics are what make our model general and applicable to almost any search algorithm or problem solver.

In these general conditions —without restricting \( d \) to be a fitness function, allowing the elements of \( S \) to be outside of \( \Omega \) and interpreting \( p \) as objects— the real potential of Equation (4.5) is displayed. Figure 4.3 shows the general case where \( t \) and the elements of \( S \) are not in the search space. Instead, they are in the space of all possible problems. In this case \( d(p, t) \) computes the similarity between problem \( p \) and problem \( t \). Please note that we refer to \( t \) as a problem instead of a target behaviour, because, as we mentioned previously, in this case, \( t \) is a problem and not necessarily a behaviour. Moreover, in this case, the only difference between \( t \) and \( p \) is that one is estimating the performance of a particular algorithm on \( t \) and \( p \)'s are the problems forming set \( S \).

At this point, we would like to draw the reader’s attention to the conceptual differences between Figure 4.3 and Figure 4.2. Figure 4.2 presents the case where some of the elements of \( S \) are selected from \( \Omega \) and others from outside \( \Omega \), while Figure 4.3 depicts the case where all the elements of \( S \) do not belong to \( \Omega \). The structure of elements in Figure 4.2 allows us to think of \( d \) as a fitness function even though this is not required; however, for the case depicted in Figure 4.3 it is impossible to think of \( d \) as a fitness function. Instead we can think of it as being a closeness measure between problems.

Moreover, we can see that Figure 4.2 depicts the case where the search space is a subset of the set of all problems, which is usually the case for program-induction problems. However, in general, we can only
Figure 4.3: General case of Equation (4.5) where all the members of $S$ (i.e. $S = \{t, p_1, \ldots, p_4\}$) are not included in the search space instead they belong to the set of all problems. Each of the circles in the rectangle represent a search space, for example, the circle at the top represents the search space of problem $p_1$.

say that both sets are in the same domain and that these sets may overlap. On the other hand, Figure 4.3 presents the case where the set of problems and the search space are disjointed. More specifically, the set of problems and the search spaces are contained in different domains. In addition, each problem (point in the set of problems) induces a search space, as indicated by the line connecting a point with a circle inside the rectangle, and so we have as many search spaces as problems in the set, something that is not present in the case depicted in Figure 4.2, where there is only one search space regardless of the problem chosen.

One of the consequences of having problems and search spaces in different domains is that our model is abstracted from the particular characteristics of the search space. Graphically, we can see that our model only requires the information contained in the left of the figure, that is, the elements of $S$ and closeness measure $d$. And so, it can ignore everything about the search space (on the right of the figure). This characteristic is extremely important because it is the core of the generality of our modelling technique.

For instance, if one wants to model the performance of an EPA or the performance of a SAT (Boolean satisfiability) solver, it is possible to create both models using our technique. This is possible, because our model does not require details on the particular representation for solution, instead the model is formed by the problems and a function $d$ that measures the closeness between different problems. Thus, for the case of program-induction problems, we only need to choose a set of different program-induction problems and a function $d$ that can measure the similarity between these problems. Similarly, for the case of SAT solver, we need a set of different SAT problems and a function to measure their similarity.

Whilst this sounds promising, there is an important question to be addressed before we can proceed with using Equation (4.5). In principle, one could get a model with any arbitrary choice of the set $S$. However, clearly we should expect different choices of $S$ to produce models of different accuracy. In which case, how should we choose the elements of $S$? In this work we will use a form of search in order to find good sets of $S$, from a much larger set of potential candidates. The process is described in the next sections.
Table 4.1: Boolean induction problems of 3 inputs.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>( t_a )</th>
<th>( t_b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 1 0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0 1 1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 1 1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

4.5 Model Identification

We are now in a position to obtain performance models for EPAs and other algorithms. Our models require the following set of ingredients to instantiate them: a training set of problems \( T \), a validation set of problems \( V \) and a closeness measure \( d \). We also require a set of problems \( \Sigma \) from which to draw the elements of \( S \).

The training set \( T \) is used to identify the coefficients \( a_p \) of Equation (4.5) so as to produce a model of good quality. The validation set \( V \) is used to test the generality of the model. \( T \) and \( V \) are composed by pairs \((t, P(t))\) where \( t \) is a problem and \( P(t) \) is the performance of the algorithm under study in problem \( t \).

Let us remember that \( t \) is a vector and for the case of program-induction problems \( t \) represents a target behaviour (see Section 3.2). For example, Table 4.1 shows 2 problems of class of the 3 input Boolean induction problems. From the table, it is observed that each problem is specified with the same inputs, as a result one can represent the target behaviours as a vector (i.e., \( t_a = (0, 0, 1, 0, 1, 0, 0, 1) \) and \( t_a = (1, 0, 0, 1, 1, 0, 1) \)) knowing that the first element of the vector corresponds to the output of the input 0, 0, 1 and so on.

\( P(t) \) is obtained by running the algorithm on \( t \) and estimating its performance. For stochastic algorithms this typically requires performing multiple independent runs and taking the average (or some other statistical measure of tendency, such as the median).

Although the process of initialising Equation (4.5) requires the training set \( T \) and set \( S \) to be obtained first, let us start by presenting the process used to obtain the values of coefficients \( a_p \) supposing that we already have a training set \( T \) and a set \( S \). Section 4.6 presents the procedure used to obtain the elements of \( S \).

Let \( S = \{ t_1, \ldots, t_n \} \) be a set formed by problems \( t_i \), \( d \) be a closeness measure, and \( T = \{(\tilde{t}_1, P(\tilde{t}_1)), \ldots, (\tilde{t}_m, P(\tilde{t}_m))\} \) be the minimum test set required to identify the coefficients \( a_p \) where \( m \geq n + 1 \). Once we have all the ingredients to create the model, the only step missing is obtaining the coefficients \( a_p \) that can be calculated solving the following linear equation

\[
Wa = p. \tag{4.6}
\]
where \( \mathbf{a} \) is a vector representing the \( a_p \) coefficients (i.e., \( \mathbf{a} = (a_0, a_1, \ldots, a_n) \)), \( \mathbf{p} \) is a vector that contains the measure performance for every problem in training set \( T \) (i.e., \( \mathbf{p} = (P(\tilde{t}_1), \ldots, P(\tilde{t}_m))) \), and \( \mathbf{W} \) is a matrix formed by one column and the result of computing \( d \) between the elements in \( S \) and the problems in \( T \). That is,

\[
\mathbf{W} = \begin{pmatrix}
1 & d(t_1, \tilde{t}_1) & \ldots & d(t_n, \tilde{t}_1) \\
\vdots & \vdots & \ddots & \vdots \\
1 & d(t_1, \tilde{t}_m) & \ldots & d(t_n, \tilde{t}_m)
\end{pmatrix}.
\]

For example, let \( d \) be the sum of absolute errors then \( d(t_a, t_b) \) is computed as:

\[
\sum_{\ell} |t_{a,\ell} - t_{b,\ell}| = |0 - 1| + |0 - 0| + |1 - 0| + |0 - 1| + |1 - 1| + |0 - 0| + |0 - 0| = 3,
\]

where \( \ell \) is the number of different input values.

Now, we are in the position to solve Equation (4.6). Generally, the equation cannot be solved exactly. This comes from the fact that traditionally \( \mathbf{W} \) has considerable more rows than columns, i.e., there are more points in the training set than coefficients to identify. As a result, matrix \( \mathbf{W} \) cannot also be inverted\(^6\) to identify coefficients \( a_p \). To solve this, we decide to use the ordinary least square method\(^7\). The idea there is to minimise \( ||\mathbf{W}\mathbf{a} - \mathbf{p}||^2 \) where \( || \) denotes the Euclidean norm in \( \mathbb{R}^m \). It follows that the \( a_p \) coefficients are given by

\[
\mathbf{a} = (\mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^T\mathbf{p}.
\]

### 4.6 Model Optimisation

Various procedures to obtain a feasible set \( S \) exist. In this thesis, we followed the strategy of choosing the elements of \( S \) from a larger set of prospective problems \( \Sigma \) (which will be discussed in further detail later). The idea is to form \( S \) from \( \Sigma \) by selecting those elements that would produce a model with good quality. It is evident that we can solve this problem using optimisations techniques, but, before describing the technique used to obtain \( S \), a measure of the quality of the model is needed in order to decide which is the best \( S \).

In this thesis, we decided to measure the quality of the model via the Relative Squared Error (RSE)\(^4\), which compares the performance of an algorithm as predicted by the model with the actual performance for all the problems in the training set \( T \). This is defined as follows

\[
rse = \frac{\sum_i (P_i - \hat{P}_i)^2}{\sum_i (P_i - \bar{P})^2},
\]

where \( i \) ranges over a set of test problems used to evaluate the accuracy of a model, \( P_i \) is the average performance recorded for problem \( i \), \( \hat{P}_i \) is the performance predicted by the model, and \( \bar{P} \) is the average performance over all problems. The objective is to obtain values of \( rse \) as close as possible to zero.\(^8\)

---

\(^6\)For the case where \( \mathbf{W}^{-1} \) could be computed then \( (\mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^T \equiv \mathbf{W}^{-1} \).

\(^7\)For a better introduction to the subject, we refer the reader to [82].

\(^8\)A value of \( rse \) close to 1 means that the model is as good (or bad) at predicting performance differences as the mean. A value of \( rse \) less than 1 means that the model predicts better than the mean, while \( rse > 1 \) implies worse predictions than the mean.
One of the simplest approaches one can use is the use of a Genetic Algorithm (GA) to find the elements of $S$ from $\Sigma$. The GA can be configured as follows. Each individual includes $n$ loci; each allele represents a problem $t$ (created with the same procedure used to create the problems in the training and validation sets). Each GA individual, therefore, encodes a potentially different $S$ set of size $n$. The objective is to find the set of $n$ problems which gives the minimum $rse$. In one sense, therefore, each GA individual is a linear model. As a result, to evaluate each individual, first we need to derive the corresponding $a_t$ coefficients. These can be obtained following the procedure presented in Section 4.5. Naturally, when the GA optimisation is over, one needs to test the best model ($S$) evolved using an independent validation set to ensure the model generalises correctly.

Although the use of a GA to obtain $S$ is feasible and, in fact, we used it in [37] to instantiate our model, in this thesis, we decided to use the Least Angle Regression (LARS) algorithm (see [24]) to obtain $S$. The reason is that the quality of the models obtained with LARS is comparable with the quality of the models using the GA; however, the time required by the GA to obtain a model is considerably greater than the time needed by LARS. Furthermore, LARS is a deterministic algorithm while the GA is a stochastic one.

LARS is a subset selection algorithm. Among all the available selection algorithms we decided to use LARS because it is less greedy than other model selection algorithms, such as, forward selection or all-possible-subsets regression. Moreover, simple modifications of it implement Lasso [125] or the forward stepwise linear regression algorithm which make it ideal for making comparisons.

LARS works as follows. It starts by setting all the coefficients $a_p$ for $p \in \Sigma$ to zero and finds the predictor (which is a member of $\Sigma$) 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.

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

The question is then how to choose the prospective elements of $\Sigma$. There are many options. For discrete domains of reasonably small size, one option is to use all the possible elements in a class of problems. For example, in the case of Boolean functions with 4 inputs, we could just consider all 65,536 possible such functions. If the cardinality of a class of problems is too big or infinite, one could construct $\Sigma$ by simply drawing a representative set of samples from the class. In this work, we decided to use the latter approach. More specifically, we take $\Sigma = T$, where $T$ is the training set.

To sum up, our version of LARS identifies the $m$ most important members of $T$ (based on their correlation
with the performance measure) to form the set $S$. Then, we use a least square method to obtain the model’s coefficients $a_p$ for $p \in S$.

Although this procedure automates the process of selecting $S$, it still requires the user to specify the size of $S$. To free the user from this delicate duty, we decided to use a cross-validation technique on the training set to determine the best size for $S$. This works as follows. The training set is split into five sets of equal size. Four sets are joined together and are used to produce a model (i.e., these sets are now the training set), while the remaining set is used to assess its generalisation. The process is repeated 5 times, each time leaving out a different fifth of the training set $T$. At the end of this process, we have a prediction of the performance of an algorithm for all the problems in $T$. Since these predictions are obtained via cross-validation they give us an idea of the generalisation capability of our model.

This cross-validation procedure is iteratively applied to the models produced by LARS for $m = 1, 2, \ldots, |T|$ with the aim of identifying the value of $m$ which provides the best generalisation. The process ends when the best generalisation on the training set has been reached. Note that the process of identifying $S$ from $T$ with a known $m$ involves first running LARS and then doing least squares. However, to save computation during the determination of the optimal $m$ we did not perform the least square method after LARS. Instead, we simply used the models produced by the LARS algorithm itself, since these are more accurate.

4.7 Summary

The chapter starts by discussing the applicability of the information landscape to model the performance of EPAs. We found that applying the information landscape to search in program spaces is not possible for various reasons. One is scalability, that is, the size of the training set. Another is the symmetries present in the program-induction search spaces.

As a result, we abandoned this idea and came up with another re-representation of the fitness function that does not suffer from the same problems as the information landscape approach. We then specialised this new re-representation to evolutionary program induction algorithms and found that this specialisation is in fact a framework that can be used to model more search algorithms that just EPAs.
The generality of our modelling technique comes from the structure of the model. That is, the model is based on measuring the closeness of the problem we want to estimate the performance of with a set of reference problems (i.e. $S$). In principle we can model any type of algorithm if we have a sufficient training set and a function to measure the similarity between problems.
Chapter 5

Testing the Performance Models

In Chapter 4 we presented our modelling technique, we described the algorithm used to initialise our models and the measure used to test their quality. The theory presented in the previous chapter is put into practice in this chapter. That is, we will create performance models and test the quality of the models for different algorithms, parameter settings, classes of problems and performance measures.

The chapter is organised as follows. Section 5.1 presents different problems and their representation in our model. The algorithms and heuristics modelled are described in Section 5.2. Section 5.3 presents some result.

5.1 Test Problems and their Representation in our Models

To illustrate the scope and effectiveness of the approach, we consider different classes of problems as well as different algorithms and performance measures. Firstly, we test our approach in problems related to EPAs, namely continuous symbolic regression of rational functions and Boolean inductions problems. The algorithms modelled are different versions of GP, GEP, CGP, and a Stochastic Iterated Hill Climber (SIHC). Secondly, we model an artificial neural network (ANN) training algorithm applied to Boolean induction problems. Finally, we also test the approach on the one-dimensional off-line bin-packing problem and we model two well-known human-designed heuristics: First Fit Decreasing (FFD) and Best Fit Decreasing (BFD). We present the problems in more detail in the rest of this section, while we describe the algorithms used to solve them in Section 5.2.

5.1.1 EPA’s Problems

For program induction, we consider 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 best of run fitness (more on this below), and success rate.
Continuous symbolic regression is the problem of finding a program (seen as a function that transforms some numerical inputs into one output), which fits a set of data points. It is called “symbolic” because, unlike ordinary regression where the task is to find coefficients for a pre-fixed function, here we want to find the function together with any coefficients it may depend on. Boolean function induction problems are problems where one is given a truth table and asked to find a program (a logic function) that implements that truth table.

A benchmark set was created for continuous symbolic regression by generating and then sampling random rational functions. We created 1,100 different rational functions using the following procedure. Two polynomials, \(W(x)\) and \(Q(x)\), were built by randomly choosing the degree of each 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 degree. A rational function in our training set is then given by \(t(x) = \frac{W(x)}{Q(x)}\). Each of the rational functions in the set was then sampled at 21 points uniformly distributed in the interval \([-1, 1]\). This resulted in a target vector \(t \in \mathbb{R}^{21}\).

For each \(t\) vector, we performed 100 independent runs recording the BRF, i.e., the fitness of the best individual found in each run. Then we associate to each \(t\) a value \(P(t)\) obtained by averaging the BRF recorded in these 100 runs. Since fitness is a measure of distance between the target functionality \(t\) and the functionality of a program, values of \(P(t)\) close to zero represent good performance.

For symbolic regression problems we also used a normalised version of BRF (or NBRF for short), which was computed by 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. The normalised target vector was computed as follows. Let \(\mu\) and \(\sigma\) be the mean and standard deviation of the elements \(t_i\) of \(t\), respectively. The elements of the normalised target vector are defined as \(\frac{t_i - \mu}{\sigma}\). The normalised program-behaviour vectors are similarly defined by shifting and scaling the components of the \(p\) vectors.

The second benchmark we used was the class of 4 input Boolean induction problems. As we mentioned previously, this includes 65,536 different functions. Each one is fully represented by a vector of length 16 (a truth table), that is formed by the functions’ output for each possible combination of the 4 inputs. Therefore, in this case our \(t\) vectors are in \(\{0, 1\}^{16}\). We randomly selected 1,100 different Boolean functions from this set and for each counted the number of times the algorithm found a program that encoded the target functionality in 100 independent runs. We took as our performance measure the success rate, i.e., the fraction of successful runs out of the total number of runs.

For each benchmark set we divided the 1,100 randomly generated problems into two sets: a training set \(T\) composed of 500 elements, and a validation set \(V\) comprising the remaining 600 elements. \(^2\)

\(^1\)The fitness function used in all systems is the sum of absolute errors, i.e., we used \(|\cdot|\) as our function \(g\) in Equation (3.2).

\(^2\)The rational and Boolean functions generated, as well as the performance of all systems can be downloaded from http://lsc.fie.umich.mx/~emgrafg/phd_thesis_data/
5.1.2 Artificial Neural Network Problems

We also used Boolean induction problems to test the ability of our approach in modelling a learning algorithm for ANNs. The job of the neural network is to learn Boolean functions with 3 inputs and 2 outputs. There are 65,536 such functions. Each can be represented with a $8 \times 2$ matrix where the rows contain the output for each of the 8 possible different input patterns. Again, we randomly selected 1,100 different Boolean functions of this type; 500 of these functions were used in the training set $T$, and the remaining 600 functions formed the validation set $V$.

The performance measure we focused on was the number of epochs needed to train a ANN (more on this in Section 5.2.5). The training of a neural network differs from the program induction situations in three respects. Firstly, typically, neural networks have multiple outputs while above, for simplicity, we only considered the case of programs with one output. Secondly, once the initial set of weights and biases of a network is fixed, typically training algorithms behave deterministically, being guided by the gradient of the error function. Thus, the only stochasticity in such algorithms is due to the choice of initial conditions. Thirdly, the functionality of neural networks is determined by their topology, weights, biases and activation functions, rather than instructions organised into a program tree or a linear structure.

Despite these differences, there are also key similarities between ANN training and program induction from examples. In program induction, a problem is defined in terms of a target behaviour, and the objective is to find a program that implements such a behaviour. As a result, the fitness (objective) function is typically a measure of how similar the behaviour of a program is to a target behaviour. When behaviours are defined in terms of the outputs produced by a program to given inputs, behaviours are effectively represented with collections of numbers (e.g., vectors). In this case, as we discussed in Section 4.4, problems and program behaviours live in the same space. The same is true also for ANNs where problems are typically described in terms of training sets and behaviours are defined in terms of outputs produced to the input patterns in the training set. Error functions are also typically defined in terms of similarity between such outputs and corresponding desired outputs. Therefore, we linearise the $8 \times 2$ matrix representing a ANN-training problem by reading out the elements of the matrix column by column and row by row. We obtain a bit vector of length 16. This also makes it possible to use Equation (4.5) for the modelling of the performance of ANN training algorithms.

5.1.3 One-dimension Off-line Bin Packing

The objective in the bin-packing problem 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 bin-packing algorithms and the other domains described above is that, in the case of bin packing, problems and algorithm outputs (or behaviours) do not live in the same space (that is, we are in the case depicted in Figure 4.3). 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 a target assignment, which could then be used to measure similarity with the output produced by an algorithm. This is where our extension of the interpretation of the terms \( p \) and \( t \) in Equation (4.5) becomes important. There we suggested interpreting them as *problems* rather than program behaviours (solutions/outputs). What we need now is a good representation of bin-packing problems, which allows the application of our performance model.

While the natural representation for a bin packing problem is a list of item sizes, in the off-line version of the problem considered here, the order of the items in the list is unimportant. This is because the solver can freely choose the order in which to bin the items. For this reason, we decided to represent problems using histograms indicating how many items of each size need packing. Since in bin packing the maximum size of the bins is typically known in advance and fixed, no item bigger than a bin can be packed. Thus, the number of slots in a histogram of item sizes is upper-bounded by the bin size. Furthermore, it is commonly the case that the size of the largest possible item is known and is smaller than the bin size. As a result one needs only use histograms covering up to the maximum item size to represent bin packing problems.

In our experiments we used bins of either size 100 or size 150, while item sizes were integers between 1 and 100. Since the maximum item size in our test problems was 100 different problems can then be represented with histograms including 100 slots, which in turn can be represented as 100-dimensional vectors whose elements take non-negative integer values.

With this representation, we are in a position to describe problems with exactly the same vector formalism we used to describe problems for symbolic regression, Boolean function induction or ANN training. This mean we can use Equation (4.5) and the same procedure presented for EPAs and other induction algorithms to model the performance of bin-packing algorithms.\(^3\)

We created random bin-packing problems using the procedure presented in [99]. All problems required the packing of 1,000 items. We considered five problem classes: for each problem class we specified a maximum and minimum size for the items. We used the following five ranges (one for each problem class): \([1, 100]\), \([10, 90]\), \([20, 80]\), \([30, 70]\), 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 we denote with \( B_{\text{min}} \) and \( B_{\text{max}} \) the minimum and maximum size of the items, respectively, the histograms followed a multinomial distribution with success probabilities given by: 
\[
p_i = \frac{1}{B_{\text{max}} - B_{\text{min}} + 1} \quad \text{for } i = B_{\text{min}}, \ldots, B_{\text{max}}
\]
and 
\[
p_i = 0 \quad \text{otherwise.}
\]
In total the training set consisted of 2,000 different problems and the validation set included a further 2,000 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 did not need to perform multiple runs with each problem.

\(^3\)The difference is, of course, that here vectors represent problems while previously they could also be interpreted as behaviours.
Table 5.1: Parameters used in the GP experiments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function set (rational problems)</td>
<td>${+, -, \cdot, \div, (\text{protected})}$</td>
</tr>
<tr>
<td>Function set (Boolean problems)</td>
<td>${\text{AND, OR, NAND, NOR}}$</td>
</tr>
<tr>
<td>Terminal set (rational problems)</td>
<td>${x, R}$</td>
</tr>
<tr>
<td>Terminal set (Boolean problems)</td>
<td>${x_1, x_2, x_3, x_4}$</td>
</tr>
<tr>
<td>Random constants (i.e., $R$)</td>
<td>100 real value constants in the interval $[-10, 10]$</td>
</tr>
<tr>
<td>Crossover rate $p_{xo}$</td>
<td>$100%, 90%, 50%,$ and $0%$</td>
</tr>
<tr>
<td>Mutation rate $p_m$</td>
<td>$100%, 50%,$ and $0%$</td>
</tr>
<tr>
<td>Maximum tree depth used in mutation</td>
<td>4</td>
</tr>
<tr>
<td>Maximum number of nodes</td>
<td>262,143</td>
</tr>
<tr>
<td>Selection mechanism</td>
<td>Tournament (size 2) and roulette-wheel</td>
</tr>
<tr>
<td>Population size</td>
<td>1000</td>
</tr>
<tr>
<td>Number of generations</td>
<td>50</td>
</tr>
<tr>
<td>Number of independent runs</td>
<td>100</td>
</tr>
</tbody>
</table>

5.2 Systems and Parameter Settings

5.2.1 GP Systems

We used two different implementations of GP, both tree-based and both using subtree crossover and subtree mutation. One system is essentially identical to the one used by Koza [59], the only significant difference being that we select crossover and mutation points uniformly at random, while [59] used a non-uniform distribution. The other system is TinyGP (see [89]) with the modifications presented in [92] to allow the evolution of constants, which is required in the case of rational symbolic regression problems. The main difference between the two systems is that Koza’s system is generational (all selected individuals in the population reproduce in parallel to produce the next generation) while TinyGP uses a steady-state strategy. That is, as soon as a new offspring is produced, in TinyGP this is inserted in the population and has the potential to become a parent immediately, without having to wait for a full generation to be completed. The second difference between Koza’s system and TinyGP is that Koza’s system uses a roulette-wheel selection whilst TinyGP uses a tournament selection.

Table 5.1 shows the parameters and primitives for the GP systems. To avoid divisions by zero, we used the standard protected division function in the rational symbolic regression problems. Naturally, only combinations of crossover and mutation rates such that $p_{xo} + p_m \leq 100\%$ could be used, since crossover and mutation are mutually exclusive operators in GP. Furthermore, the crossover rate and mutation rate represent the fraction of individuals that are created using crossover or mutation, respectively. For example, 100% mutation means that all the individuals from generation 2 to generation 50 are created using only mutation.

For the first GP system (i.e., Koza’s system), besides the traditional roulette-wheel selection (which gives parents a probability of being chosen for reproduction proportional to their fitness), we also used tournament selection with a tournament size of 2. This form of selection selects parents by randomly forming a group of individuals, then ranking them based on fitness and finally picking the best of the group.
A source of inspirations for these parameters is the configuration used by Koza in [59] Table 6.1. That is, the generational system with no mutation, 90% crossover rate and roulette-wheel selection. The differences between the parameters used by Koza and the ones presented in the table are: the size of the population was 500 and that the number of constant in our implementation is fixed to 100. We decided to use a population of 1000 individuals because the problems used here are more complex to represent than the problems solved in Koza’s book. This GP system was the starting point and we decided to based some of the parameters of the other systems on it. For example, in all the runs at most 50,000 individuals are evaluated.\(^4\)

To sum up, in total we tested three variants of GP: generational with tournament selection, generational with roulette-wheel selection, and steady-state with tournament-selection.

### 5.2.2 GEP Systems

The performance of Gene Expression Programming [28] in preliminary runs was considerably worse than the performance of the worst GP system. Upon closer observation we found that this was largely due to the standard GEP initialisation method. When we replaced it with a technique equivalent to the ramped-half-and-half method in GP [59] we obtained a considerable performance improvement. We, therefore, retained this modification in our experimentation.

We used three different versions of GEP. The first two systems are identical except for the selection mechanism used. In the first one, we used roulette-wheel selection, while in the second we used tournament selection with tournaments of size 2. Both versions are generational, as is standard in GEP. The third version, instead, used a steady-state reproduction strategy with tournament selection (size 2). The parameters and primitives used in our GEP runs are shown in Table 5.2. Most of the parameters used in GEP are equivalent to the ones presented in [28]; however, there are some differences being the most significant that the head length was changed from 6 to 63. It is worth to mention that besides the modifications mentioned previously, we did not make any other effort to tune the parameters of GEP. As a result, it is likely that these setting are not the best possible configuration.

### 5.2.3 CGP Systems

We decided to use the program implemented by [76] as the CGP system, which is a generational system with a particular characteristic that a new population is created by mutating all the members of the current population.

Table 5.3 shows the parameters used in the CGP runs. The table highlight some differences between CGP and the other EPAs systems used (see Section 5.2.2 and Section 5.2.2). Firstly, CGP uses very small

---

\(^4\)Fixing all the runs to 50,000 evaluations may be not the best way to obtain the best performance for all the algorithms tested. In fact, changing this parameter will affect the performance of all the algorithms. However, we decided to fix it firstly because this is the one of the few parameters that appears in all the systems. Secondly, using the a fix number of evaluations gives each algorithm the same change to obtain a suitable solution. Thirdly, this parameter provided us with variation in the performance. This variation in performance is of two forms: the first one is that the average performance of all the systems is different and the second is that within a class of problems the performance of the algorithms varied from problem to problem.
Table 5.2: Parameters used in the GEP experiments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function set (rational problems)</td>
<td>{+, −, *, / protected}</td>
</tr>
<tr>
<td>Function set (Boolean problems)</td>
<td>{AND, OR, NAND, NOR}</td>
</tr>
<tr>
<td>Terminal set (rational problems)</td>
<td>{x, \mathbb{R}}</td>
</tr>
<tr>
<td>Random constants (i.e., \mathbb{R})</td>
<td>100 real value constants in the interval ([-10, 10])</td>
</tr>
<tr>
<td>Terminal set (Boolean problems)</td>
<td>{x_1, x_2, x_3, x_4}</td>
</tr>
<tr>
<td>Head length</td>
<td>63</td>
</tr>
<tr>
<td>Number of genes</td>
<td>3</td>
</tr>
<tr>
<td>Mutation rate (p_m)</td>
<td>5%</td>
</tr>
<tr>
<td>1 point recombination rate</td>
<td>20%</td>
</tr>
<tr>
<td>2 point recombination rate</td>
<td>50%</td>
</tr>
<tr>
<td>Gene recombination rate</td>
<td>10%</td>
</tr>
<tr>
<td>IS-transposition rate</td>
<td>10%</td>
</tr>
<tr>
<td>IS elements length</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>RIS-transposition rate</td>
<td>10%</td>
</tr>
<tr>
<td>RIS elements length</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>Gene transposition rate</td>
<td>10%</td>
</tr>
<tr>
<td>Selection mechanism</td>
<td>Tournament (size 2) and roulette-wheel</td>
</tr>
<tr>
<td>Population size</td>
<td>1000</td>
</tr>
<tr>
<td>Number of generations</td>
<td>50</td>
</tr>
<tr>
<td>Number of independent runs</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 5.3: Parameters used in the CGP experiments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function set (rational problems)</td>
<td>{+, −, *, / protected}</td>
</tr>
<tr>
<td>Function set (Boolean problems)</td>
<td>{AND, OR, NAND, NOR}</td>
</tr>
<tr>
<td>Terminal set (rational problems)</td>
<td>{x, \mathbb{R}}</td>
</tr>
<tr>
<td>Terminal set (Boolean problems)</td>
<td>{x_1, x_2, x_3, x_4}</td>
</tr>
<tr>
<td>Random constants (i.e., \mathbb{R})</td>
<td>5 real value constants in the interval ([-10, 10])</td>
</tr>
<tr>
<td>Mutation rate</td>
<td>1%</td>
</tr>
<tr>
<td>Number of rows</td>
<td>1</td>
</tr>
<tr>
<td>Number of columns</td>
<td>2,000</td>
</tr>
<tr>
<td>Population size</td>
<td>5</td>
</tr>
<tr>
<td>Number of generations</td>
<td>12,500</td>
</tr>
<tr>
<td>Number of independent runs</td>
<td>100</td>
</tr>
</tbody>
</table>

population sizes; however, it evolves for a large number of generations. In fact, the number of individuals evaluated using GP, GEP, CGP and SIHC is kept constant. Secondly, the only genetic operator used by CGP is mutation. Another important difference is that in CGP each new population is created by mutating the best individual found in the previous generation. As a result, the mutation rate refers to the percentage of mutation done to the best individual in order to an offspring. The parameters presented in the table were suggested by Julian Miller; however, he did not have the problems were this system was tested and so he could not make preliminary runs to optimised them. As a result, it is likely that these parameters are not the absolute best configuration to solve these problems.

5.2.4 SIHC Systems

The Stochastic Iterated Hill Climber we used is similar to the one presented in [85, 86] although here we used different mutation operators. We used two types of mutation: sub-tree mutation, which is the same type of mutation used in the GP runs, and a mutation operator similar to the one described in [85, 86],
Table 5.4: Parameters used in the SIHC experiments

<table>
<thead>
<tr>
<th>Mutation</th>
<th>Maximum number of mutations</th>
<th>Maximum number of fitness evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub-tree</td>
<td>50, 500, 1000, and 25000</td>
<td>50000</td>
</tr>
<tr>
<td>Uniform</td>
<td>25000</td>
<td>50000</td>
</tr>
</tbody>
</table>

which we will call *uniform mutation* since all the elements of a tree may be modified. Uniform mutation works as follows. Each node in the tree is mutated with a certain probability. A node selected for mutation can be modified with one of three different operations. First, the node can be replaced with another node of the same type, i.e., a terminal is replaced with another terminal, whilst a function is replaced with another function of the same arity. The second option is to replace it with a randomly generated subtree. This operations also involves replacing one of the children of the random subtree with the tree originally rooted at the mutated node. The third option is to replace the node with one of its children. The child chosen is the one having the most nodes.

The SIHC algorithm starts by creating a random program tree. To create the tree, we adopted the same procedure as for the GP and GEP experiments using the same terminal and function sets as for the GP systems (Table 5.1). SIHC then mutates this tree until a better program is found (i.e., when the fitness of an offspring is better than the parent’s fitness). This then becomes the parent and the process is repeated. When a maximum number of allowed mutations is reached the current individual is set aside, a new random individual is created and the process is repeated. Finally, the algorithm terminates when a solution has been found or when a maximum number of fitness evaluations is reached. The output of the algorithm is the program which scored the highest fitness throughout the search process.

Table 5.4 shows the parameters used for the SIHC experiments.

### 5.2.5 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 [49]. The initial weights and biases for the network were randomly and uniformly chosen in the range \([-0.1, +0.1]\).

The ANN was 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). We took the average of the number of epochs required to train the network in these runs as the performance of the learning algorithm. This process was repeated for all Boolean functions in the training and validation sets.

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 decreasing size. Then, FFD places each item in the first bin where it can fit, while BFD places items where they fit most tightly. As
Table 5.5: Distances or Similarities functions for rational problems

<table>
<thead>
<tr>
<th>Name</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum of Absolute Errors</td>
<td>$\sum_{i}</td>
</tr>
<tr>
<td>Ruzicka similarity</td>
<td>$\frac{\sum_{i} \min(t_i, p_i)}{\sum_{i} \max(t_i, p_i)}$</td>
</tr>
<tr>
<td>Root Mean Square</td>
<td>$\sqrt{\frac{\sum_{i} (t_i + p_i)^2}{\ell}}$</td>
</tr>
<tr>
<td>Canberra distance</td>
<td>$\sum \frac{</td>
</tr>
<tr>
<td>Penrose shape distance</td>
<td>$\sqrt{\sum((t_i - t) - (p_i - p))^2}$</td>
</tr>
<tr>
<td>Lorentzian distance</td>
<td>$\sum \ln(1 +</td>
</tr>
<tr>
<td>Square of the dot product</td>
<td>$(t \cdot p)^2$</td>
</tr>
</tbody>
</table>

mentioned above we used two different bin sizes: 100 and 150.5

5.3 Results

In this section we present the results of the performance models for different EPAs, an ANN training algorithm and two bin backing heuristics. We begin by outlining that the performance models depend on the similarity measures used to create them (see Section 5.3.1). Section 5.3.2 presents the quality of the models for different EPAs. The results for the ANN are presented in Section 5.3.3. And Section 5.3.4 presents the quality of the models for the bin packing problems.

5.3.1 Not all Similarity Measures are Equal

In Section 4.6, we described the procedure used to obtain $S$. Let us remember that $S$ is a set composed by different problems and that these problems need to be identified in order to produce a model with good quality. However, there are other parts of the performance model that can be changed to modify the quality of the model produced. In this section, we study the quality of the models when the closeness measure $d$ is changed. As we will see, some closeness measures can produce models with better quality than others. In order to perform this analysis we used cross-validation (with 5 folds) on the training set (see Section 4.6).

Table 5.5 presents the closeness measures used for the rational problems. In the equations $t_i$ represents the $i$-th component of $t$. Let us remember that $t$ and $p$ represent problems; the difference being that $t$ stands for the problem being solved and $p$ is a problem in $S$. Furthermore, in all the cases presented here, both of them are vectors. $\ell$ is the number of inputs, and $\cdot$ is the dot product between vectors. All the functions presented in the table are distance functions except $(t \cdot p)^2$.

Table 5.6 shows the distance or similarity functions used for the Boolean problems. All the functions are a distance except the square of the dot product and the cube of the dot product. Most of the closeness

---

5When the size of the bins is 150 both heuristics gave the same performance, therefore in the experiments we only present the results for FFD.
Table 5.6: Distances or Similarities for Boolean problems

<table>
<thead>
<tr>
<th>Name</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jaccard distance</td>
<td>[ \frac{\sum_j (1-\delta(t_j, p_j))}{t \cdot p + \sum_j (1-\delta(t_j, p_j)) + \ell} ]</td>
</tr>
<tr>
<td>Kulsinski distance</td>
<td>[ \frac{2 \sum_j (1-\delta(t_j, p_j))}{\sum_j \delta(t_j, p_j) + 2 \sum_j (1-\delta(t_j, p_j))} ]</td>
</tr>
<tr>
<td>Rogers-Tanimoto similarity</td>
<td>[ \frac{4 \sum_j (1-\delta(t_j, p_j))}{\sum_j \delta(t_j, p_j) + \sum_j (1-\delta(t_j, p_j))} ]</td>
</tr>
<tr>
<td>Sokal-Michener similarity</td>
<td>[ \frac{4 \sum_j (1-\delta(t_j, p_j))}{t \cdot p + \sum_j (1-\delta(t_j, p_j)) + \ell} ]</td>
</tr>
<tr>
<td>Sokal-Sneath similarity</td>
<td>[ \frac{4 \sum_j (1-\delta(t_j, p_j))}{t \cdot p + \sum_j (1-\delta(t_j, p_j)) + \ell} ]</td>
</tr>
<tr>
<td>Square of the dot product</td>
<td>[ (t \cdot p)^2 ]</td>
</tr>
<tr>
<td>Cubic of the dot product</td>
<td>[ (t \cdot p)^3 ]</td>
</tr>
</tbody>
</table>

Table 5.7: Quality of the model for different closeness measures (GP with 90% crossover)

<table>
<thead>
<tr>
<th></th>
<th>Rational Functions</th>
<th>Boolean Functions</th>
<th>Success rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BRF</td>
<td>NBRF</td>
<td></td>
</tr>
<tr>
<td>Sum of Abs. Errors</td>
<td>55</td>
<td>0.0215</td>
<td>131</td>
</tr>
<tr>
<td>Ruzicka</td>
<td>6</td>
<td>0.4432</td>
<td>398</td>
</tr>
<tr>
<td>Root Mean Square</td>
<td>26</td>
<td>0.0292</td>
<td>251</td>
</tr>
<tr>
<td>Canberra</td>
<td>49</td>
<td>0.9536</td>
<td>150</td>
</tr>
<tr>
<td>Penrose</td>
<td>69</td>
<td>0.0278</td>
<td>251</td>
</tr>
<tr>
<td>Lorentzian</td>
<td>399</td>
<td>0.5621</td>
<td>126</td>
</tr>
<tr>
<td>Sqr. dot product</td>
<td>5</td>
<td>0.1197</td>
<td>48</td>
</tr>
</tbody>
</table>

The data in the table are for the Koza-style GP with 90% crossover rate and no mutation.

As expected, the quality of the models depends on the closeness measure used. In particular the table reports the cross-validation RSE of the models obtained with two performance measures — the best of run fitness (BRF) and normalised best of run fitness (NBRF) — for the rational symbolic regression problems and one performance measure — the success rate — for the Boolean induction problems. The data in the table are for the Koza-style GP with 90% crossover rate and no mutation.

Table 5.7 shows the quality of the model when different closeness measures are used. In particular the table reports the cross-validation RSE of the models obtained with two performance measures — the best of run fitness (BRF) and normalised best of run fitness (NBRF) — for the rational symbolic regression problems and one performance measure — the success rate — for the Boolean induction problems. The data in the table are for the Koza-style GP with 90% crossover rate and no mutation.

As expected, the quality of the models depends on the closeness measure used. As can be observed from Table 5.7, for the case of rational problems using NBRF as performance, Lorentzian distance was the similarity measure having the lowest RSE among all closeness measures tested. However, there is only a small difference with the RSE in the sum of absolute errors which presents the lowest RSE when the performance is BRF. For the case of Boolean problems the square of the dot product obtained the lowest RSE.

Although Table 5.7 only shows the quality of the models for one version of GP, we can see that different closeness measures do produce models with different qualities. Unfortunately, given that the table only

\[ \delta(a, b) = \begin{cases} 
1 & \text{if } a = b \\
0 & \text{otherwise}
\end{cases} \]
presents one system, we cannot discard any closeness measure or assume that one particular closeness measure produces the model with the best quality. In order to complement this information, Table 5.8 presents the mean RSE obtained by averaging the RSE of 21 different models, each one corresponding to a different EPA.

From Table 5.8, it is observed that the sum of absolute errors presents the lowest RSE when the performance is BRF and the second lowest for the NBRF. When the performance is NBRF the lowest RSE was for the Lorentzian distance. For the Boolean problems, the square of dot product was the closeness which had the lowest RSE.

All the closeness measures have a RSE average well below 1, which means that these models are predicting better than the mean. The only exception, as can be observed from Table 5.8, is Ruzicka with NBRF as performance. This bad accuracy can be understood by looking at the results of computing the distance between any pair of problems: almost all the comparisons produce similar values. Therefore, this closeness measure does not provide enough variety to produce a good model. Another way of looking at this is by plotting an histogram of these values. This histogram has one high peak and is almost zero everywhere else.

### 5.3.2 Performance of Performance Models of EPAs

RSE figures provide an objective indication of model quality. However, it may be difficult for the reader to appreciate the accuracy of our models from just such figures. In order to provide a more visual indication of the quality of our models, Figure 5.1 shows scatter plots of the actual performance vs the performance estimates obtained by the model in the training and validation sets for the symbolic regression of rational functions (for both performance measure used) and Boolean function induction problems. To reiterate, the models used as closeness measures the best measures identified in Table 5.7. The data in Figure 5.1 refer to the GP system with 90% crossover rate, no mutation, and roulette-wheel selection, but other parameter settings provided qualitatively similar results. The solid diagonal line in each plot represents the behaviour of a perfect model. As can be seen, the points form tight clouds around the perfect models which is a clear qualitative indication of the accuracy of the predictions of our performance models.

Table 5.9 presents an accuracy comparison (in terms of RSE) for the performance models of the different
Figure 5.1: Scatter plots of the performance measured vs the performance obtained from the model for continuous regression problems using BRF (a & b) and normalized 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.

program induction systems presented in Section 5.2, namely GP, GEP, CGP and SIHC, across a range of parameter settings. The models were created using the closeness measure produced by the model with lowest RSE in the cross-validation. The closeness measures having the lowest RSE in Koza-style GP system (see Table 5.7) are the ones utilised in the majority of models.  

The models using closeness measures different that the ones reported in Table 5.7 are: TinyGP with \( p_{x0} = 90\% \) and the SIHC with uniform mutation both using the BRF performance measure and the Lorentzian distance as similarity measure. For the NBRF performance measure the sum of absolute errors distance was used in the performance models of the Koza-style GP.
Table 5.9: Quality of the model (RSE) for different GP, GEP, CGP and SIHC systems and parameter settings

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Rational Functions</th>
<th>Boolean Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best of Run Fitness</td>
<td>Normalized BRF</td>
</tr>
<tr>
<td>Type Selection</td>
<td>$p_{cr}$</td>
<td>$p_m$</td>
</tr>
<tr>
<td>Generational</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Roulette</td>
<td>0.90</td>
<td>0.00</td>
</tr>
<tr>
<td>GEP</td>
<td>0.50</td>
<td>0.50</td>
</tr>
<tr>
<td>SIHC</td>
<td>0.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 5.9 is divided into five blocks. The first three blocks contain the different EPAs under study: the generational systems with roulette-wheel selection, the generational systems with tournament selection, and the steady state systems with tournament selection, respectively. The last row in each block corresponds to the GEP system. The fourth block in the table contains SIHC systems with the two mutation operators and, for the case of subtree-mutation, with different maximum numbers of mutations before restarts. The last block contains the CGP system.

The lowest RSE values in Table 5.9 are associated with the generational systems of roulette-wheel selection and to the BRF performance measure. However, in all the cases we see that the RSE is well below 1 in both training and validation sets, implying that our performance models are able to predict the performance of the different systems tested much better than the mean.

From the table, it is observed that the values of $|S|$ vary from system to system. This behaviour is expected given that the size of $|S|$ is obtained using a cross-validation technique in the training set (see Section 4.6). The idea of using a cross-validation technique is to optimise the accuracy of the models. That is, the values of $|S|$ correspond to the best models found in the cross-validation.

From the table, one can also observed that the accuracy of the models (RSE) changed when the parameters with tournament selection and $p_m = 100\%$, and the different SIHC systems. For the case of Boolean induction problems the Kulinsiki similarity was used in the models of the generational GEP systems and the Sokal-Sneath similarity function was used in the model of the steady state GEP system and the CGP system.
of the algorithm are varied. For example, let us focused at the first two generational GP systems with roulette-wheel selection. There we have changed the crossover rate from 100% to 90%. As can be observed, these two systems have different RSE values in the training and validation set. Comparing all the algorithms whose parameters were modified, we can see that each modification to an algorithm produces a variation on the respective RSE values. This is an indication that our models are sensitive with the parameters of the algorithms; however, as mentioned previously, in all the cases the RSE values indicated that our modelling technique makes accurate predictions.

As mentioned previously, we decided to model all these different algorithms to experimentally show that our modelling technique is able to produce accurate predictions and that it is not restricted to a particular algorithm or an algorithm with certain parameters. However, there is a particular case where it is not even necessary to instantiate our performance model or any other model of performance. This case is where the performance of the algorithm is constant independently of the problem solved. For example, in the case of Boolean induction problems, this case is present when the algorithm finds a perfect solution in every run. As can be seen, for this case is not necessary to instantiate the model because the model is a constant, i.e., \( P(t) = \text{constant} \). As expected, there are different ways one can end up in this situation being one of them when the algorithm evaluates too many individuals and so in every run a perfect solution is found.

In the next sections we will extend the application of our models to some learners and problem solvers other than program-induction algorithms. More specifically we will model the training of an ANN and two human-designed heuristics for the off-line bin packing problem.

5.3.3 Artificial Neural Networks Results

As indicated previously, to create models of ANN training algorithms we can use the same models and, thus, the same distance measures used for program induction. We can also follow an equivalent procedure to the one used in Section 4.6 to identify performance models.

Table 5.10 shows the quality of the model of ANN training resulting from the use of different similarities measures. The table reports the RSE obtained using the cross-validation technique (column 2), as well as the quality obtained on the training and validation sets (columns 3 and 4, respectively). The similarity measure producing models with the best quality using the cross-validation technique is \((t \cdot p)^2\), which also corresponds to the lowest RSE on the validation set.

In Figure 5.2 we show a scatter plot of the actual performance vs the performance estimated by the model (when using \((t \cdot p)^2\) as a similarity measure). As can be observed from the figure, the performance model is able to accurately predict actual performance for most problems. The model only starts to significantly underestimate performance problems where training took longer than approximately 15 epochs. The reason for this behaviour is that there are only very few problems requiring a high number of learning epochs in the training set (and, in fact, also in the validation set). Naturally, if accurately predicting the performance of
Table 5.10: Quality of the models of ANN training for different closeness measures

<table>
<thead>
<tr>
<th>Closeness measure</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$</td>
<td>S</td>
<td>$</td>
</tr>
<tr>
<td>Jaccard</td>
<td>228</td>
<td>0.4195</td>
<td>0.0664</td>
</tr>
<tr>
<td>Kulsinski</td>
<td>215</td>
<td>0.4133</td>
<td>0.0663</td>
</tr>
<tr>
<td>Rogers-Tanimoto</td>
<td>243</td>
<td>0.4286</td>
<td>0.0546</td>
</tr>
<tr>
<td>Sokal-Michener</td>
<td>400</td>
<td>0.4421</td>
<td>0.0031</td>
</tr>
<tr>
<td>Sokal-Sneath</td>
<td>400</td>
<td>0.4840</td>
<td>0.0033</td>
</tr>
<tr>
<td>Sqr. dot product</td>
<td>135</td>
<td><strong>0.3711</strong></td>
<td>0.1814</td>
</tr>
<tr>
<td>Cubic dot product</td>
<td>190</td>
<td>0.4798</td>
<td>0.1220</td>
</tr>
</tbody>
</table>

Figure 5.2: Scatter plots of the actual performance (epochs) vs the performance estimated by the model for ANN training problems.

rare cases (such as particularly 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.

5.3.4 Off-line Bin Packing Results

Naturally for bin-packing, we start our modelling effort by testing the quality of models obtained with the closeness measures used for the other systems. Table 5.11 presents the RSE values for the FFD and BFD heuristics for two bin sizes (100 and 150) when different similarities measures were used. The table includes the size of set $S$, the RSE value obtained using cross-validation, and the RSEs for the training and validation sets. In all cases and for all similarities measures our models predict very accurately the performance of the two bin-packers we considered and generalise very well. However, the square of the dot product was the similarity with the lowest RSE for all the configurations. As can be observed from the table, the size of $S$ is considerably bigger than for the other models presented above. This is likely to be due to the much larger number of degrees of freedom associated with 100-dimensional vectors. This is, of course, no problem, since the calculations involved in the model can still be performed in milliseconds.

In Figure 5.3 we show scatter plots of the actual performance vs the estimated performance in the validation set for the FFD bin packer with two bin sizes. As can be observed from the figure, the data
### Table 5.11: Quality of the model for the different heuristics and similarity measures in the off-line bin packing problem.

| Configuration | Distance | $|S|$ | RSE  | $T$ set | $V$ set |
|---------------|----------|-----|------|--------|--------|
| FFD 150       | Sum of Absolute Errors | 1315 | 0.0031 | 0.0002 | 0.0028 |
|               | Ruzicka   | 1578 | 0.0047 | 0.0000 | 0.0037 |
|               | Root Mean Square | 399  | 0.0216 | 0.0010 | 0.0030 |
|               | Penrose   | 1599 | 0.0031 | 0.0000 | 0.0023 |
|               | Lorentzian| 383  | 0.0487 | 0.0114 | 0.0252 |
|               | Square of the dot product | 557  | 0.0006 | 0.0003 | 0.0005 |
| FFD 100       | Sum of Absolute Errors | 828  | 0.1405 | 0.0414 | 0.1413 |
|               | Ruzicka   | 1548 | 0.1354 | 0.0046 | 0.1240 |
|               | Root Mean Square | 399  | 0.2684 | 0.0528 | 0.0974 |
|               | Penrose   | 1599 | 0.0945 | 0.0009 | 0.0849 |
|               | Lorentzian| 364  | 0.7028 | 0.2044 | 0.3840 |
|               | Square of the dot product | 1284 | 0.0622 | 0.0095 | 0.0556 |
| BFD 100       | Sum of Absolute Errors | 855  | 0.1342 | 0.0377 | 0.1361 |
|               | Ruzicka   | 1561 | 0.1274 | 0.0044 | 0.1176 |
|               | Root Mean Square | 399  | 0.2490 | 0.0520 | 0.0938 |
|               | Penrose   | 1599 | 0.0884 | 0.0009 | 0.0810 |
|               | Lorentzian| 370  | 0.6436 | 0.1947 | 0.3642 |
|               | Square of the dot product | 1267 | 0.0582 | 0.0098 | 0.0549 |

Closely follow the perfect-model line.

### 5.4 Summary

This chapter evaluates the quality of the performance models for different EPAs, ANN training and bin packing heuristics. The algorithms modelled include: three version of GP with four different parameter settings each; three versions of GEP; one version of CGP; two versions of SIHC (one with four different parameters settings); one ANN learning algorithm; and two human designed heuristics for the off-line bin packing problem. Furthermore, our tests considered four different classes of problems: symbolic regression of rational functions and Boolean induction (4 inputs and 1 output), which were used with the GP, GEP and SIHC systems; another version of Boolean induction (3 inputs and 2 outputs) which was used in the ANN’s training; and several variants of the off-line bin packing problem.

All the models created present a RSE value lower than 1 meaning that all our models predict better than the mean. This quantitative information was complemented with the scatter plots of the actual performance vs the estimated performance for some of the models created. These showed that the predictions made closely follow the actual values of performance.

In the next chapter we will consider the applicability of our models in the context of the algorithm selection problem. In other words, we will present how our modelling technique can be used to tackle the algorithm selection problem.
Figure 5.3: 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 and using the square of the dot products as $d$. 
Chapter 6

Performance Models and Algorithm Selection Problem

In Chapter 4, we described the procedure used to arrive at the performance model. Chapter 5 has experimentally shown that this modelling technique makes accurate predictions. Furthermore, as mentioned previously, the time needed to make a prediction is of the order of milliseconds with ordinary computational power. Clearly, the next step is to take advantage of this model. In this chapter, we will show that the predictions made by the performance model can be used to solve the algorithm selection problem. That is, the predictions will be used to decide which algorithm to run to solve a problem given a set of different algorithms with different parameters setting. In other words, we will test the applicability of the performance model in the construction of an algorithm portfolio.

Before creating an algorithm portfolio, we will define the algorithm selection problem. We devote the next section to this and describing the prerequisites needed to solve it. Section 6.2 presents related work. This is followed by a discussion of the similarities and difference between our modelling technique and other approaches found in the algorithm selection literature (Section 6.3). Finally, Section 6.4 presents the methodology we used to tackle the algorithm selection problem.

6.1 Algorithm Selection Problem

The process of solving a problem often starts by looking at the different procedures available to do the job and deciding which one of them to apply. In automated problem solving this is known as the *algorithm selection problem* [102].

The algorithm selection process is depicted in Figure 6.1. There, we have a set of different algorithms (right) and for each problem the selection mechanism (middle box) needs to select which algorithm to use to solve the problem. Generally, the process of selecting which algorithm to use is done manually.
Figure 6.1: Algorithm Selection

This decision process is inferred by prior knowledge. That is, an algorithm is selected based on the experience one has in relation to the problem or the algorithm; it is also common that the algorithm is chosen from procedures that have shown some success in solving “similar” problems. For example, a person familiar with EAs might try to use EAs to solve the problem. However, this technique is far from the ideal scenario where one would like to know the most suitable algorithm to solve a particular problem in advance.

However, the task can be automatised if one has a way to predict the performance of each algorithm on the right when solving the problem on the left.

That is, let $t$ be a particular problem to be solved, $\hat{P}(a,t)$ be a function that predict the performance of algorithm $a$ on problem $t$ and $\Delta$ be a set of different algorithm that can solve problem $a$. Clearly, if one had $\hat{P}$ then the optimal selection mechanism is just the result of $\arg\max_{a \in \Delta} \hat{P}(a,t)$. Given that our performance model is specialised in predicting the performance of algorithms, we can set $\hat{P}(a,t)$ to be the performance model instantiated to model algorithm $a$.

Before continuing with our description of how our performance model can be used to tackle the algorithm selection problem, we will list the prerequisites (as described in [117]) needed to solve it. These are:

1. A large collection of problem instances of variable complexities.
2. A diverse set of algorithms, each having the best performance on some of the instances of the collection.
3. A measure to evaluate the performance of each algorithm in the set.
4. Finally, a set of features that describe the properties of each problem in the collection.

These elements are necessary because most approaches to the algorithm selection problem use machine learning techniques which attempt to predict which algorithm from the collection will have the best performance, starting from a description of the problem to be solved. Each problem is described using a set of features which, together, may be seen as a problem signature.

Naturally for this to work well, the feature set must be good. The idea is that an algorithm should have a similar performance on problems having similar features. Therefore, generally, the features are selected by an expert in the field or a person that is familiar with the class of problems in the collection. With very few exceptions, these features are problem-specific; that is, they depend on the class of problems. For example, in the Boolean satisfiability (SAT) problem the feature set might include the number of variables,
the number of clauses, the ratio between variables and clauses, and so on. In other problems completely different features will typically have to be used.

Since techniques to solve the algorithm selection problem tend to rely on some form of prediction of algorithm performance, they bear some similarities with our approach. However, there are also important differences. We will explicitly highlight these similarities and differences in Section 6.3.

6.2 Related Work

The methodologies used to solve the algorithm selection problem can be divided into two groups, depending on when the decision of which 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. We will survey techniques for dynamic selection in Section 6.2.1, while we will look at static selection, which is more relevant to the work presented here, in Section 6.2.2.\(^1\)

6.2.1 Dynamic Selection

One algorithm that uses a prediction model to guide the search process dynamically is STAGE\(^1\)[11]. STAGE is a two steps algorithm composed by a hill climber and a heuristic that decides where to start the search. The hill climber starts from a random point. It is allowed to run for a fixed number of steps. When it stops, STAGE uses the points sampled to produce or update the prediction model of the heuristic. Linear regression is used as a model. The heuristic is used to find a promising point in the search space to restart the hill climber. This process is repeated until a stopping criterion is met. STAGE was successfully tested on SAT problems. Note that at the end of a run of STAGE, a prediction model is obtained. This model, however, can only be used to predict the performance of the algorithm on the same problem instance for which it was originally created.

Following a similar idea, in [123] a prediction model was used to guide search in a search tree. That is, the model was used to decide which path in the search tree to follow. This approach was tested on Knapsack problems and set partitioning problems. This work also used linear regression.

Working on SAT problems, in [62] the algorithm selection problem was modelled as a Markov decision process (MDP). The Davis-Putnam-Logemann-Loveland algorithm was enhanced with this modelling technique to decide, for each node of the search tree, which branching rule to use next. Similarly, in [45], a Bayesian model was created to predict the run time of an algorithm. This prediction was used to decide when the search should be restarted.

When different recursive algorithms are available to solve a problem, one can modify each algorithm to decide which algorithm from the set of recursive algorithms to call next, instead of using the default\(^1\)\footnote{For a detailed discussion of how the algorithm selection problem has been tackled in different fields we refer the reader to [117].}
behaviour of always calling itself. This idea was suggested in [61] where a MDP was used to make the decision. Testing on sorting algorithms and on the order statistic selection problem gave good results.

6.2.2 Static Selection

Static-selection methodologies do not alter the course of an algorithm; instead they first decide which procedure to use and then wait until the chosen algorithm terminates. A substantial body of work focusing on predicting the run-time of algorithms on problems like matrix multiplication [135, 124], sorting [12, 135, 124], solving partial differential equations [12] and signal processing [116] falls in this category. These approaches attempt to solve at run-time the problem of choosing the best implementation of an algorithm to use to perform a calculation most efficiently. The decision is based on factors like the computer’s architecture, the workload, and the instance size. Some methodologies focusing on NP-hard problems have also been proposed. Predicting the run-time of SAT solvers was presented in [81, 145, 144, 146, 46], where two different machine learning techniques were used: linear regression [81] and a combination of linear regression and a mixture-of-experts [146, 145, 144]. Similar methodologies based on linear regression were used in [69, 68, 67, 66, 65] to solve the auction winner determination problem. The algorithm selection problem was also studied in relation to a constraint programming problem — the social golf problem — in [35, 34, 33] using case-based reasoning.

Algorithm portfolios [69, 68, 46, 81, 65, 66, 67, 146, 145, 144] 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. Models of performance are typically employed to build algorithm portfolios. Given a new problem instance, models are used to rank algorithms from best to worst. Typically, the three top ranking algorithms are then executed in sequence. The execution of the portfolio stops as soon as a solution is found. The idea is to match a problem instance with a good set of algorithms for that instance. As a result, the average performance of the portfolio over a set of instances is often better than the average performance of any algorithm in the portfolio. Besides using models of performance to solve the algorithm selection problem, in [69, 68, 67] such models were used to identify sets of particularly hard problems for an algorithm.

As we indicated at the beginning of this section, when addressing the algorithm selection problem, normally the features fed into the models are obtained by an expert (perhaps via a careful analysis of the literature on a particular class of problems). In [15], 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. The machine learning technique used in [15] was a Bayesian classifier and the problem domain was job shop scheduling. Following a similar approach, in [70] preliminary runs were used to measure the expected time that the algorithm would spend in processing a node in a search tree. This time was used in conjunction with Knuth’s method
to estimate the size of a search tree and therefore to predict the run-time of different algorithms.

6.3 Similarities and Differences with our Model

As we have seen above, the algorithm selection problem is generally approached using some form of machine learning technique in order to predict the performance of a collection of algorithms (and then use this information to match problems with algorithms). From this point of view, the similarity between our objectives and those of prior work on the algorithm selection problem is evident.

For instance, as we have mentioned previously, most of the equations used in the algorithm selection problem are linear. Commonly, it has the form:

\[ P(x) \approx w_0 + \sum_{i} w_i x_i, \]  

(6.1)

where \( x \) represents the features of the problem, \( \ell \) represents the number of features and \( w_i \) are coefficients. Given that Equation (6.1) and the information landscape (Equation (2.3)) are both linear models, one can see that there is a strong similarity between these two equations. However, the difference is that the information landscape uses an exact re-representation of the fitness function while the components of Equation (6.1) are features that somehow describe the problem. Furthermore, these approaches bear some similarity with our performance model given that the performance model also uses a linear equation. However, here, we see a considerable difference, which is that our performance model (Equation 4.5) uses a set of reference problems (i.e. set \( S \)) and a closeness measure to make the predictions.

Moreover, our methodology presents significant differences with respect to such works. Firstly, we focus primarily on program induction and more specifically EPAs (although as we have seen in Chapter 5.3 our method extends naturally to other domains). This is an area neglected by prior work on algorithm selection.

Secondly, in related work two different techniques have been used to characterise problems. One is to use problem-specific features that experts in the problem domain consider to be useful hardness measures for the problem. The other is to compute features based on preliminary runs of the algorithms being modelled. Both methods are good, but the first requires considerable domain knowledge (and effectively relies on previous attempts to characterise hardness), while the second produces models that lack generality having been derived for one specific problem instance. Instead, our approach in Equation (4.5) represents problems via features that simply measure how similar a problem is to a set of reference problems, which are automatically identified via LARS and cross-validation. Therefore, our features are generic rather than problem-specific and the models of algorithms we obtain are applicable to whole classes of problems, not single instances.

A final difference between our work and prior work is that we do not just use performance models to predict performance. As we will see in Chapter 7, we elicit important knowledge on the similarities and
differences between algorithms from such models.

6.4 Program-induction Portfolios

Despite some good attempts (e.g., the \textit{nsc} coefficient reviewed in Section 2.3), so far researchers have had relatively little success in deriving practical features to characterise the difficulty of program induction problems. It is then not surprising that no successful attempt to extend the work on portfolios to such a domain has been reported. The good results obtained with our models, however, suggest that they might allow such an extension.

To put this idea to the test, we decided to attempt to develop algorithm portfolios that could be used to solve the problem of optimally choosing program induction algorithms for two classes of problems: symbolic regression and Boolean function induction.

We built portfolios following a procedure similar to one presented in [69]. It was suggested in [69] that the algorithms forming the portfolio should behave differently on different problem. Ideally all the algorithms of the portfolio should beat all other algorithms in the portfolio on some problems. So, for our tests we decided to form portfolios using a \textit{subset} of the 21 program induction algorithms considered in Section 5.3. For each problem class, in order to select which algorithms to insert in the portfolio we looked at each algorithm’s actual performance on the problems in the training set. We then included in the portfolio all the algorithms that had the best performance in at least one problem of the set. This procedure resulted in 7 algorithms for the symbolic regression problem and 18 algorithms for Boolean induction. For the case of Boolean induction problems, we decided to remove from the portfolio the CGP system because this system has an extremely good performance (0.9879). That is, it finds a perfect solution in almost every run. The consequence of this high performance is that the difference in performance between the performance of the best portfolio and the performance of CGP is not statistically significant with a 95% confidence interval. As a result, for this case is not necessary to create a portfolio. However, we want to illustrate that our modelling technique is able to produce a portfolio that have a better performance than the performance of the algorithms composing the portfolio instead of just showing that the performance models are able to produce a portfolio whose performance is statistically equivalent to the performance of CGP which in fact it does.

A question that naturally comes to mind is whether we should keep all these algorithms in the portfolios or whether we could perhaps discard those algorithms that produced the best result in a very small number of problems. If one went for the latter, where should we set the threshold? In order to free the user from this dilemma, we decided to use cross validation to identify an optimal threshold. The procedure is as follows.

We predicted the performance of the training set using a cross-validation technique equivalent to the one used in Section 4.6. We started by creating a portfolio having only one algorithm: the one that resulted best in the biggest number of problems in the training set. Then, we added a second algorithm to the portfolio:
the one that was overall second on the training set. We then 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. After that we simulated running the predicted best algorithm on each problem and averaging the resulting performance values. This gave us the average performance of the portfolio in the training set. We then added the third, fourth, etc. best algorithm to the portfolio, repeating the phases above until all the algorithms were included which were best on at least one problem.

Figure 6.2 shows the performance of the algorithm portfolios constructed using the methodology described above. Each plot is composed by four curves. The curves labelled “performance model”, “perfect model” and “best algorithm” represent different mechanisms for deciding which algorithm from the portfolio to use for each problem in the training set. The “performance model” curve shows the portfolio’s performance when cross-validation is used in conjunction with our model to select the algorithm. The “perfect model” curve shows the portfolio’s performance that would be obtained if a perfect model of each algorithm’s performance was available to make this choice. The “best algorithm” curve presents the performance of the portfolio when we always select the algorithm with the best average performance across the training set. There is a fourth curve in the plots, labelled “mean of the portfolio”. This does not represent an algorithm selection method: it is simply the average performance of the algorithms composing the portfolio.

If we focus on the “performance model” curves, we can see from the figure that the best performance for symbolic regression of rational functions is obtained when the portfolio is composed of just two algorithms. These are the TinyGP system with 100% mutation and the SIHC with 25,000 maximum mutations. For the case of Boolean function induction, the portfolio’s best performance was found when four algorithms are considered: the three TinyGP systems with no reproduction (i.e., for which \( p_{zo} + p_m = 100\% \)) and the steady state GEP system. In all cases the performance of portfolios based on our model is superior to the performance of the best algorithm. At this point, it is important to note that even though the portfolios created have the best performance among all the systems tested, this does not mean that the algorithms involved in the portfolio nor the portfolio itself are the best configurations to solve these problems. Furthermore, it may be the case that an algorithm that was not even included in the portfolio could obtained a better performance than the ones presented here if the parameters of that algorithm were optimised for these problems. This behaviour is expected because, as mentioned previously (see Section 5.2), we did not optimise the parameters of the algorithms used here.

The results presented in Figure 6.2 require a little discussion. 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 of the portfolio. Instead, what we see is that as we increase the number of algorithms, the portfolio’s performance rapidly peaks and then either remains stable (as for the Boolean

---

2Since we had already run all algorithms on all problems to create our training set, this phase amounted to a simple look up operation.

3Let us remember that CGP was removed from the portfolio. If this was not the case, the portfolio would be composed by the CGP system and steady state GEP system.
induction problems) or slightly decreases (as for the symbolic regression of rational functions). To explain this behaviour we need to consider that we have ordered algorithms based on the number of problems in the training set where they resulted best among the 21 algorithms in our pool. Therefore, as we add more and more algorithms to the portfolio effectively the average portfolio’s performance is likely to decrease, as is illustrated by the “mean of the portfolio” curves in Figure 6.2. These curves give us the performance we should expect to obtain if the choice of algorithm within a portfolio was uninformed (e.g., random). So, the difference between these curves and the “performance model” curves represent the improvement in selection ability provided by our models.

The selection improvements for both classes of problems are depicted in Figure 6.3. The figure shows the ratio between the performance of the portfolio using as predictor the “performance model” and the “mean of the portfolio”. From the figure one can observed that in both cases the improvement\(^4\) is very significant irrespective of the size of the portfolio and in fact increases as the selection problem becomes harder (as we add algorithms to the portfolio the risk of making a wrong prediction increases). We can see that in both cases the tendency of the ratio shows that our prediction model is getting better in comparison to an heuristic that randomly selects an algorithm from the portfolio.

Having chosen portfolios via cross-validation, we can now see whether our portfolios perform and generalise well by testing them on the validation set. Table 6.1 shows the average portfolio’s performance when our performance models are used to select algorithms against the performance provided by an ideal perfect model. The values reported in the table are very close to those shown in Figure 6.2 which were obtained via cross-validation on the training set. Again our method provides a performance which is very similar to that of an ideal model, and is far better than the performance of the best algorithm composing the portfolio.

\(^4\)Note that for the figure on the left low is good while for the figure in the right high is good.
Table 6.1: Performance of our algorithm portfolios on the validation set

<table>
<thead>
<tr>
<th>Portfolio</th>
<th>Rational Functions</th>
<th>Boolean Induction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. Dev.</td>
</tr>
<tr>
<td>Performance Model</td>
<td>0.6261</td>
<td>1.0139</td>
</tr>
<tr>
<td>Perfect model</td>
<td>0.5702</td>
<td>0.9466</td>
</tr>
<tr>
<td>Best algorithm</td>
<td>0.6816</td>
<td>1.2105</td>
</tr>
</tbody>
</table>

6.5 Summary

In this chapter we presented an application of our performance model. That is, we showed how the performance model can be used to tackle the algorithm selection problem. To exemplify its applicability, we built a program-induction portfolio. It is worth mentioning that to the best of our knowledge this is the first time a program-induction portfolio has been proposed in the literature. This portfolio clearly had a better average performance than any algorithm composing the portfolio. Besides, its average performance was very close to the performance of the portfolio using a perfect model as predictor.

Our modelling technique shares some similarities with the models used to solve the algorithm selection problem; however, there are also some important differences. One of them is that our technique is not specific to a class of problems (like program-induction algorithms) and it can be applied to a variety of domains as was shown in Chapter 5.3. Given that the models found in the algorithm selection literature are specialised to work in a particular domain, they cannot be applied to other classes of problems unless they undergo a number of modifications.
Chapter 7

Eliciting Knowledge from Performance Models

So far, we have shown, through a series of experiments, that the predictions made by our performance models are accurate and can be used to tackle the algorithm selection problem. In this chapter, we will start analysing how the performance model can be used to study the behaviour of program-induction algorithms. That is, we are interested in what can be learnt from an analysis made using the performance models.

This chapter will particularly focus upon the applicability of our modelling technique in the automatic creation of taxonomies. We will describe the process of automatically creating a taxonomy after having created a performance model for each of the algorithms involved in the taxonomy. We will provide examples of our procedure by presenting taxonomies formed by different evolutionary program-induction algorithms with different parameters setting.

The chapter is organised as follows. Section 7.1 describes how our performance model can be used to measure the similarity between algorithms. Section 7.2 presents related work. The procedure used to create taxonomies is described in Section 7.3. Sections 7.4 and 7.5 present the taxonomies for different program-induction algorithms. In Section 7.6, we describe what the sort of information that can be gathered by following traditional approaches and compare this with the information obtained using our methodology.

7.1 Comparing Algorithms and Parameter Settings

When considering different algorithms to solve a problem it is important to understand what the similarities and differences are in the behaviour of such algorithms. Since our models capture what is important performance-wise in an algorithm, it seems reasonable to attempt to infer some such similarities and differences via a direct comparison between model equations.

The performance model (see Equation (4.5)) can be used to compare the similarity between systems. In
order to do so, Equation (4.5) will be re-written in a more suitable manner. In this section we present two different methods with which to measure the similarity between systems.

In particular we will highlight the fact that Equation (4.5) represents a hyperplane and, therefore, one can compare systems using the angle between the hyperplane representations of their models. We will also see that an algorithm is characterised by a vector, and thus, we can compute the distance between the vectors representing two algorithms to measure their similarity.

### 7.1.1 Using Angles to Measure Similarity

As mentioned previously, Equation (4.5) represents a hyperplane. To see this more clearly, we re-write Equation (4.5) in normal form using the scalar product between two vectors, namely

\[
(-1, a_{p_1}, \cdots, a_{p_{|S|}}) \cdot ((P(t), d(p_1, t), \cdots, d(p_{|S|}, t)) - x_0) = 0
\]

where \( \cdot \) is the scalar product, \( p_1, \cdots, p_{|S|} \) are the elements of \( S \) and \( x_0 = (a_0, 0, \ldots, 0) \) is a particular point on the hyperplane which depends only on the constant term \( a_0 \). Then, to compute the angle between two hyperplanes written in normal form, one only needs the normal vector \((-1, a_{p_1}, \cdots, a_{p_{|S|}})\). Therefore, we can say that an algorithm is characterised by the normal vector, and so, to measure the similarity between two algorithms, one represented by the vector \( n' = (-1, a'_{p_1}, \cdots, a'_{p_{|S|}}) \) and the other by the vector \( n'' = (-1, a''_{p_1}, \cdots, a''_{p_{|S|}}) \), we can simply compute the angle between the two vectors, namely

\[
\alpha = \arccos \left( \frac{n' \cdot n''}{||n'|| ||n''||} \right).
\]

Here angles are measured in radians.

Naturally, for this idea to work, we need to make sure we are not mixing pears with bananas when comparing hyperplanes. That is, in principle the \( i \)-th coefficient of one model’s hyperplane might be associated to a program behaviour (or, more generally, a problem) while the \( i \)-th coefficient in another model’s hyperplane might be associated to a different program behaviour (problem). In fact, since the sets \( S \) are independently chosen in different models, chances are that this will happen on a regular basis. Of course, one could solve this problem by forcing all models to use the same \( S \), but then almost certainly the accuracy of the models would suffer. To circumvent this problem we used the following procedure to measure angles (and then study similarities and differences) between algorithms.

Let \( S' \) and \( S'' \) be the sets of reference vectors associated with two performance models we want to compare. The models also include two corresponding sets of coefficients, \( a'_{p_1}, \cdots, a'_{p_{|S'|}} \) and \( a''_{p_1}, \cdots, a''_{p_{|S''|}} \), respectively. These may be incomparable since the sets \( S' \) and \( S'' \) may include different elements and can even be of different cardinality. Therefore, we construct two new sets of coefficients as follows. The first set of coefficients, which we will call \( b'_{p_1}, \cdots, b'_{p_{|S'|}} \), is obtained by re-running linear regression on the training set
associated with the first model but using the reference vectors $S''$ (this is why the subscripts range from 1 to $|S''|$). We obtain the second set of coefficients, $b'_p, \ldots, b'_{p|S''|}$, symmetrically. While the sets $a'_p, \ldots, a'_{p|S'|}$ and $a''_p, \ldots, a''_{p|S''|}$ are incomparable, and so are the sets $b'_p, \ldots, b'_{p|S''|}$ and $b''_p, \ldots, b''_{p|S''|}$, we can compare the set $a'_p, \ldots, a'_{p|S'|}$ with the set $b''_p, \ldots, b''_{p|S''|}$ and the set $a''_p, \ldots, a''_{p|S''|}$ with the set $b'_p, \ldots, b'_{p|S''|}$. Thus, we define the vectors $a' = (-1, a'_p, \ldots, a'_{p|S'|})$, $a'' = (-1, a''_p, \ldots, a''_{p|S''|})$, $b' = (-1, b'_p, \ldots, b'_{p|S''|})$ and $b'' = (-1, b''_p, \ldots, b''_{p|S''|})$, and we compute two angles:

$$\alpha_1 = \arccos\left(\frac{a' \cdot b''}{\|a'\| \|b''\|}\right) \quad \text{and} \quad \alpha_2 = \arccos\left(\frac{a'' \cdot b'}{\|a''\| \|b'\|}\right).$$

(7.3)

From these we finally compute the angle (or dissimilarity) between two algorithms as

$$\alpha = \frac{\alpha_1 + \alpha_2}{2}.$$  

(7.4)

### 7.1.2 Using Distance Functions to Measure Similarity

In the previous section, we measured similarity between algorithms using the idea that Equation (4.5) represents a hyper-plane. Here, we use a different approach to measure similarity. We will show that an algorithm can be represented as a vector and then we can measure similarity between algorithms by computing the distance between their associated vectors.

The first step is to re-write Equation (4.5) as the scalar product between two vectors:

$$P(t) \approx (a_0, \ldots, a_{|S'|}) \cdot (1, d(p_1, t), \ldots, d(p_{|S'|}, t)).$$

(7.5)

Clearly, the performance model in (7.5) depends only on a vector $c = (a_0, a_p, \ldots, a_{p|S'|})$. We can therefore calculate the similarity between pairs of systems by computing the distance between their associated $c$ vectors. We can compute the distance between vectors using any of the functions presented in Table 5.5 except the square of the dot product and the Ruzicka similarity.

As for the use of angles to measure similarity, again for this to work, we must make sure corresponding components of the $c$ vectors being compared are associated to the same elements of $S$. Here, we used the same approach of computing two distance measures and then averaging them.

### 7.1.3 Using Similarity Measures

With these procedures in hand, one can ascertain whether two algorithms behave similarly or not performance-wise. If the similarity between them is large, then 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. Thus, one might decide to favour the faster algorithm. If the similarity between two
algorithms is lesser, then we can expect to find 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 of solving it with the other.

In the presence of more than two algorithms, we can also compute the similarity between all pairs of algorithms under consideration and infer useful information on their mutual relationships. The information on the results of pair-wise comparisons can be stored in a matrix. However, when one compares many algorithms a new problem presents itself: the matrix reporting the results of pair-wise comparisons between algorithms is necessarily very large. As a result, finding interesting patterns within the matrix manually may be a very lengthy task. This further increases in complexity because we can have more than one comparison matrix. For example, we would get at least two comparison matrices, one for each of the procedures previously described. These two comparison matrices will complement each other and, therefore, we will need to analyse both in order to gain a richer understanding of the similarities between the algorithms under study.

In order to aid the manual analysis of the comparison matrix, in the next sections we present an automatic produce that generates a taxonomy of program-induction algorithms given a comparison matrix.

7.2 Taxonomies for Program-induction Algorithms

Before introducing the procedure used to create a taxonomy, in this section we want to explain why having a taxonomy is important.

A taxonomy is the coherent arrangement of elements into groups. For many sciences, the construction of a taxonomy has been an important step towards maturity. Taxonomies have many applications. In biology, for example, taxonomies of animals and plants are the starting point in understanding the biodiversity of a region. Also, taxonomies help to model a group of individuals as a single entity, thereby removing the need to analyse each member of the group separately.

The importance of taxonomies is also clear for EPAs. For example, from a practitioner’s point of view, an algorithm taxonomy may reduce the time-consuming task of finding the most suitable algorithm to solve the problem at hand. This is generally done by selecting an algorithm that has already been applied to a “similar” problem. If this algorithm fails to return a satisfactory result, one would want to look for an algorithm that, due to its characteristics, is sufficiently different from the first, e.g., an algorithm that belonged to a different group of the taxonomy. Taxonomies of problems are also useful since it is often possible to associate algorithms to problem classes. Knowing which taxonomic class a problem is in can then guide a practitioner towards suitable algorithms for solving it.

So, what taxonomies are available for EPAs? There are some general taxonomies; in fact, every author of a book on EAs is forced to come up with some structure within which to present the material in an
orderly fashion; for example see [7, 92]. This structure can be seen as a taxonomy of algorithms. Typically these taxonomies are constructed on the basis of a set of characteristics which are deemed to represent such algorithms. These often focus on each of the main components of an algorithm, e.g., the representation, the genetic operators, the number of parents used to produce an offspring, the number of offspring, the way the offspring is included in the population, etc. That is, they take a reductionist approach, which looks at similarities and differences among the components of an algorithm, as perceived by the user. Also, often the topmost levels in the taxonomy are determined by historical reasons.

While these taxonomies are useful, they are, rather naturally, subjective. Furthermore, while some of the most useful taxonomies in other disciplines are based on behaviour; not just form (think, for example, of the periodic table), these types of EA taxonomies are typically only based on “form”, i.e., the components of an algorithm. Thus, they are not designed to give answers to practitioners, who are really interested in determining whether an algorithm will solve their problems.

Of course, there are other EA taxonomies which are much more precise and useful. For example [13] proposed using a tabular representation to describe genetic algorithms, scatter search and ant systems. In [80] a taxonomy of parallel genetic algorithms was proposed, which was based on the different ways in which a genetic algorithm can be parallelised. [93] categorises various EAs and other mathematical and AI search algorithms. In [41] a taxonomy of crossover operators for real-coded genetic algorithms was presented. The taxonomy consisted of 18 different crossover operators grouped into 3 classes. There have also been a very small number of taxonomic efforts of this kind in GP, e.g., [148]. Also, a GP problem taxonomy was proposed in [71].

One feature of these taxonomies is that they capture only specific aspects of an algorithm or they are very specific to a special class of algorithms. The reason is combinatorial: not only are there many different EPAs to consider, but also each may have many parameters (e.g., 17 parameters were used in the GP system described in [59]), which can alter its behaviour, in some cases very significantly. If one considers the combinations of these parameters, then the set of algorithms with their different configurations immediately becomes astronomically large. Another feature of these taxonomies is that, like the ones described above, they are manually built and, as a result, they are subjective and are normally based on form rather than the behaviour of algorithms.

A step in the direction of automating the creation of taxonomies as well as turning them into practical tools has been made by Ashlock et al. in [2, 3] where several tens of problems where automatically grouped into a problem taxonomy. The idea was to run a particular EA where mating is controlled via a neighbourhood structure recording the average number of fitness evaluations required to solve each problem using each of a small number of neighbourhood structures. The resulting performance signatures (vectors) were then clustered hierarchically, thereby producing a taxonomy.

Here, we present a method to create taxonomies which has some similarity with the one proposed by
Ashlock et al., but also with some differences. Firstly, we focus on EPAs. Secondly, we are interested in producing a practical algorithm taxonomy, which we think may be more useful than a problem taxonomy. Similar to Ashlock’s work, our approach is automated and objective. We apply it to the problem of grouping EPAs systems with their associated parameter settings. Reminiscent of [2, 3], we do this on the basis of the performance signatures (vectors in Equations (7.1) and Equation (7.5)) and a hierarchical clustering algorithm. However, our performance signatures represent the behaviour of each system across a class of problems, not the behaviour of multiple versions of an algorithm over one problem. Also, our signatures are not the measurement of performance over multiple cases: they are obtained through a process which involves the instantiation of a mathematical model of EPA’s performance. It is then this model which provides the signature for each algorithm.

7.3 Creating EPA’s Taxonomies

The procedure for automatically creating a taxonomy is based on the idea of treating the pair-wise comparison matrix (see Section 7.1) as the weight matrix of a graph. The nodes of the graph represent the systems under comparison. Thereafter we can use this matrix alongside a clustering algorithm in order to group systems based on the similarity of their performance. Finally, we compute the distance between the resulting clusters and use a graph-drawing package to produce a graph layout where pairs of nodes corresponding to clusters with a small distance (high similarity) are placed closer to each other than clusters corresponding to systems that are dissimilar performance-wise.

To explore this idea, we used a hierarchical clustering algorithm (see [53]) to create the clusters and the neato package, which is part of the GraphViz library and is available in standard Unix distributions, to draw the graph layout.

The hierarchical clustering algorithm works as follows. Let us assume we are given a set of objects (GP, GEP, CGP and SIHC systems, in our case) that we wish to group. Let $M$ be a square matrix where element $(i, j)$ represents the distance (or similarity) between the $i$-th and $j$-th objects. The distance between objects is computed using any of the procedure described in Section 7.1. We derive from $M$ a similarity measure $s(\mathcal{X}, \mathcal{Y})$ between pairs of clusters, $\mathcal{X}$ and $\mathcal{Y}$ (more on this below). Then we apply the clustering algorithm of [53]. This involves the following three steps:

1. First, each object is assigned to a separate cluster.

2. A new cluster is created by merging the two clusters which are closest based on the similarity measure $s(\mathcal{X}, \mathcal{Y})$, thereby reducing the number of clusters by one.

3. Repeat step 2 until there is only one cluster left.

Naturally the behaviour of the clustering algorithm is influenced by the similarity measure $s(\mathcal{X}, \mathcal{Y})$ used.
Here, we used the average linkage as a similarity measure. That is

$$s(X, Y) = \frac{\sum_{i \in X} \sum_{j \in Y} M(i, j)}{|X||Y|}$$

(7.6)

where $|\cdot|$ denotes the number of elements in a cluster.

The result of this clustering algorithm is usually represented using dendrograms such as the one shown in Figure 7.1. While dendrograms are a useful tool, they are not a taxonomy. Firstly, there are far too many classes for a taxonomy (the dendrogram for $n$ objects includes $n - 1$ classes). Secondly, the dendrograms produced in the presence of many objects (such as our 21 EPAs systems) are complex and difficult to digest. For example, it is difficult to appreciate the relative distance between clusters. Therefore, to transform dendrograms into taxonomies we first perform a data reduction exercise. We focus only on the $m$ topmost clusters in the dendrogram starting from its root node and then we use neato to draw these clusters, thereby obtaining the taxonomy.

The key strategy used by neato to position the nodes of a graph is to interpret them as physical bodies connected by springs (which correspond to the edges between nodes 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. Often this relaxation process leads to nicely drawn graphs where groups of nodes with either strong connections or many connections tend to be placed next to each other (since this effectively minimises the associated elastic energy).

We then imagined that each of these $m$ clusters found by the clustering algorithm was a node in a fully connected graph. We associated a weight proportional to $s(X, Y)$ to each edge, in order that the nodes would be pushed apart when producing the graph’s layout proportionally to their dissimilarity. We also set the stiffness of springs to be approximately inversely proportional to similarities using the formula $\frac{1}{0.01 + s(X, Y)}$.

Finally, this information was given to neato to generate a graph layout and, thereby, a taxonomy.

In summary, to produce the taxonomy we first need to compute the comparison matrix $M$ using the procedure described in section 7.1. Then, this matrix $M$ is used in conjunction with the hierarchical clustering algorithm to group the systems in clusters. Finally, these clusters are drawn by neato producing, as a result, the taxonomy.
7.4 Taxonomies

Having presented the procedure to create taxonomies, we are now in a position to create one. Here we present taxonomies composed by the different program-induction algorithms described in Section 5.2.

The first step in creating the taxonomy is to obtain the comparison matrix $M$. As we have mentioned previously, in order to get $M$, we need to create the performance model for each algorithm involved in the taxonomy and then use either the procedure described in Section 7.1.1 or in Section 7.1.2. To initialise a performance model, we need a class of problems and a performance measure. In Section 5.1.1, we described two classes of problems for program-inductions algorithms, namely continuous symbolic regression and Boolean function induction problems. There we used two performance measures for the continuous symbolic regression problems and one for the Boolean induction problems. Thus for every system in the taxonomy we have 3 different performance models: two that correspond to the rational problems and one for the Boolean induction problems.

In Section 7.1 we described two procedures which can be used to compute comparison matrices. The first gives us one comparison matrix created using angles (see section 7.1.1). The second gives us six more comparison matrices obtained using the procedure in Section 7.1.2 (each of these six corresponds to a different distance measure of Table 5.5). The result is that there are seven comparison matrices; one available for each of the different performance measures. Therefore, in total we have 14 different comparison matrices $M$ created using the performance model initialised in the rational problems and 7 for the Boolean induction problems.

Each of these comparison matrices is looking, from a different perspective, at the similarities or differences in the behaviour of a program-induction algorithm when solving a particular class of problems.

At this point, we have a few options for creating the taxonomy. One is to create one taxonomy for each of the comparison matrices mentioned above and then try to draw some conclusions from all these taxonomies. One of the drawbacks of this option is the amount of space required to present these 21 taxonomies. Furthermore, we would like to have only one taxonomy for every class of problems instead of having several taxonomies for each class. We therefore decided to follow a different approach. That is, we condense the information of the multiple $M$’s associated with a class of problems into one comparison matrix and then use this new comparison matrix to obtain the taxonomy of the algorithms for that class of problems.

In order to create the final comparison matrix corresponding to the rational problems, we decided to average corresponding entries in its 14 matrices ending up with a $21 \times 21$ matrix $M$. For this to work well, we need to make sure that all the distances use the same scale. If not, the result will be heavily influenced by those distances that produce the highest values. Therefore, before averaging these matrices, we decided to scale the distances in the interval 0 to 1 and then averaged the scaled version of the matrices to obtain the final $M$. An equivalent procedure was followed to obtain the average matrix in the case of Boolean
problems.

With these average $M$’s we used a hierarchical clustering algorithm. This clustering algorithm forms a dendogram of 21 levels and we decided to focus on the 8 topmost clusters in the dendogram. We then imagined that each of these 8 clusters was a node in a fully connected graph. Finally, we use neato to draw the graph layout and the taxonomy.

Figure 7.2 shows the taxonomy produced by neato for program-induction algorithms on the rational problems. 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. To distinguish different forms of selection and reproduction we use different graphical symbols to represent the corresponding nodes. Namely, the steady state systems are represented with ellipses, while the generational systems with roulette-wheel selection and tournament selection are represented using octagons and double octagons, respectively. The SIHC systems with subtree mutation are represented using hexagons and the SIHC with uniform mutation is represented using a pentagon. The CGP system is represented with a septagon.

From the figure we can see how the SIHC systems with sub-tree mutation are grouped in a cluster (bottom left of the figure), the generation GP systems with tournament selection are grouped together in one cluster (top left), while the generational GP systems with roulette-wheel selection are arranged in another cluster (top right). In the top of the figure we find another cluster containing the steady state GP systems and the
SIHC with uniform mutation indicating that these systems behave similar. Somehow surprisingly, each of the GEP systems was placed in a separate cluster (middle and bottom right of the figure) indicating that these systems are very different performance-wise. Finally, we find the CGP system at the middle of the figure.

A conclusion that we can draw from the taxonomy in Figure 7.2 is that the type of selection and type of evolution used in an algorithm have a bigger impact on performance than crossover and mutation rates. This is evident from the clusters formed by the steady state systems, generational systems with tournament selection and the generational systems with roulette-wheel selection. In relation to the SIHC, the taxonomy indicates that the type of mutation is more important than the maximum number of mutations. Surprisingly, the taxonomy also suggests that the SIHC with uniform mutation is very similar to the steady state systems. This is something one could have never guessed simply by looking at structural similarities between these two classes of algorithms.

Figure 7.3 presents the taxonomy that corresponds to the Boolean induction problems. In the bottom right of the figure, we can see all the SIHC with sub-tree mutation grouped in one cluster. The steady state GP systems are arranged in another cluster (middle left). The generational GP systems with roulette-wheel selection are group in a cluster (left of the figure). On the top of the figure, we observe the cluster composed by the generational GP systems with tournament selection. As can be seen in the figure, all the GP systems are arranged in clusters according to the type of evolution and selection mechanism. These clusters indicate that the type of evolution and selection mechanism are more relevant performance wise than crossover rate and mutation rate. The generational GEP systems are grouped in one cluster (top left of the figure). The rest of the systems, i.e, the steady state GEP system, SIHC with uniform mutation and the CGP system, are grouped in their own clusters (top middle, bottom middle and middle of the figure).

Figures 7.2 and 7.3 present some similarities and some differences. Some of the similarities correspond to the clusters formed by SIHC with sub-tree mutation, the generation GP systems with roulette-wheel selection, the GEP systems and CGP system. The position of the CGP system caught our attention because in both figures this system is placed in the middle of the figure. This is an indication that this system behaves different with respect to the other systems. This difference in behaviour might be the consequence of being the only system that does not use trees to evaluate the individuals, as mentioned previously it is a graph based system.

The only substantial difference is that whilst in Figure 7.2 the steady state systems and the GP systems with tournament selection are grouped in different clusters, these systems are grouped in the same cluster in Figure 7.3. This indicates that for the Boolean problems these systems behave more similarly performance-wise than they do for the case of rational problems.

The message we can take from these two taxonomies is that the type of evolution or the selection mechanism are more important than the crossover or mutation rate. For the case of SIHC the type of
mutation is more important than the maximum number of mutations.

7.5 Taxonomies for EPAs with Different Population Sizes

In the previous section, we have experimentally shown that our method produces meaningful taxonomies. These taxonomies have been used to analyse the relation, performance wise, that exists between different program-induction algorithms. Given the success of this approach, we decided to go further and create richer taxonomies. Instead of including different types of program-induction algorithms, we decided to study how the performance of the program-induction algorithms used in the previous section varied when the size of the population and the number of generations are varied.

The program-induction algorithms composing the taxonomies showed in this section are the 21 algorithms described in Section 5.2 with the exception of the systems with a 90% crossover rate, plus 36 new algorithms created by varying the size of the population of many of these algorithms. Firstly, we used 4 different sizes of population, i.e., $P = 10$, $P = 50$, $P = 250$, and $P = 5000$. The number of generations was also modified to keep the number of individuals evaluated per run constant. That is, $P \times G = 50000$ where $G$ is the number of generations and $P$ is the population size. We varied the population size for all the GP systems, i.e., the generational GP systems with roulette-wheel selection and tournament selection and the steady state GP systems. This results in 36 new settings, 9 for each different population size. Below we will build taxonomies composed by 54 different program-induction algorithms.
The procedure used to create the taxonomies is equivalent to the one described in the previous section. The only difference is that here we use the 10 topmost clusters found by the hierarchical clustering algorithm, instead of the 8 used previously. Furthermore, given that many of the clusters contain all the GP systems with a particular population size, we chose to represent these systems using only one node labelled with the size of the population. For example, a cluster that contains all the steady state GP systems with a population size of 50 will be represented by a ellipse with the label “$P = 50$”. We decided to make this further simplification in order to increase the readability of the taxonomies produced.

Figure 7.4 shows the taxonomy that corresponds to the rational problems. In the top of the figure, we can see a cluster containing the generational GP systems with a population of 1000, the steady state GP systems with $P = 5000$ and the CGP system. Below this, we see two clusters one composed by only one system: the generation GP with tournament selection, no mutation and $P = 5000$. The other group the steady state GP systems with $P = 250$ and $P = 1000$, the steady state GEP system, the SIHC with uniform mutation, the generation GP systems with $P = 250$, no crossover and 50% crossover rate. In the left of the figure, there is a cluster containing the SIHC with sub-tree mutation and several GP systems with tournament selection, small population ($< 250$) and high mutation rate. Most of the generational GP systems with roulette-wheel selection among other generational systems are group in another cluster on the left of the figure. In the bottom of the figure, we find 3 cluster composed with GP systems with no mutation and small population sizes plus all the generational GP systems with roulette wheel selection and $P = 10$. The middle of the figure contains two clusters composed by only one member. That is the steady state GP system with no mutation and $P = 50$ and the generational GP system with tournament selection, 50% crossover rate and $P = 10$.

From the taxonomy presented in Figure 7.4 one might infer some of the influences that the different parameters have on the behaviour of the algorithms. On one hand, we can see that in relation to performance the crossover and mutation rates are only important when the algorithms have a small population (let say, 10 or 50 individuals) as indicated by the clusters formed by the systems with no mutation (bottom of the figure) and the cluster composed by the systems with high mutation rates (middle right of the figure). Furthermore, these clusters grouped the systems according to the rates of crossover. That is, when the population is small we see that the algorithms with no mutation are grouped together and the algorithms with 50% mutation or no crossover are in another group. In fact, we see that when the population is small the algorithm with no mutation and the algorithms with mutation do not live in the same cluster.

On the other hand, when the population is bigger than or equal to 250 individuals we see that crossover and mutation rates are not discriminant factors in the creation of groups. In fact, all the different systems with these population sizes are grouped in the same cluster with the only exception of the generational GP systems with tournament selection and $P = 250$. For example, all the generational systems with roulette-wheel selection and a particular population size are grouped in a cluster (top left). Furthermore, with these sizes of population, the clusters in the taxonomy show that for some types of systems a change in the size
of the population does not produce any effect on the behaviour of the algorithm. See, for example, the cluster formed by the steady state GP systems with $P = 1000$ and $P = 250$ (top right) and another cluster composed by the generational GP with roulette-wheel selection and $P = 250$, $P = 1000$ and $P = 5000$ (top left). In addition, we see that a modification in the population size produces the result that different systems behave similarly as it is the case with the steady state GP systems and the generation GP with tournament selection and $P = 5000$ and $P = 1000$, respectively (see cluster in the top). Previously, we saw the same result for the Boolean induction problems the steady state GP systems and the generational GP systems with tournament selection behave similarly (see Section 7.3).

Figure 7.5 shows the taxonomy for the case of Boolean induction problems. At the top of the figure we see a cluster composed by many of the steady state systems (i.e., the steady state GP systems with $P = 50$, $250$ and $1000$, the steady state with $P = 10$ and no crossover plus the steady systems with 50% mutation rate, besides of the steady state GEP system), the generational GP systems with tournament selection $P = 50$, $P = 250$ and $P = 1000$, the generational GP systems with tournament selection, $p_m \leq 50\%$ and $P = 10$, the CGP, and the generational GP systems with roulette-wheel selection, $p_m \leq 50\%$, $P = 10$ and $P = 50$, all the generational GP systems with roulette-wheel selection and $P = 250$. Both generational GEP systems are grouped in one cluster (bottom right). The generational GP systems with $P = 5000$ are
Figure 7.5: Taxonomy of different program-induction algorithms with different population size and created initialising the performance models on the Boolean induction problems.

set in a cluster (bottom left), while the steady state GP systems with $P = 5000$ are located in the bottom of the figure. In the top left of the figure, we see all the SIHC systems with sub-tree mutation group in one cluster. Above it, we find the SIHC with uniform mutation.

Here we can draw conclusions similar to the ones drawn for the previous taxonomy (Figure 7.4). That is, here too we find that the crossover and mutation rates change the behaviour of algorithms only when the size of the population is small, as indicated by the different clusters in the middle and top of Figure 7.5. However, in this taxonomy we do not have the additional characteristic that the systems with small population are grouped according to whether they are crossover or mutation based. Previously, we saw that for some population sizes (equal to or above 1000 individuals) the generational GP systems with tournament selection are similar to the steady state GP systems. In Boolean problems, this similarity goes even further, including more population sizes. That is, in the top of the figures, we see a cluster that contains the steady state GP systems with $P = 50, 250, 500$ and 1000 and the generational GP systems with tournament selection and $P = 50, P = 250$ and 1000. Somewhat surprisingly this similarity between the GP systems with tournament selection disappears when the population reaches 5000 individuals as shown in the clusters at the bottom left of the figure (generational GP systems with $P = 5000$) and in the bottom of the figure (cluster with the steady state GP systems with $P = 5000$).

A further useful lesson we can learn from our graphic representation of GP, GEP, CGP and SIHC
performance is the following. Given that the systems were grouped based on our performance models, it is reasonable to expect that if a particular system fails to solve a problem in repeated attempts, it will be more efficient to try one or more alternative systems from a different cluster, rather than try to finely optimise the parameters of the first system. Parameter optimisation can be used later to further improve performance once a satisfactory system has been found. In other words, if we are not satisfied with the performance of a system in a particular cluster, we should try a different cluster before we explore the first cluster any further.

There may be a lesson for algorithm portfolios here. Perhaps when forming portfolios one should not just pick the $n$ (say, 3) algorithms with best performance: one should also look at how orthogonal or independent the performance of such algorithms is. In other words, to obtain better generalisation, we may need to look at how well the algorithms in the portfolio cover the performance space, e.g., using our algorithm taxonomy based on performance.

7.6 What Knowledge can we Extract from Measuring Performance Empirically?

Before we conclude this chapter, we would like to compare and contrast what we have learnt from analysing our performance models against 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.

Table 7.1 presents the typical results one might obtain from a traditional approach. The table reports the performance of each of the GP, GEP, CGP and SIHC systems and parameter settings we looked at in Section 7.4 on the rational-function and the Boolean-function testbeds. The statistics were collected by running each system on the 1,100 different problems obtained by joining the training set $T$ and the validation set $V$ for each problem class. For each problem instance we performed 100 independent runs of the system to estimate its performance. So, the results in the table are effectively a summary of 4,620,000 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 look at the BRF or the NBRF performance measures. CGP has the best performance on Boolean induction problems.

The worst performance on the rational symbolic regression problems was obtained by the generational GP system with 90% crossover rate. In the Boolean induction problems the worst performance corresponds to the SIHC with sub-tree mutation and a maximum of 50 mutations between restarts.

Looking further into the table one might be tempted to conclude that for the case of the generational GP system with roulette-wheel selection 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, the differences in
performance observed when crossover and mutation rates are varied are not statistically significant, and
this is generally true of other systems. On the other hand, the performance differences observed when the
selection mechanism changes from roulette-wheel to tournament are statistically significant. Therefore, the
empirical data also suggests that for these GP systems and for these sets of problems, changing the selection
mechanism has a greater effect than varying the crossover and mutation rates.

However, there are a variety of other phenomena that we were able to capture with an analysis of our
models that an analysis based on simple statistics cannot reveal. This is because, it is entirely possible that
two systems achieve exactly the same average performance but one does so by solving a set of problems, A,
very well and another set, B, not so well, while the other solves the problems in B very well and those in
A not so well. Means and standard deviations of performance cannot tell such systems apart. An analysis
based on comparing our models can.

This explains why, as we have seen earlier, while Table 7.1 suggests that there might be cases where
varying the rates of application of the search operators would influence performance, as a matter of fact such
a suggestion does not withstand statistical testing. Simply put, empirical results such as those reported in
Table 7.1 do not have the resolution to provide a clear picture of what is going on.

For the same reasons Table 7.1 is unable to reveal that, in fact, generational systems with tournament
selection are more similar to steady state systems with tournament selection than to generational systems
with roulette-wheel selection. Likewise, the table cannot reveal that SIHC with uniform mutation is similar
to generational GP systems with tournament selection and very different from the SIHC with sub-tree
mutation.

For these reasons we think that not only can our models of performance provide quick and accurate
predictions of how well an algorithm will do on an unseen instance of a problem, which can reliably be
used for a variety of practical applications, including portfolios, but they can also be the building blocks
for sophisticated tools for the analysis and comparison of algorithms and parameters, which can effectively
complement traditional approaches.

### 7.7 Summary

This chapter presented an automatic procedure for the creation of taxonomies for different program-induction
algorithms. The procedure is based on the performance models of the different algorithms involved in the
taxonomy. We used the models’ coefficients as signatures which were then clustered via a hierarchical
algorithm to create dendrograms. The dendrograms were further processed to identify the key groups and
their relationships. These were formalised into a weighted graph, which really represents our taxonomies.
Plotting the graph produced a highly informative representation of taxonomies from which a variety of
lessons could be learnt.
We created two sets of taxonomies, one for each of the class of problems described in previous chapter. That is, the first set of taxonomies corresponds to the rational symbolic regression problems and the other taxonomies are for the Boolean induction problems. Each set has two taxonomies, one with 21 different program-induction algorithms and the other with 54 different program-induction algorithms.

A general conclusion that we can draw from the different taxonomies presented is that the crossover and mutation rates are not significant with respect to performance, unless the population size is small. In particular, we can see that for the rational problems it is important to consider whether the systems are crossover based or mutation based. For the case of Boolean induction problems the taxonomies showed that the steady state systems and the generational GP systems with tournament selection behave similarly.

In the last part of the chapter (Section 7.6), we presented the results of the traditional procedure commonly followed in the community to highlight the differences and similarities between evolutionary algorithms. From this analysis, we showed that our procedure not only complements the traditional analysis, but also provides more information than the traditional approach.
Chapter 8

Performance Models using Hardness Measures

As mentioned earlier, one of the objectives of this thesis is to model the performance of EPAs. In this chapter, we pursue this objective following a different approach that one taken in Chapter 4. Here, we create a model for the performance of EPAs based on variables that are related to the hardness of problems. As can be seen, this contrast with the model presented previously where the performance was model using a similarity between a set of reference problems and the problem of interest.

More specifically, in this chapter we used two measures that are problem specific and related to the hardness of problems. The first hardness measure is for Boolean functions and the second is for continuous functions (e.g., rational functions). With these two measures, we create two models, which we will call hardness models, for predicting the performance of EPAs.

The rest of the chapter is organized as follows. In Section 8.1, we describe a measure that is related to the hardness of the Boolean function. Starting from this measure, we develop a measure for continuous functions in Section 8.2. Section 8.3 describes the procedure used to create models of performance, based on these hardness measures. In Section 8.4, we describe the procedure used to identify and optimise the models. Section 8.5 presents the test problems, the algorithms under study and the way the parameters of the model are obtained. A study of the quality of the models and a comparison between the modelling techniques presented here and the performance model (see Chapter 4) are presented in Section 8.6. Finally, Section 8.7 presents how the hardness models can be used to tackle the algorithm selection problem.

8.1 Hardness Measure for Boolean Functions

In [32] a hardness measure for Boolean functions was introduced. From the point of view of this thesis, the principal characteristic of this measure is its correlation with the time (number of epochs) needed by a
learning algorithm to train a feed-forward neural network. That is, a learning algorithm will require more time to train a neural network with a Boolean function with a high value than to train it with a function with a lower value.

This hardness measure is based on counting the number of neighbouring examples in the training set with a different output value. Firstly, let us describe what a neighbour is. To some degree, all pairs of input patterns are neighbours; however, this measure only takes into consideration the outputs that correspond to inputs that are at a specific distance. In [32], the Hamming distance was used to measure the distance between pairs of inputs. The hardness measure works by counting the number of pairs of outputs that are different when the corresponding inputs are at 1 or 2 Hamming distance. Of course, one could count the differences in the outputs for larger Hamming distances. However, in [32] it was decided that it is enough to compute only the differences for the inputs having a Hamming distance of one or two. In formulae, this hardness measure is defined as:

\[ \vartheta(f) = \tilde{\vartheta}_1(f) + \tilde{\vartheta}_2(f), \]  

where \( \tilde{\vartheta}_i(f) \) counts the number of pair of outputs that are different when the Hamming distance of their inputs is \( i \). That is,

\[ \tilde{\vartheta}_i(f) = \frac{1}{2^N \times \binom{N}{i}} \sum_{j \in I} \sum_{\{k \in I: H(j,k)=i\}} |f(j) - f(k)|, \]  

where \( f(j) \) is the output of Boolean function \( f \) for input \( j \), \( N \) is the number of variables, \( I \) contains all the possible input patterns (e.g., for 3 variables \( I \) contains 8 different input patterns), \( H(j,k) \) computes the Hamming distance between input \( j \) and \( k \). Term \( \frac{1}{2^N \times \binom{N}{i}} \) is a normalization factor to ensure the value of \( \vartheta_i(f) \) is in the interval [0, 1]. The second summation is for all inputs \( k \) that are at a Hamming distance \( i \) from input \( j \).

It might be observed from Equation (8.1) that the building blocks of this equation are the terms \( \tilde{\vartheta}_i(f) \). Therefore, the first thing to do in order to apply an equivalent idea to other types of function is to see if it is feasible to use \( \vartheta_i \)'s in other types of function. In the next section, we will study these terms and propose a modification to them in order to address some of the problem encountered.

8.2 Hardness Measure for Continuous Functions

Before describing our modification to Equation (8.2), let us start by providing examples of the neighbourhood structured induced by a Hamming distance. Figure 8.1 shows a graphical representation of all the input patterns for the 2 variables. The inner and outer arc\(^1\) represent the Hamming distance of one and two, respectively. For example, input (0, 0) and input (1, 1) have a Hamming distance of 2 thereby input (1, 1) is

\(^1\)It may be evident that the Hamming distance does not induce an arc, instead a graphical representation of it is a line. However, for the sake of explanation let us assume that this distance can be represented using arcs.
Figure 8.1: Schematic representation of all the input patterns for two Boolean variables and the distance between input pattern \((0,0)\) and all other input patterns on the outer arc.

From the figure, one can see that Equation (8.2) might have an awkward behaviour on continuous functions due to the fact that the equation requires all pairs of inputs that are at a specific distance to be summed. Moreover, the equation uses the Hamming distance, which is not very suitable for continuous spaces. Regarding the distance, let us assume that Hamming is replaced with a distance for continuous variables (e.g., sum of the square differences). Even with this modification, Equation (8.2) is still hard to apply. For instance, in the worst case we would have as many values of \(i\), and therefore as many values of \(\tilde{\vartheta}_i(f)\) as the possible pairs of inputs. This is the case when all the pairs of inputs have a different distance value.

Let us address these two problems. Firstly, we address the problem induced by the strategy used to decide whether the outputs of two inputs patterns should be compared. The proposed strategy is to relax the equality needed by the previous equation, and, instead, compare the outputs of any pair of inputs when their distance is within an interval. Specifically, let \(d\) be a distance function, and \(c_1\) and \(c_2\) two constants where \(c_1 < c_2\). Then, for any pair of inputs \(j, k\) we compare their outputs if \(d(i,k)\) is less or equal to \(c_2\) and greater than \(c_1\), i.e., \(c_1 < d(i,k) \leq c_2\). Secondly, as we mentioned previously, the Hamming distance can easily be replaced by another distance that is appropriate for the space of the inputs. For example, if the inputs are real numbers then a possible distance function could be the sum of the absolute errors, just to mention one. With these modification Equation (8.2) transform into

\[
\tilde{\vartheta}_i(f) = \sum_{j \in I} \frac{1}{\vert C_i(j) \vert} \sum_{k \in C_i(j)} \vert (f(j) - f(k)) \vert, \tag{8.3}
\]

where \(I\) contains all the inputs and \(C_i(j)\) is defined as:

\[
C_i(j) = \{k \in I : c_{i-1} < d(j,k) \leq c_i\}. \tag{8.4}
\]
Figure 8.2: Graphical representation of the sets $C_i(j_1)$.

That is, $C_i(j)$ contains the inputs values that are in the interval defined by input $j$ and coefficients $c_i$ and $c_{i-1}$. The value of coefficients $c_i$ and their number are set by the user; the constraints are that $c_0 = 0$ and $c_{i-1} < c_i \forall i$.

Comparing Equations (8.2) and (8.3) one can see that Equation (8.2) is a particular case of Equation (8.3). That is, Equation (8.3) transforms into Equation (8.2) by setting $c_i = i$ and $d = H$.

Figure 8.2 shows a graphical representation of how set $C_i(j)$ is formed for the case of two continuous variables. The figure shows 11 hypothetical inputs and four circles that represent the different values of coefficients $c_i$. The circles are centred at input $j_1$ indicating that we are looking for the inputs $C_i(j_1)$. When $i = 1$ there are no inputs in $C_i(j_1)$, beside $j_1$, thereby $\vartheta_i(f) = 0$. $C_2(j_1)$ contains only input, $j_3$, because it is the only input that is contained within the second innermost circle and outside of the innermost circle. This process is repeated for all the coefficients $c_i$. We would like to draw the reader’s attention to the inputs $j_5$, $j_7$ and $j_8$. These inputs do not in appear in any of the $C_i(i_1)$ sets; however, given the first summation in Equation (8.3) is for all inputs, these inputs will participate in the calculation of some of the terms of $\vartheta_i(f)$.

In principle, Equation (8.3) can be used to compute $\vartheta_i(f)$ for at least two different types of functions, namely continuous functions and Boolean functions. As a result, in the next section we will use this equation to create a model for the performance of EPAs solving symbolic regression of rational functions problems and Boolean induction problems.
8.3 Modelling the Performance of EPAs using Hardness Measures

In the previous section, we extended the fundamental elements for the hardness measure proposed in [32] with the objective of applying it to continuous functions. With this extension, we are in a position to use Equation (8.1) to create a model for the performance of EPAs on both of the benchmarks described in Chapter 5. However, instead of using Equation (8.1) to create the model, we decided to further extend its generality first.

Based on our previous experience in creating models of performance, we decided to use a linear equation to create the model. It is worth mentioning that most of the work related to the algorithm selection problem (see Section 6.2) also uses linear equations to predict the performance.

So we will combine the terms \( \vartheta_i(f) \) using the following equation

\[
P(f) \approx a_0 + \sum_i a_i \cdot \vartheta_i(f),
\]

where \( a_i \) are coefficients that need to be identified (more about this later) and \( i \) ranges over all the sets of inputs \( C_i \). For example, for the case of the Boolean function of 4 inputs and using the Hamming distance then \( i \) ranges from 1 to 4.

8.4 Model Identification

We are now in the position to instantiate Equation (8.5) to start making predictions. The equation requires the following set of ingredients to be instantiated: a training set \( T \), a validation set \( V \), a distance function \( d \) and the sets of inputs \( C_i \).

The training set \( T \) is used to identify the coefficients \( a_i \) and the validation set \( V \) is used to test the generality of the model. \( T \) and \( V \) are composed by pairs \((P(f), f)\) where \( f \) is a problem and \( P(f) \) is the actual performance of the algorithm under study on the problem \( f \). Please note that here, we use the procedure described in Section 5.1.1 to build the training and validation set. \( d \) is a distance function that is suitable for the type of the inputs. For the case of symbolic regression of rational function, \( d \) is defined as: \( \sum_{n=1}^{\ell} |j_n - i_n| \), where \( \ell \) is the number of variables. For Boolean induction problems \( d \) is the Hamming distance. Finally, we need to compute the different sets of inputs \( C_i \) (more about this in the next section).

Once the ingredients have been identified, to compute the \( a_i \) coefficients we use a procedure equivalent to the one used in Section 4.5. That is, given the sets \( C_i \), one can compute \( \vartheta_i(f) \) for each \( f \in T \). With this information, Equation (4.6), i.e., \( \mathbf{W}a = \mathbf{p} \), where \( \mathbf{p} \) contains the measure of performance for the problems in \( T \), is used to identify the coefficients \( a_i \). Thereafter, the only thing missing is the matrix \( \mathbf{W} \). This matrix
is constructed as:

\[ W = \begin{pmatrix}
1 & \vartheta_1(f_1) & \ldots & \vartheta_n(f_1) \\
\vdots & \vdots & \ddots & \vdots \\
1 & \vartheta_1(f_m) & \ldots & \vartheta_n(f_m)
\end{pmatrix}. \]

### 8.4.1 Model Optimisation

The process of identifying the inputs \( C_i \) requires the setting of the values of the coefficients \( c_i \). However, this process does not take into consideration the importance of the input sets \( C_i \) and their associated components \( \vartheta(f) \) for the quality of the model. For instance, it might be the case that some \( C_i \) do not provide more information to the model and therefore can be removed without sacrificing the quality of the model. Or it might be the case that some \( C_i \) mislead the predictions, and thus, by removing them better predictions are obtained.

In order to know which of the \( C_i \) inputs will be include in the model, we follow a procedure similar to the one used to determine the elements of set \( S \) (see Section 4.6). That is, we use the LARS algorithm to select the sets of inputs \( C_i \). As previously established, LARS stops after \( m \) steps and returns the \( m \) best sets of inputs \( C_i \). Again, to free the user from setting the parameter \( m \), we use a cross-validation technique to identified \( m \) (see Section 4.6).

Once we have obtained the \( C_i \) that produces the model with best quality in the training set, we can solve the linear systems \( W a = p \), which was introduced in the previous section, to identified the coefficients \( a_i \).

### 8.5 Test Problems, Parameter Settings and Model Initialisation

In order to test the modelling technique presented in this chapter, we will use the classes of problems described in Section 5.1.1: namely, symbolic regression of rational functions and Boolean induction problems. Furthermore, the training set \( T \) and the validation set \( V \) are the ones used to train our previous modelling technique (see Chapter 4). That is, the training set \( T \) is composed of 500 different rational functions or Boolean functions (depending on the domain) and the validation set \( V \) is composed of 600 different functions.

The EPAs that are used to test this technique are the ones described in Section 5.2. That is, we model 21 different algorithms. These algorithms include three variants of GP, three versions of GEP, one CGP, and one SIHC.

The only parts that are missing before one can instantiate the model are the input sets \( C_i \). In order to compute them, we need to decide how many coefficients \( c_i \)'s are necessary and the values of them. In Chapter 5 each rational function was sampled at 21 points uniformly distributed in the interval \([-1, 1]\). This implies that the distance between any pair of consecutive inputs is 0.1. Thus, based on this we decided that \( c_o = 0 \) and \( c_i = c_{i-1} + 0.1 \) for \( i = 1 \ldots 20 \). \( i \) has a maximum value of 20 because no two points can be further away that 2.0.
For the case of Boolean induction problems, we follow the procedure described in [32]. That is, we have as many different $c$ as the number of variables and the minimum distance is 1. Specifically, $c$ has the following values: 1, 2, 3, and 4.

8.6 Results

Let us now experimentally study how good the predictions made by the hardness models are. To illustrate the quality of the models, Table 8.1 shows the RSE values for training set $T$, validation set $V$ and the number of set of inputs $C_i$.

The lowest RSE values in Table 8.1 correspond to the generational GP systems with roulette-wheel selection using BRF as a performance measure. From the table, it might be observed that almost all the RSE values are well below 1, indicating that the models are predicting better than the mean. The only exceptions are the RSE values for the validation set for two SIHC with sub-tree mutation with 500 and 25000 as the maximum number of mutations. However, a further analysis of the predictions made by these two models revealed that these high RSE values are due to the fact that there are a few functions (i.e., symbolic regression problems) that are particularly difficult for these two systems. That is, they performed very poorly. Furthermore, this behaviour was not present in the training set. As a result, for these problems our models produced quantitatively bad predictions. Although the predictions made highly differ from the actual performance, these predictions indicated that the algorithms would perform poorly. Thus, from a qualitative point of view in both cases the same performance of the algorithms was correctly predicted.

Besides the RSE values for the training and validation sets, the table also presents the number of input sets $|C|$ chosen via cross-validation. For the Boolean induction problems, in all the different algorithms, the cross-validation technique used to decide which sets of inputs $C_i$ to include in the model decided that the best model is the one that has all the available $C_i$, i.e., $C_1, C_2, C_3, C_4$. For the symbolic regression problems of rational function, instead, one can see that the value of $|C|$ varies from 2 to 19. Surprisingly, for the case of symbolic regression of rational functions, for some systems this modelling technique is able to produce models with good accuracy using only 2 or 3 coefficients (plus $a_0$). It is also somewhat surprising that for the Boolean induction problems, this technique only requires 4 coefficients (plus $a_0$) to produce a model.

In the next section, we will make quantitative and qualitative comparisons between the performance models and hardness models.

8.6.1 Comparison between Performance Models and Hardness Models

Table 8.1 shows the quality of the hardness models using the RSE values. Previously, we presented the RSE values of the performance model (see Table 5.9). Using these two tables, we can make a quantitative comparison of the quality of both modelling techniques. Comparing the RSE value of each corresponding
system shows which of the modelling technique produced the model with better quality, i.e., lower RSE value.

In order to make a fair comparison of the quality of the models, we used the RSE values of the validation set. We might observe from the tables that, for the case of symbolic regression of rational functions when using the BRF as a performance measure, the hardness model was the better model in 10 systems i.e. it marginally lost to the performance models. However, for the same class of problems, but instead using the NBRF as performance, the hardness model got the lower (better) RSE value in 19 out of the 21 systems tested. The only two systems where the hardness model was beaten by the performance model were the GEP systems with tournament selection. For the Boolean induction problems, the hardness model outperforms the performance model in 18 out of 21 systems. The systems where the hardness model did not obtain the lowest RSE value are the GEP systems.

These tables also show the complexity of the models. By complexity, we mean the number of $a_i$ coefficients that need to be identified. The number of coefficients depends on the number of elements in set $S$ for the case of the performance models and in the number of input sets $C_i$ for the hardness model. Comparing the columns $|S|$ and $|C|$, one can observe that the hardness model requires fewer coefficients than the performance model. For example, the steady state GEP system modelled with the performance model on the Boolean induction problems requires the identification of 399 coefficients. On the other hand, for the same system

| Table 8.1: Quality of the hardness model (RSE) for different EPAs and parameter settings |
|-----------------------------------------------|-------------------|---------------------|------------------|
| **Configuration** | **Rational Functions** | **Boolean Functions** |
| **Type Selection** | **Best of Run Fitness** | **Normalized BRF** | **Success rate** |
| **Generational** | **|** | **|** | **|** |
| **Pc, Pm** | **|** | **|** | **|** |
| 1.00 | 0.00 | 6 | 0.0244 | 0.0525 | 12 | 0.4252 | 0.4511 | 4 | 0.2473 | 0.2306 |
| 0.90 | 0.00 | 7 | 0.0213 | 0.0436 | 11 | 0.4273 | 0.4552 | 4 | 0.2449 | 0.2289 |
| 0.50 | 0.50 | 9 | 0.0358 | 0.1405 | 12 | 0.3550 | 0.4074 | 4 | 0.2602 | 0.2493 |
| 0.00 | 1.00 | 9 | 0.0116 | 0.0149 | 11 | 0.4225 | 0.4926 | 4 | 0.4971 | 0.5265 |
| **Steady State** | **|** | **|** | **|** |
| **Tournament** | **|** | **|** | **|** |
| 1.00 | 0.00 | 14 | 0.0678 | 0.3100 | 18 | 0.2965 | 0.3506 | 4 | 0.3145 | 0.3166 |
| 0.90 | 0.00 | 3 | 0.1542 | 0.2608 | 15 | 0.2949 | 0.3525 | 4 | 0.3492 | 0.3182 |
| 0.50 | 0.50 | 7 | 0.1178 | 0.3003 | 16 | 0.2845 | 0.3448 | 4 | 0.3359 | 0.3267 |
| 0.00 | 1.00 | 15 | 0.0790 | 0.2304 | 16 | 0.2941 | 0.3545 | 4 | 0.3935 | 0.3643 |
| **SHIC** | **|** | **|** | **|** |
| **Sys. Mut.** | **Max. Mut.** | **|** | **|** | **|** |
| **Sub-Tree** | **|** | **|** | **|** |
| 50 | 10 | 0.0811 | 0.1363 | 15 | 0.2355 | 0.2859 | 4 | 0.3238 | 0.3226 |
| 500 | 18 | 0.0610 | 1.4368 | 17 | 0.2316 | 0.2885 | 4 | 0.3127 | 0.3025 |
| 1000 | 19 | 0.0593 | 0.5212 | 14 | 0.2354 | 0.2676 | 4 | 0.3066 | 0.2855 |
| 25000 | 19 | 0.0532 | 1.5914 | 17 | 0.2247 | 0.2935 | 4 | 0.2708 | 0.2416 |
| **Uni.** | **|** | **|** | **|** |
| 25000 | 18 | 0.1083 | 0.3622 | 11 | 0.3205 | 0.4002 | 4 | 0.2987 | 0.2792 |
| **CGP** | **|** | **|** | **|** |
| 4 | 0.2507 | 0.2992 | 18 | 0.3398 | 0.4348 | 4 | 0.7693 | 0.7400 |
and class of problems, the hardness model requires only 4 coefficients. However, it is important to mention that for this case the performance model has a lower RSE value in the validation set and that the difference between them is quite large (0.2559). Therefore, in order to decide which of the models to use, we need to consider the quality of the predictions as well as the complexity of the model.

Although the hardness models obtained lower RSE values in the majority of systems tested, we cannot discard the performance model as a modelling technique for EPAs. One of the reasons is that there are some configurations of systems and problems where the hardness models did not produce the model with the lowest RSE value. Thus, for these cases one should create the model using the performance models. Also, and more importantly, the performance model did not require an expert to come up with a set of hardness measures: all was done automatically.

### 8.7 Hardness Models and Algorithm Selection Problem

In Chapter 6, we applied the performance models to tackle the algorithm selection problem. There, we experimentally showed that the performance models can be used to choose from a program-induction portfolio which algorithm to run to solve the problem at hand. In this section, we will replace the performance model with the hardness model to test if this technique is also suitable to solve the algorithm selection problem.

Surprisingly, replacing the performance models with the hardness models in the symbolic regression problems using BRF as a performance measure does not produce an algorithm-portfolio with better performance than the performance of the best algorithm in the portfolio. The reason is that the systems with best performance are better modelled using performance models. A similar result is obtained when the class of problems is Boolean induction. In this case, as well as for the case of symbolic regression of rational functions, some of the best models are obtained with performance models.

Given that in order to tackle the algorithm selection problem, one only needs one modelling technique, we can combine both modelling techniques. That is, we can decide to model each system using the technique that produces the best model in the cross-validation of the training set.

For the symbolic regression of rational functions the modelling technique used in the best systems is the performance model. Thus, the results of the algorithm selection problem are equivalent to the ones showed in Chapter 6. For the case of Boolean induction problems on all GEP systems, the performance model is used while for all other systems the hardness models are used.

The result of mixing our modelling techniques is that the average performance of the portfolio is improved in contrast to using only the performance model. The average performance of the portfolio on the validation set is 0.8802 which is a better average performance than the one obtained when using only the performance models.
8.8 Summary

This chapter presented a modelling technique based on two hardness measures. This is inspired by the models found in the algorithm selection literature. Those models are built by selecting some features of a problem that are related to the hardness of that problem. Following a similar approach, for the Boolean induction problems, we used the hardness measure described in [32]. By extending this hardness measure, we developed a hardness measure for continuous functions. Then, we presented how these measures are used to create a model for the performance of EPAs. We decided to combine these measures into a linear equation in order to produce a model of performance.

To experimentally test the quality of the models produced by this approach, we modelled a variety of EPAs. The algorithms modelled include three versions of GP with four different parameter settings each; three versions of GEP; one version of CGP; and two versions of SIHC (one with four different parameters settings). Almost all the models created have a RSE value well below 1, indicating that these models are predicting better than the mean.

We used the RSE values of the models created to compare the modelling technique described here against the performance model presented in Chapter 4. We observed that the hardness-models have lower RSE values in almost all the Boolean induction problems and rational function with NBRF as performance. For the BRF performance measure, hardness models obtained the best RSE value in 10 out of 21 systems. These results are very encouraging. At first one might be tempted to discard the performance models when modelling EPAs. However, there are some configurations of algorithms and problems where the predictions made by the hardness model are not accurate enough. Also, performance models can be obtained without any prior knowledge.

We also tested the idea of modelling each system with the technique that produces the lower RSE value in cross-validation. We tackled the algorithm selection problem with the hardness models and compared it with a combination of hardness models and performance models. The result was that when all the systems are modelled with the hardness models the average of the portfolio is worse than the average of the best algorithm in it. However, when there is a combination of models, we obtained a better average performance than the best algorithm in the portfolio and also a better average performance than the one obtained when the models are created only with the performance models.
Chapter 9

Conclusions

In this chapter, we present the conclusions of this thesis. In Section 9.1, we review the achievements of this thesis. Section 9.2 presents the possible limitations of the two modelling techniques developed in this thesis. Finally, in Section 9.3, we make a number of suggestions about the possible future works.

9.1 Achievements

In this thesis, we presented a set of techniques for the building of efficient and accurate models for the performance of different types of evolutionary program induction algorithms working on a variety of problems. Specifically, we proposed two modelling techniques: Chapter 4 presented the performance models and Chapter 8 described the hardness model. With these two techniques, we modelled the performance of different algorithms including: three version of GP, three versions of GEP, two versions of SIHC, and one CGP. Furthermore, our tests considered two different classes of problems: symbolic regression of rational functions and Boolean induction (4 inputs and 1 output) which were used with the GP, GEP, CGP and SIHC systems.

The two modelling techniques proposed in the thesis are suitable for modelling different algorithms and problems. That is, the hardness models are specialised to predict the performance of the two classes of problems presented in this thesis, while the performance models are general models that can be applied to a variety of algorithms and problems.

The generality of the performance models comes from the heuristics used to make predictions. The performance models use the concept of closeness between the problem for which performance is being estimated and some reference problems (set $S$) previously automatically selected. This is why, in principle, one can apply the performance models to any algorithms and class of problems where one has a suitable function to measure the similarity between any pair of problems belonging to the class under study.

To illustrate the generality of our modelling technique, we modelled one ANN learning algorithm applied
to another class of Boolean induction problem (3 inputs and 2 outputs); and two human designed heuristics for the off-line bin packing problem.

The hardness model was inspired by models used to solve the algorithms selection problems. The models used in the algorithm selection problem are based on a set of features that describe the problem whose performance is being predicted and are related to hardness of the problem. Typically, these features are defined by an expert and are manually selected. Naturally they are particular for each class of problems.

An important similarity between our models and those used in algorithm selection techniques is that all these models are linear. One might wonder why such linear models work so well. We believe that this is because the features used as input to such models are highly informative in the sense that these may be correlated with the hardness of the problem. From another perspective, using a linear equation may enable the generalisation of the model. That is, given the limitations of the model, it is harder to learn the noise. The result is that the model is able to make accurate predictions on unseen cases.

The difference between our models and other approaches used in the evolutionary community is that our models can be used to predict the performance of evolutionary systems (EPAs, in particular) in unseen problems. Other techniques, such as the fitness distance correlation, fdc, and the negative slope coefficient, nsc, reviewed in Chapter 2, can only predict if a particular problem is hard or easy, but not precisely how hard or easy it is in relation to a chosen performance measure. Instead, our approaches provide a much more precise answers. Also, the execution of our models involves an extremely low computational load. Of course this comes at the price of running the system on a training set of problems to build the model in the first place (a preliminary effort which is not required to calculate either fdc or nsc). However, this cost is similar to what one needs to afford to compare the performance of different algorithms empirically by performing repeated runs over a variety of problems. The difference here is that once our models are constructed, they can be used over and over again to test performance on new problems, whilst empirical testing requires effectively running all the systems we want to compare on the new problem to see how they behave.

There are many possible applications for our models. As discussed in Chapter 6 it is, for example, possible to determine the best system for a job. We tested this, building algorithm portfolios which had significantly better average performance than the best overall algorithm in the portfolios. Moreover, in Chapter 8, we showed that an algorithm portfolio built with a mixture of performance models and hardness models performed better than a portfolio built using only one technique.

In Chapter 7 we illustrated that by comparing models it is possible to find out what similarities and differences there are between algorithms across whole problem classes (i.e., beyond a specific problem). The analysis performed led to four algorithm taxonomies. Among other things, these revealed that evolution program-induction systems with a population of 1000 individuals are only slightly sensitive to the choice of genetic operator rates, while generally the reproduction and selection strategies influence performance more significantly. The taxonomies created showed that for smaller populations sizes the genetic operator rates
are important. It was also shown that algorithms that are similar but rely on different representations (i.e., GP vs GEP) may produce considerably different behaviours.

9.2 Limitations

One of the principal limitations of the two modelling techniques described is that one needs to have a training set in order to instantiate either Equation (4.5) or Equation (8.5). Of course, this problem is inherent to any supervised learning technique.

The need for a training set imposes some limitations on which problems it is possible to model. That is, in order to apply our approach, one needs to make sure that there are enough problems in the class of problems of interest. This requirement really limits our approaches for problems where the execution of the algorithm is very slow and, therefore, estimating the performance of the algorithm is even slower.

Regarding the performance models, we find that the closeness measure also imposes some sort of limitation. That is, for any representation we use for the problems, we need to find a function that can measure how similar or different two problems are, i.e., function $d$. Furthermore, as we have seen in Section 5.3.1, having found a closeness measure $d$, is not a warranty that the chosen $d$ will produce the best performance model. Nonetheless, experimentally we have seen that almost all closeness measures produce acceptable models. As a rule of thumb, we have found that selecting the function used to compute the fitness of the individuals as closeness measure generally produces models with good accuracy.

The hardness models have been tested on different domains (symbolic regression problems and Boolean induction problems). Nonetheless, hardness models are not general enough to be suitable for all classes of problems. That is because, the hardness models firstly require that the problems be specified using a set of inputs. Secondly, there must exist a function capable of measuring the distance between pairs of inputs.

9.3 Future Work

This section suggests possible future work. It is organised as follows. Section 9.3.1 describes a number of avenues that need to be explored in order to improve the quality of the performance models. Furthermore, it highlights some similarities between this modelling technique and Support Vector Regression. Section 9.3.2 presents a number of ideas to improve the performance of the hardness models. Section 9.3.3 proposes future work on algorithm selection problems. Finally, Section 9.3.4 describes how the taxonomies of algorithms can be enhanced to increase their practicality.

9.3.1 Performance Model

In Section 5.3.1, we showed that some closeness measures produce better models than others. It is likely that there are even better measures than the ones we settled for in this thesis. In future research, we want to use
GP to obtain even more predictive closeness measures. Finally, we want to explore whether there are some performance measures and problem classes for which the approach is particularly suitable or unsuitable and determine why.

Subsequent to the introduction of the performance models, we have realised that there are some similarities between our model (Equation (4.5)) and a regression technique called Support Vector Regression (SVR)[23]. SVR has the form of:

\[
f(t) = \sum_{i=1}^{N} (\alpha_i^* - \alpha_i) \cdot K(x_i, t) + b,
\]

(9.1)

where \(\alpha_i^*, \alpha_i\) and \(b\) are constants, \(\{x_1, x_2, \ldots, x_N\}\) is a set of reference vectors called support vectors, \(t\) is a vector, and \(K\) is a kernel function.

Clearly, Equation (4.5) and Equation (9.1) are equivalent. One only needs to note that \(d = K, a_p = (\alpha_i^* - \alpha_i), a_0 = b\) and \(S = \{x_1, x_2, \ldots, x_N\}\). However, the procedures used to instantiate the equations are different. In SVR, an optimisation problem is solved to obtain \(b, \alpha, \alpha^*\) and \(x\), while our procedure uses a subset selection algorithm, specifically LARS, and a cross-validation technique in the training set to obtain \(S\). Finally, a LSM is used to obtain the \(a\) coefficients in our model.

Instead, in SVR one tries to minimise the following function:

\[
\frac{1}{2}||w||^2 + C \cdot \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(y_i - f(x_i)),
\]

(9.2)

where \(w = ((\alpha_1^* - \alpha_1), \ldots, (\alpha_N^* - \alpha_N))\), \(C\) determines the trade-off between errors \(\mathcal{L}(y_i - f(x_i))\) and the model complexity (i.e., \(||w||^2\)). \(y_i\) is the actual output of the process when the input is \(x_i\). Finally, \(\mathcal{L}\) is defined as:

\[
\mathcal{L}(x) = \begin{cases} 
0 & \text{if } |x| < \epsilon \\
|x| - \epsilon & \text{otherwise.}
\end{cases}
\]

(9.3)

\(\mathcal{L}\) is defined in such a way that it creates a pipe around the actual measurements. The objective is to remove from the errors those predictions that are close enough to the actual readings.

In future work, we will study the difference between these two techniques paying particular attention to the advantages or disadvantages that one technique has over the other.

The similarities between the performance models and SVR, provides an explanation for why the performance models are able to predict the performance of different searchers or problem solvers. Therefore, in future work we will investigate the applicability of our modelling technique to other sort of predictions like time series prediction and classification, just to mention a few.
9.3.2 Hardness Model

In Section 8.6, we saw that the hardness model is able to produce accurate models. However, for some algorithms and some problems the predictions obtained are not as accurate as we wish them to be. Therefore, in future research, we will try to improve accuracy by combining the hardness measures ($\vartheta_i(f)$) with different types of equations. For example, we will test the quality of the models when the components ($\vartheta_i(f)$) are combined in a quadratic equation or we can go even further by using GP to find a function whose inputs are $\vartheta_i(f)$.

Although the hardness models were specifically designed to models Symbolic regression problems and Boolean induction problems, there are no constraints that limit applicability to only these two classes of problems. In future work, we will test the hardness measures in other domains that may not belong to program-induction algorithms. As we mentioned above, the only requirement is that the problems are specified using a set of inputs and that the distance between the inputs can be measured.

9.3.3 Algorithm Selection

In Chapter 6, we applied the performance model to create an algorithm portfolio. The procedure we used to create the algorithm portfolio required a set of algorithm where each algorithm beats all other algorithms in some problems. Given that the performance models can be used to automatically create a taxonomy of algorithms, it is interesting to combine these methodologies in the creation of the portfolio. The idea is to use the information given by the taxonomy in order to include those algorithms that have a good performance and that behave differently according to the taxonomy. At least in principle, this procedure will give greater variety to the algorithms in the portfolio and perhaps as a result better performance will be obtained.

Here, the algorithm selection problem was solved when each algorithm is associated to only one performance value. We can think of this as a single-objective algorithm selection problem. However, we can generalise the idea to the core where the performance of each algorithm is measured using many performance measures. Thus, the performance of each algorithm is represented using a vector, instead of a scalar value. In this general case, we have a multi-objective algorithm selection problem. Clearly, we can start handling this case by creating, for each algorithm, as many performance models as the different performance measures used. Although this approach seems feasible, we still need to decide how to combine all the models in order to choose which algorithm of the portfolio to run. This multi-objective algorithm selection problem will be analysed in future work.

9.3.4 Algorithm Taxonomies

In Chapter 7, we presented an automatic procedure to create taxonomies for program-induction algorithms. These taxonomies were created based on the performance of the algorithms involved that completely differ from the taxonomies created based on syntactic difference between algorithms. Although the taxonomies
presented include different algorithms with different parameters, there are still some algorithms that missing in these taxonomies. In future work, we will extend the taxonomies by using a variety of other algorithms or by varying the parameters of the algorithms already in the taxonomy. The final goal would be to provide a way to compare a new technique with all the algorithms already in the taxonomy. This sort of comparison will highlight the similarities and differences that a new technique has with respect to the algorithms in the taxonomy.
Bibliography


Peter Martin. Building a taxonomy of genetic programming. In Spector et al. [118], page 182.


[122] Matthew J. Streeter. Two broad classes of functions for which a no free lunch result does not hold. In Cantú-Paz et al. [14], pages 1418–1430.


