Arcuri, A., White, D.R., Clark, J., and Yao, X. (2008) *Multi-objective improvement of software using co-evolution and smart seeding.* Lecture Notes in Computer Science, 5361 (2008). pp. 61-70. ISSN 0302-9743

http://eprints.gla.ac.uk/55375

Deposited on: 22 February 2012

Multi-Objective Improvement of Software using Co-evolution and Smart Seeding

Andrea Arcuri[1], David Robert White[2], John Clark[2], and Xin Yao[1]

[1] The Centre of Excellence for Research in Computational Intelligence and Applications (CERCIA), The School of Computer Science, The University of Birmingham, Edgbaston, Birmingham, B15 2TT, UK.
[2] Department of Computer Science, University of York, YO10 5DD, UK.

Abstract. Optimising non-functional properties of software is an important part of the implementation process. One such property is execution time, and compilers target a reduction in execution time using a variety of optimisation techniques. Compiler optimisation is not always able to produce semantically equivalent alternatives that improve execution times, even if such alternatives are known to exist. Often, this is due to the local nature of such optimisations. In this paper we present a novel framework for optimising existing software using a hybrid of evolutionary optimisation techniques. Given as input the implementation of a program or function, we use Genetic Programming to evolve a new semantically equivalent version, optimised to reduce execution time subject to a given probability distribution of inputs. We employ a co-evolved population of test cases to encourage the preservation of the program's semantics, and exploit the original program through seeding of the population in order to focus the search. We carry out experiments to identify the important factors in maximising efficiency gains. Although in this work we have optimised execution time, other non-functional criteria could be optimised in a similar manner.

1 Introduction

Software developers must not only implement code that adheres to the customer's functional requirements, but they should also pay attention to performance details. There are many contexts in which the execution time is important, for example to aid performance in high-load server applications, or to maximise time spent in a power-saving mode in software for low-resource systems. Typical programmer mistakes may include the use of an inefficient algorithm or data structure, such as employing an $\Theta(n^2)$ sorting algorithm.

Even if the correct data structures and algorithms are employed, their actual implementations might still be improved. In general, compilers cannot restructure a program's implementation without restriction, even if employing semantics-preserving transformations. The alternative of relying on manual optimisation is not always possible: the performance implications of design decisions may be dependent on low-level details hidden from the programmer, or be subject to subtle interactions with other properties of the software.

To complicate the problem, external factors contribute to the execution time of software, such as operating system and memory caches events. Taking into account these

factors is difficult, and so compilers usually focus on optimising localised areas of code, rather than restructuring entire functions.

More sophisticated optimisations can be applied if we take into account the probability distribution of the *usage* of the software. For example, if a function takes an integer input and if we know that this input will usually be positive, this information could be exploited by optimising the software for positive input values.

In this paper we present a novel framework based on evolutionary optimisation techniques for optimising software. Given the code of a function as input to the framework, the optimisations are performed at the program level and consider the probability distribution of inputs to the program. To our best knowledge, we do not know of any other system that is able to automatically perform such optimisations.

Our approach uses Multi-Objective Optimisation (MOO) and Genetic Programming (GP) [1]. In order to preserve semantic integrity whilst improving efficiency, we apply two sets of test cases. The first is co-evolved with the program population [2] to test the semantics of the programs. The second is drawn from a distribution modelling expected input, and is used to assess the non-functional properties of the code. The original function is used as an oracle to obtain the expected results of these test cases.

Evolving correct software from scratch is a difficult task [2], so we exploit the code of the input function by seeding the first generation of GP. The first generation will not be a random sample of the search space as is usually standard in GP applications, but it will contain genetic material taken from the original input function. Note that this approach is similar to our previous work on *Automatic Bug Fixing* [3], in which all the individuals of the first generation were equal to the original incorrect software, and the goal is to evolve a bugfree version. A similar approach has also been previously taken in attempting to reduce the size of existing software [4].

We present a preliminary implementation of the novel framework, and we validate it on a case study. We then apply systematic experimentation to determine the most important factors contributing to the success of the framework. Although our prototype is still in an early stage of development, this paper gives the important contribution of presenting a general method to automatically optimise code using evolutionary techniques. We are also able to provide some guidance to other practitioners in applying such an approach, based on our analysis of empirical results.

The paper is organised as follows. Section 2 describes in detail all the components of the novel framework, whereas Section 3 presents our case study. Section 4 describes our results and Section 5 suggests further work.

2   Evolutionary Framework

An overview of our framework is given in Figure 1. The framework takes as input the code of a function or program, along with an expected input distribution, and then it applies GP to optimise one or more non-functional criteria. Note that in our experimentation, we chose to parameterise the use of MOO and Coevolution in order to assess their impact on the ability of the framework to optimise non-functional properties of the software. The main differences from previous GP work are how the first generation is seeded, how the training set is used and generated, the particular use of multi-
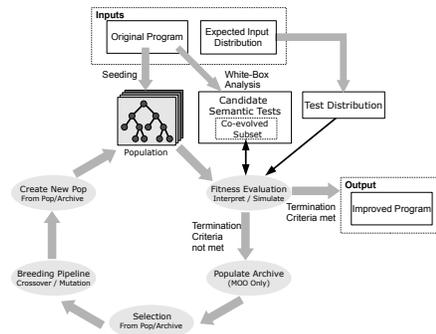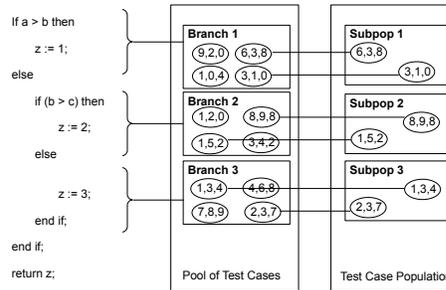
Fig. 1. Evolutionary Framework

Fig. 2. The Relationship between a Program and the Semantic Test Set Population

objective optimisation, and the employment of simulation and models in estimating non-functional properties of individuals.

2.1 Seeding Strategies

Usually, in GP applications the first generation is sampled at random, for example, using Koza's ramped half-and-half initialisation method. Evolving bugfree software from scratch with GP is an hard task [5], but in our case we have as input the entire code of the function that we want to optimise, and we can exploit this information.

Different seeding strategies can be designed, and this is a case of the classic "exploration versus exploitation" trade-off that is so often an issue in heuristic search, and in particular evolutionary computation. On one hand, if we over-exploit the original program we might constrain the search in a particular sub-optimal area of the search space, i.e. the resulting programs will be very similar to the input one. On the other hand, ignoring the input genetic material would likely make the search too difficult. The point here is that, although we do not want a final program that is identical to the input one, its genetic material can be used as *building blocks* in evolving a better program. This has interesting implications for understanding how GP achieves its goal: *can building blocks be recombined in different ways to improve performance?*

In this work we consider a simple strategy: given a fraction δ of the initial random population, then δ individuals will be replaced by a copy of the input function. The remaining individuals are generated using a standard initialisation method.

2.2 Preserving Semantic Equivalence

Modifications to the input program can compromise its original semantics and our goal is to output an improved yet semantically equivalent program. It is important that our evaluation of individuals is effective in testing the semantics of new programs against the original. Exhaustive testing is usually impossible, and any testing strategy is therefore open to exploitation by an evolutionary algorithm through over-fitting.

To improve the effectiveness of our fitness evaluation method, we employ a form of coevolution, in principle similar to that used by Hillis [6]. Before the evolutionary algorithm begins, we first generate a large set of test cases using a White Box testing criterion [7], specifically *branch coverage*. This set is partitioned into subsets, one for each branch of the program. The partitioning ensures a degree of behavioural diversity amongst test cases.

The test set is then co-evolved as a separate population (the "training set"), from a selection from the larger pool produced prior to evolution. This training set is also partitioned, so that it samples each branch of the original program.

At each generation, the GP individuals are tested with the test cases in the training set. The sum of the errors from the expected results is referred to as the *semantic score* and is one component of the fitness of a GP individual. Figure 2 illustrates the relationships between the program and test set populations.
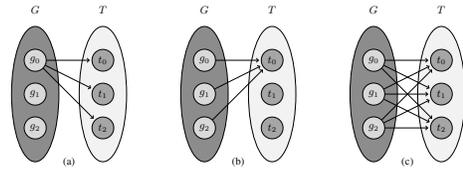

Fig. 3. G is the population of programs, whereas T is the population of test cases. (a) shows the test cases used to calculate the fitness of the first program g_0, (b) shows the programs used to calculate the fitness of the first test case t_0. Note the common arc between the first program and the first test case. Finally, picture (c) presents all possible $|G| \cdot |T|$ connections.
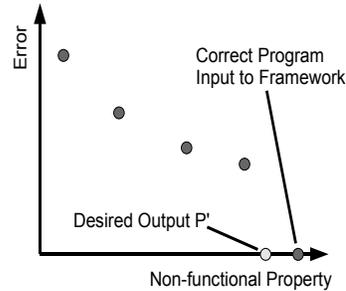

Fig. 4. A Pareto Front Composed of Five Programs in Objective Space

2.3 Evaluating Non-functional Criteria

To evaluate non-functional properties of individuals, a separate training set from that used to evaluate the semantic score is employed. The set is drawn from the expected input distribution provided to the framework, which could be based on probe measurement of software. For each non-functional criterion, a score is calculated for GP individuals using this set. The final fitness function of a GP individual will be composed of these scores and the semantic score. The set of tests is resampled at the start of each generation, to prevent overfitting of non-functional fitness for a particular set of inputs.

In this paper, we will estimate (by modelling and simulation) the number of CPU cycles consumed by each individual, assuming a uniform distribution of integer inputs for the case study. Note that this work is distinct from previous work on program compression [4] as the number of cycles used will depend on the path taken within a program. The framework can be extended to handle other types of non-functional criteria.

Simulation The cycle usage of an individual can be estimated using a processor simulator and here we have used the M5 Simulator [8], targeted for an ARM Microprocessor. The parameters of the simulator were left unchanged from their default values. Individuals are written out by the framework as C Code and compiled with an ARM-Targeted GCC cross-compiler. A single program is executed along with test code that executes the given test cases, and a total cycle usage estimate provided.

Whilst simulation does not perfectly reflect a physical system, it is worth noting that we are only concerned with *relative accuracy* between individuals, and also that the accuracy of simulation is an issue beyond the scope of our framework: we can easily incorporate alternatives or improvements.

Model Construction Compiling and then testing each individual in a simulator can be computationally expensive. In this work we have carried out a large quantity of experiments as part of the analysis of the problem. Thus, we opted to study the approach of modelling the cycle usage as a linear model of instructions executed:

$$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots \beta_n x_n + \epsilon$$

Where Y is the estimated cycles consumed by a program, $x_1 \ldots x_n$ are the frequencies that each of the n instructions appear within a program, and the coefficients $\beta_1 \ldots \beta_n$ are an estimate of the cost of each instruction. ϵ is the noise term, introduced by factors not considered by the other components of the model. This is a simplification, because the ordering of the instructions affects the total cycles consumed due to pipelining and caching and because subsequent compiler optimisations will be dependent on the program structure.

To use such a model, the coefficients must be estimated. We achieved this by executing one large evolutionary run of the framework, and logging the frequencies with which each instruction appeared in each individual, and their corresponding cycle usage. Least Squares Linear Regression was then used to fit this model. It was possible to verify the relative accuracy of this model for the data points used in constructing it. As we are using tournament selection, we compared the results of using a model to carry out a tournament size 2 against the results of using the simulator results. The model was found to be in agreement with the simulator 65% of the time. It was not clear if this would be sufficient, and therefore the model was treated as a parameter of our experiments.

Using the Model During experimentation, we execute the individuals through interpration within the framework, whilst storing a profile of the nodes visited during evaluation. This profile is then used in conjunction with the model provided to the framework to estimate the number of the cycles the individual would consume. Thus a combination of interpretation and model-based estimation (or alternatively, simulation) can be used by the framework.

2.4 Multi-Objective Optimisation

Our framework is faced with the challenge of optimising one or more non-functional properties, whilst retaining the correct semantics of the original program provided as input. This problem can be formulated as Multi-Objective Optimisation (MOO), indeed this is the approach we have taken previously to a similar problem [9]. In that work, we were searching for a *set of possible trade-offs* between non-functional and function properties, but here we are concerned only with finding a *single program*. Hence it was unclear whether a fully MOO approach based on pareto-dominance was necessary.

We therefore adopted two approaches to combining objectives in fitness evaluation. The first was to use a simple linear combination of the functional and non-functional fitness measures. The second is to use the Strength Pareto Evolutionary Algorithm Two (SPEA2) [10]. This is a popular pareto-based method that attempts to approximate the pareto-front in objective space, as illustrated by Figure 4.

In Figure 4, it is assumed that the aim is to minimise both the non-functional property of the software and its error, that is both fitness components are *cost functions*. A pareto front would consist of the darker points, where no improvement in one objective can be made without worsening fitness in one of the other objectives. Our framework would like to find the point P', a program with zero error and an improved non-functional fitness.

One possible justification of using a pareto-based MOO approach is the building block hypothesis often used to provide some rational for genetic recombination in evolutionary algorithms. SPEA2 should find a set of programs that provide varying levels of error for different non-functional property values. Recombination between these smaller building blocks may produce re-orderings of instructions and new combinations that provide the same functionality but at a lower non-functional cost.

In our experimentation, we chose to make the MOO component of the framework a parameter, in order to establish what impact the two approaches would have on the success of the optimisation process.

3 Case Study

3.1 Software Under Analysis

In our experiments, we analysed the *Triangle Classification* (TC) problem [7]. We choose that particular function because it is commonly used in the software testing literature, and the first one on which theoretical results have been obtained [11]. Given three integers as input, the output is a number representing whether the inputs can be classified as the sides of either an invalid, scalene, isosceles or equilateral triangle.

We used two different implementations, respectively published in [12] and [13] and expressed in Java in Figures 5 and 6 respectively. Note that their return values have been changed to make them consistent. The two implementations are not semantically equivalent, because they have contain related to arithmetic overflows.

```
public int triangleClassification
(int a, int b, int c) {
  if (a > b) {int tmp = a; a = b; b = tmp;}
  if (a > c) {int tmp = a; a = c; c = tmp;}
  if (b > c) {int tmp = b; b = c; c = tmp;}
  if(a+b <= c)
      return 1;
  else {
        if(a == b && b == c) return 4;
        else if(a == b || b == c) return 3;
        else return 2;}
}
```

Fig. 5. 1st TC version [12].

```
public int triangleClassification
(int a, int b, int c) {
  if(a<=0 || b<=0 || c<=0) return 1;
  int tmp = 0;
  if(a==b) tmp += 1;
  if(a==c) tmp += 2;
  if(b==c) tmp += 3;
  if(tmp == 0){
     if((a+b<=c) || (b+c <=a) || (a+c<=b)) tmp = 1;
     else tmp = 2;
     return tmp;}
  if(tmp > 3) tmp = 4;
  else if(tmp==1 && (a+b>c)) tmp = 3;
  else if(tmp==2 && (a+c>b)) tmp = 3;
  else if(tmp==3 && (b+c>a)) tmp = 3;
  else tmp = 1;
  return tmp;
}
```

Fig. 6. 2nd TC version [13].

3.2 Experimental Method

The framework was implemented in Java, and we used ECJ 16 [14] for the GP system. In particular, we used Strongly Typed Genetic Programming [15]. All the parameters of the framework that are not stated in paper have the default values in ECJ, as inherited from the koza.params parameter file.

For each TC version ($V1$ and $V2$) we carried out distinct experiments with 2 different cost models ($M1$ and $M2$), for a total of 4 independent sets of experiments. In one model, each GP primitive has unitary estimated cycle cost ($M1$), whereas in the second model ($M2$) these costs have been estimated by least squares regression on data collected from a run using simulator.

For each group of experiments, we performed a full factorial design [16] of 8 parameters that we considered most important. Table 1 shows their high and low values. The total number of tested configurations was $4 \cdot 2^8 = 1024$. However, the SPEA2 archive is used only when MOO is employed, hence 256 experiments are redundant.

The probability that a tree is not affected by either crossover or mutation is 0.1; test case population size of 200, with an archive of 50 elements and a main pool of 2000 test cases; the cycle score is evaluated on 100 test cases that are sample at each generation with uniform distribution of values in $\{-127, \ldots, 128\}$.

There are 36 GP primitives: 3 input variables, 1 other integer variable, read and write of variables, 1 variable wrapper, 10 integer constants, 5 arithmetic operators, 2 boolean constants, 8 boolean operators and 4 commands. There are 4 node return values: command, integer value, integer variable and boolean value.

If P is the program given as output by the framework, we are interested whether P is faster than the input program. Given an output, we validate P against an independent set of $10,000$ test cases. If P fails any of those test cases, the framework has failed to produce a semantically equivalent program, and P will be replaced by the original program. Note that passing $10,000$ test cases does not guarantee the equivalence of semantics, so the output programs need to be manually checked at the end of a run.

Parameter	Id	Low Value	High Value
Probability of Crossover (P_c)/Mutation(P_m)	X1	0.1/0.8	0.8/0.1
Population Size (S) and Generations (G)	X2	50/1000	1000/50
Tournament Selection Size	X3	2	7
Types of Mutations	X4	1	6
Clone Proportion δ	X5	0	1
Co-evolution Enabled	X6	false	true
SPEA2 MOO Used	X7	false	true
SPEA2 Archive Proportion	X8	$\frac{1}{9}$	1

Table 1. Factorial design of 8 parameters. Note that $P_m = 0.9 - P_c$ and $S \cdot G = 50000$ such that the total number of fitness evaluations remains constant. For the same reason, when MOO is employed, the population is reduced by the SPEA2 archive size. If co-evolution is not employed, the test cases are simply sampled at random at each generation. If MOO is not employed, the semantic and the cycle scores are linearly combined, with a weight of 128 for the semantic score. A mutation event is a single mutation from a pool of ECJ mutation operators is applied.

The performance of P is evaluated with the *gain score*, that is the difference of the cycle scores of the original program and P. These cycle scores are evaluated on 100 test cases. The faster P is, the higher gain score it will receive. If P is not correct, then the gain score is 0. It is possible that the gain score assumes a negative value.

For each of the 1024 configurations, we ran them 100 times and recorded the gain score. For each of the 4 groups of experiments we report an ANOVA analysis of the results in Table 2, whereas the configurations that gives the highest single and average gain score are reported in Table 3. Moreover, for each best configuration in Table 3 we chose the best program (out of 100 trials), and we evaluated the estimated real gain score by running it in the simulator against the original program (on 1,000 input triplets over the expected input distribution).

Configuration	X1	X2	X3	X4	X5	X6	X7	X8
V1 M1	0.0001	0	0	0.3396	0	0	0	0
V1 M2	0	0.0816	0.2707	0	0	0	0	0.8529
V2 M1	0.0026	0	0	0.2094	0	0	0	0
V2 M2	0	0	0	0	0	0	0	0.7858

Table 2. P-values of the ANOVA tests run on the 4 different types of experiments.

4 Results and Conclusions

Table 2 demonstrates that the design decisions made in selecting each parameter value have a significant impact (i.e. have a small p-value) on the performance improvement achieved. Only X3, X4 and X8 are significant for only part of the experiments. All are concerned with the amount of exploration the search performs, and it is conjectured that the significance of these parameters will be problem-specific.

Config.	X1	X2	X3	X4	X5	X6	X7	X8	Average	Max	Variance	Best
V1 M1	0.8/0.1	1000/50	2	6	1	true	false	-	1330.7	1826.0	163,700	7355
	0.8/0.1	50/1000	7	1	1	true	false	-	1052.7	2507.0	305,100	11610
V1 M2	0.1/0.8	50/1000	7	1	1	true	false	-	175.8	3402.9	2,211,000	-1618
	0.1/0.8	50/1000	7	6	1	true	false	-	-1528.2	3609.5	1,991,800	-503
V2 M1	0.1/0.8	1000/50	7	6	1	true	false	-	570.3	1475.0	181,600	6645
	0.8/0.1	50/1000	7	6	1	true	false	-	237.4	1519.0	105,700	6645
V2 M2	0.1/0.8	1000/50	2	1	1	true	true	1	39.3	1490.4	77,700	-2198
	0.8/0.1	50/1000	7	1	1	true	false	-	-318.0	5401.1	2,688,700	-11242

Table 3. For each of the 4 configurations, the parameter settings that result in the highest average and max gain scores are reported, as well as their best performance gains.

The best individual data in Table 3 shows that improvements were possible for both original programs. Model 2, based on the simple linear model building approach, performed poorly: we recommend that hand-crafted models of resource usage or full simulation should be used.

Seeding the initial population based on the original program is a useful technique that should be used. Similarly, the application of co-evolution is an effective measure to improve performance.

When employing Genetic Programming in general, a large population for a smaller number of generations is usually more effective than a smaller one evolved over a large number of generations, due to the prevalent problem of bloat [17]. However, in our experiments we see exactly the opposite trend where small populations are more successful in that they produce the largest improvements in program speed. It would usually be expected that a higher number of generations tends to lead to over-fitting and fewer correct programs over a succession of runs. However, the efficiency gains are best in the very few cases in which the resulting programs are actually correct.

The fact that a pareto-based MOO approach mostly provides worse results may be due to the fact that the programs we analysed in our case study can be optimised to some extent using multiple mutations, and with few changes in the source code. Hence a search concentrated around the input program gives better results, rather than spread across a range of program shapes and sizes. It is therefore possible that pareto-based MOO will find superior solutions than a linear combination given more resources.

It is worth noting that for the non-binary parameters (e.g., crossover rate) we analysed only low and high values in our experiments. The best tunings likely lie within those extremes and it is likely that better results than in Table 3 could be obtained by tuning these parameters.

5 Summary and Future Work

In this paper we have presented a novel framework for improving non-functional criteria of software. The framework has been successfully used for evolving new correct and faster versions of the programs in our case study. Regarding the quality of the final outcomes, the experiments also showed expected and unexpected roles of some parameter settings.

Immediate future work is to test if these results hold for other problems. We would also like to further investigate optimal parameter settings, in particular the cloning proportion used. Also, alternative seeding strategies could be investigated, potentially as an opportunity to investigate GP schema theory [17] where seeding according to schemas may have a beneficial effect.

As already discussed, further work using MOO and extended evolutionary runs may allow us to provide more guidance on parameter selection.

6 Acknowledgements

References

1. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press (1992)
2. Arcuri, A., Yao, X.: Coevolving programs and unit tests from their specification. In: IEEE International Conference on Automated Software Engineering (ASE). (2007) 397–400
3. Arcuri, A., Yao, X.: A novel co-evolutionary approach to automatic software bug fixing. In: IEEE Congress on Evolutionary Computation (CEC). (2008) 162–168
4. Langdon, W.B., Nordin, P.: Seeding genetic programming populations. In: Proceedings of the European Conference on Genetic Programming (EuroGP). (2000) 304–315
5. Reformat, M., Xinwei, C., Miller, J.: On the possibilities of (pseudo-) software cloning from external interactions. Soft Computing **12**(1) (2007) 29–49
6. Hillis, W.D.: Co-evolving parasites improve simulated evolution as an optimization procedure. Physica D **42**(1-3) (1990) 228–234
7. Myers, G.: The Art of Software Testing. Wiley, New York (1979)
8. Binkert, N., Dreslinski, R., Hsu, L., Lim, K., Saidi, A., Reinhardt, S.: The M5 simulator: Modeling networked systems. IEEE Micro **26**(4) (2006) 52–60
9. White, D.R., Clark, J., Jacob, J., Poulding, S.: Searching for Resource-Efficient Programs: Low-Power Pseudorandom Number Generators. In: GECCO 2008: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation. (2008) 1775–1782
10. Zitzler, E., Laumanns, M., Thiele, L.: SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Technical Report 103, Swiss Federal Institute of Technology (2001)
11. Arcuri, A., Lehre, P.K., Yao, X.: Theoretical runtime analyses of search algorithms on the test data generation for the triangle classification problem. In the International Workshop on Search-Based Software Testing (SBST), 2008
12. McMinn, P.: Search-based software test data generation: A survey. Software Testing, Verification and Reliability **14**(2) (June 2004) 105–156
13. Miller, J., Reformat, M., Zhang, H.: Automatic test data generation using genetic algorithm and program dependence graphs. Info. and Software Technology **48**(7) (2006) 586–605
14. ECJ: Evolutionary computation in Java, http://www.cs.gmu.edu/∽eclab/projects/ecj/.
15. Montana, D.J.: Strongly typed GP. Evolutionary Computation **3**(2) (1995) 199–230
16. Montgomery, D.C.: Design and Analysis of Experiments. John Wiley & Sons (2006)
17. Langdon, W.B., Poli, R.: Foundations of Genetic Programming. Springer-Verlag (2002)