Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Fitness Predictors in Genetic Programming

Jan Mayer

Supervisor: Ing. Petr Pošík, Ph.D.
May 2020

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Mayer Jan** Personal ID number: **474529**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Branch of study: **Computer and Information Science**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Fitness Predictors in Genetic Programming

Bachelor's thesis title in Czech:

Prediktory fitness v genetickém programování

Guidelines:

Genetic programming is an evolutionary method for searching for algorithms that solve certain problem, or for searching a structured description of certain system, e.g. in the form of a mathematical expression. Evaluation of a candidate solution is usually done with respect to a big set of test cases which is time consuming. That's why the so-called fitness predictors were proposed. A fitness predictor is a small subset of the set of test cases which provides the evolution with similar information as the whole testing set. The goal of this project is the exploration, design and evaluation of methods for constructing the fitness predictors.
Elaboration guidelines:
1) Learn the principles of GP, fitness predictors and methods for their construction.
2) Design your own method of fitness predictor construction.
3) Compare the known and proposed fitness predictor methods.
4) Evaluate the methods with respect to the time requirements and test case efficiency.

Bibliography / sources:

[1] Schmidt, M.D., Lipson, H. Coevolution of Fitness Predictors. IEEE Trans. On Evolutionary Computation, Vol. 12, No. 6, 2008
[2] Drahošová, M., Sekanina, L., Wiglasz, M. Adaptive Fitness Predictors in Coevolutionary Cartesian Genetic Programming. Evolutionary Computation, Vol. 27, Issue 3, Fall 2019

Name and workplace of bachelor's thesis supervisor:

Ing. Petr Pošík, Ph.D., Analysis and Interpretation of Biomedical Data, FEE

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **10.01.2020** Deadline for bachelor thesis submission: **22.05.2020**

Assignment valid until: **30.09.2021**

Ing. Petr Pošík, Ph.D.	doc. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. _____
Date of assignment receipt

_____
Student's signature

Acknowledgements

I would like to express my gratitude to my supervisor Ing. Petr Pošík, Ph.D for providing guidance and feedback throughout this project.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, May 20, 2020

Abstract

In genetic programming, computer programs have to be evaluated on many test cases in order to measure their performance. To accelerate this process, smaller subsets of the training dataset called fitness predictors can be used. The known methods construct the fitness predictors using coevolution of fitness predictors and computer programs. In this thesis, we proposed a method of fitness predictor construction, which does not rely on coevolution. We compared the known and proposed methods using six symbolic regression problems. On two simple problems, the proposed method accelerated the evolution the most from all compared methods, but on the rest of the problems, it performed noticeably worse than the coevolved predictors.

Keywords: genetic programming, fitness predictors, coevolution

Supervisor: Ing. Petr Pošík, Ph.D.
FEE, Department of Cybernetics

Abstrakt

Počítačové programy v genetickém programování je třeba ohodnotit na velkém množství testů, aby mohla být změřena jejich kvalita. Tento proces může být urychlen, pokud jsou k ohodnocení použity pouze malé podmnožiny testovacího datasetu zvané fitness prediktory. Používané metody konstruují fitness prediktory pomocí koevoluce fitness prediktorů a počítačových programů. V této práci jsme navrhli metodu konstruující prediktory bez použití koevoluce. Porovnali jsme navrhnutou metodu a používané metody na šesti problémech symbolické regrese. Na dvou jednoduchých problémech urychlila navrhnutá metoda evoluci zdaleka nejvíce ze všech porovnávaných metod, ale na zbytku problémů byly její výsledky znatelně horší než výsledky prediktorů vytvořených koevolucí.

Klíčová slova: genetické programování, fitness prediktory, koevoluce

Překlad názvu: Prediktory fitness v genetickém programování

Contents

Figures

Tables

Chapter 1

Introduction

This thesis focuses on the problem of construction of fitness predictors in genetic programming (GP). The main goal of fitness predictors is to reduce the computational cost needed by GP to solve certain problems. More specifically, their purpose is to reduce the computational cost needed to evaluate the candidate programs without slowing down the evolutionary process. Evaluating the candidate programs can be very computationally demanding because the fitness of the programs is often computed using their performance on a large number of test cases. Fitness predictors help to speed this process up by allowing the GP run to evaluate the programs using a significantly smaller subset of the test cases called a fitness predictor.

Methods of constructing such fitness predictors were introduced in [10] and [3]. Both of these methods use coevolution to solve this problem. In this thesis, we want to find out if using coevolution is necessary or if a simpler method can construct predictors, which are comparably effective.

In chapter 2, we explain the main principles of GP and describe some of the basic techniques and algorithms used within a GP run. In chapter 3, we first define the fitness predictors and their goal, then we describe two known methods, which use coevolution, and propose a new method to construct the fitness predictors. In chapter 4, we present the experiments we conducted to compare the methods from chapter 3 and discuss their results. Chapter 5 summarizes the results of this thesis.

Chapter 2

Genetic Programming

Genetic programming is a collection of evolutionary computation techniques that allow computers to solve problems automatically [9]. It is a technique inspired by biological evolution. The evolving individuals in GP are computer programs represented by trees. These trees consist of functions (e.g. sin, XOR) and terminals (e.g. numeric constants, program inputs). An example of such tree can be seen in figure 2.5. The main idea is to start with a usually random population of individuals, select individuals based on their ability to solve the problem, apply genetic operations to them and repeat this process generation by generation until an individual with the ability to solve the problem sufficiently well is found. The idea can be seen in the following pseudocode:

```
1 Randomly create an initial population
2 repeat
3 |   Select individuals that will take part in the next generation based
  |     on their fitness
4 |   Create new individuals by applying genetic operations
5 until an acceptable solution is found, or some other stopping
    condition is met (e.g., a maximum number of generations is reached)
```

Algorithm 1: Run of a GP algorithm

Now let us take a look at the individual steps.

2.1 Creating an Initial Population

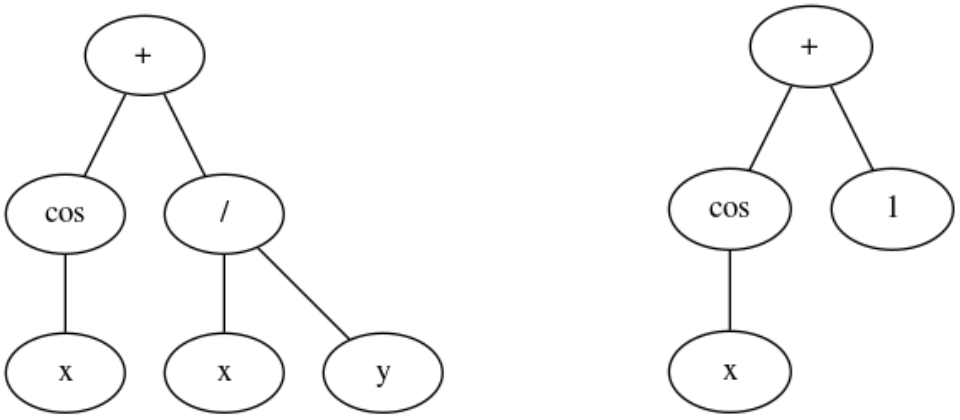
The two simplest methods are *full* and *grow*. Both generate trees that do not exceed a user defined maximum depth $depth_{max}$:

1. Full - generate the tree from the root. When adding a node, select a random function until $depth_{max}$ is reached, then add a random terminal here. This way every path from the root to a leaf is $depth_{max}$ long. (figure 2.1a)

2. Grow - generate the tree from the root. When adding a node select

random a function **or** a terminal. This way every path from the root to a leaf is at most $depth_{max}$ long. (figure 2.1b)

To ensure variety of shapes and sizes a combination called *ramped half-and-half* was introduced by Koza in [7]. Half of the population is generated by *grow* method and half by *full* method.



(a) : full **(b) :** grow

Figure 2.1: Example of trees generated by full and grow method with height limit 2

2.2 Selection

During selection, we choose individuals from the current generation, which will be used as parents for the next generation. There are many ways to do this, such as tournament selection or deterministic crowding selection[8]. When using these methods, better performing individuals (with respect to their fitness) have a higher chance of being selected.

To measure the performance of an individual, we need to define the so-called *fitness function*. More formally, we need

$$\text{fitness: } S \to \mathbb{R},$$

where S is a set of all possible individuals (solutions), which can be created during the evolutionary process. Since individuals are computer programs, their fitness is usually obtained by running them and measuring how *good* their results are. Here, by *good*, we mean how close they are to the results of an ideal solution.

To select the individuals, we need to evaluate either the whole or a significant part of the current population, which can often be very computationally demanding.

To illustrate the process of selection, we will describe the two mentioned selection methods:

1. Tournament selection - choose k random individuals from the current population and select the one with the best fitness value. k is called a tournament size. Repeat this process to create a new population.

2. Deterministic crowding selection - for this method, a distance function between two individuals must be defined. The selection method is described by the following pseudocode:

Input: two individuals from the current population: p_1 and p_2, two children created from p_1 and p_2: c_1, c_2

if $dist(p_1, c_1) + dist(p_2, c_2) \leq dist(p_1, c_2) + dist(p_2, c_1)$ **then**
  if $fitness(c_1) > fitness(p_1)$ **then**
    $p_1 \leftarrow c_1$
  else if $fitness(c_2) > fitness(p_2)$ **then**
    $p_2 \leftarrow c_2$
  end
else
  if $fitness(c_2) > fitness(p_1)$ **then**
    $p_1 \leftarrow c_2$
  else if $fitness(c_1) > fitness(p_2)$ **then**
    $p_2 \leftarrow c_1$
  end
end
return p_1, p_2

Function Deterministic crowding selection

2.3 Genetic operations

In GP, two main genetic operations are used:

1. Crossover - This operation takes two individuals (parents) and combines them together to create two new individuals (children). This is done by randomly selecting a crossover node in both parents and creating the children by replacing the subtrees rooted in the crossover nodes. An example of crossover can be seen in figure 2.2.

2. Mutation - This operation takes one individual and makes small changes to it. There are two main ways to achieve this:

   a. Randomly select a mutation node and replace the corresponding subtree by a randomly generated one. An example can be seen in figure 2.3.

   b. Randomly select a mutation node and replace the stored primitive by a randomly selecting a primitive from the primitive set with the same arity, as seen in the figure 2.4.

It is common to use these operations with specified probabilities.

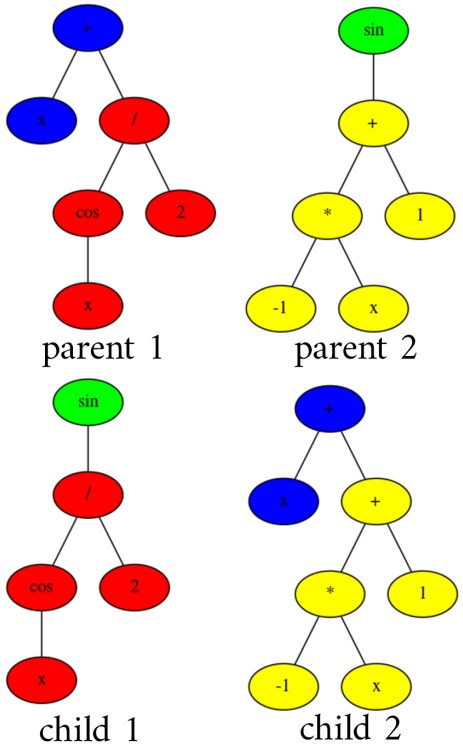


Figure 2.2: Example of crossover

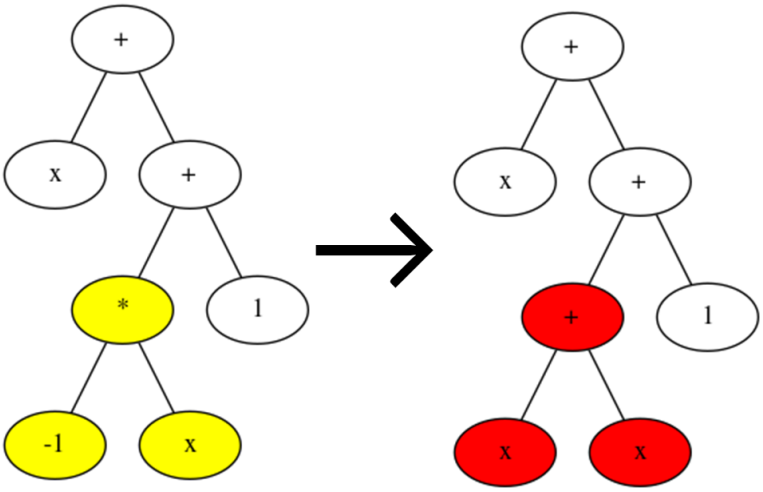


Figure 2.3: Example of mutation by replacing subtree with a randomly generated tree

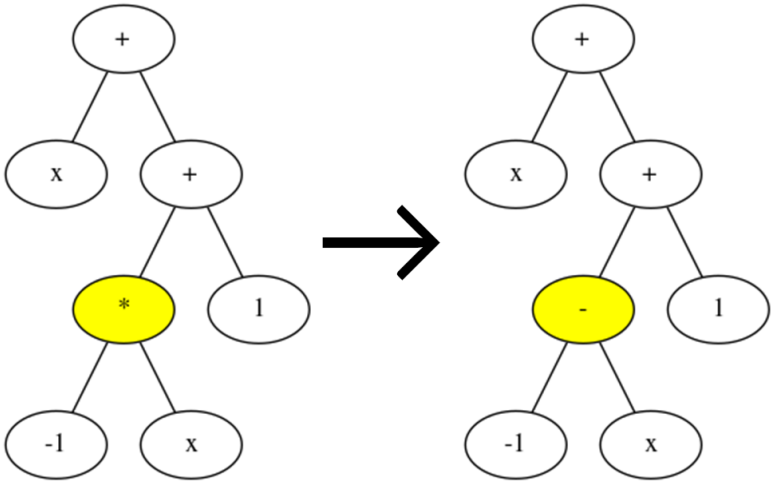


Figure 2.4: Example of mutation by changing one node

2.4 Symbolic Regression

Symbolic regression is the problem of identifying the analytical mathematical description of a hidden system from experimental data [1], [5]. In other words, given a set of inputs and corresponding expected outputs, the problem is to find an expression that only consists of symbols from given *primitive set* and for given inputs yields values that are as close as possible to the expected outputs.

Unlike polynomial regression or related machine learning data fitting methods, it does not need any prior knowledge about the distribution of the data.

2.4.1 Solving Symbolic Regression Using GP

The problem of symbolic regression described above can be solved using GP. The implementation is quite straightforward since mathematical expressions are very easily represented by tree structures (again, the example in figure 2.5). We define the training dataset as

$$T = \{(\boldsymbol{x_i}, y_i) \mid x_i \in \mathbb{R}^n, y_i \in \mathbb{R}, i = 1, \ldots, N\},$$

where $\boldsymbol{x_i}$ is i-th input, y_i is i-th output, n is the dimension of the input space and N is the number of test cases in the training dataset.

The only thing that needs to be chosen is the fitness function. The most common ones are mean square error, mean absolute error, or hit rate. In our experiments, mean absolute error was used:

$$\text{fitness(s)} = \frac{1}{N} \sum_{i=1}^{N} |s(\boldsymbol{x_i}) - y_i|$$

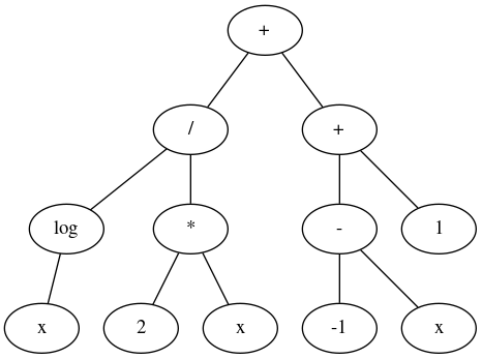


Figure 2.5: Example of expression $\frac{\log x}{2x} + (-1 - x) + 1$ being represented by a binary tree

where $s(\boldsymbol{x_i})$ is the value of candidate on i-th input in training data. Even though the implicit goal of the evolution is to find the solution with the maximum fitness, here the goal is naturally to minimize the fitness.

Chapter 3

Fitness Predictors

As we mentioned earlier, evaluating all individuals on all training dataset points can be very computationally demanding because thousands of individuals have to be evaluated on many test cases during the evolution. That is why it would be beneficial to get similar results using just a subset of the training dataset. The used subsets are called fitness predictors. Using the predictors results in a slight change in the evolution goal. Instead of a mean absolute error on all training data shown in the previous chapter (objective fitness), we compute the values of the expressions in points of the current predictor (predicted fitness). The optimization task is:

$$s^* = \arg\min_{s \in S} p(s)$$

where p is the fitness predictor used in current generation and $p(s)$ is predicted fitness of an individual s using predictor p. The problem is that maximizing the predicted fitness does not guarantee to find the solution with the best objective fitness. For this reason, it is required to find predictors, that predict the fitness close to the objective fitness.

We will often measure the effort made by the GP by the number of point evaluations. It represents a number of times some individual was tested on a test case. For example if we perform 10 generations, in each one we evaluate 128 individuals using a predictor of size 32, we used a total of $10 \times 128 \times 32 = 40960$ point evaluations. Measuring the effort by the number of generations would not be fair because generations using a predictor of smaller size use fewer point evaluations and thus take less time.

In the following sections, we describe two known methods that solve this problem using coevolution and introduce our method.

3.1 Coevolution of Fitness Predictors

In a coevolutionary algorithm, the fitness metric for one individual becomes a function of other individuals, possibly including itself. More precisely, one individual can affect the relative fitness ranking between two other individuals in the same or a separate population[6].

In case of fitness predictors coevolution, predictors from predictor population affect fitness ranking in the solution population and the other way round. The aim of coevolving fitness predictors is to allow both solutions and fitness predictors to enhance each other automatically until an optimal problem solution is found [10]. More specifically, the predictors help the solutions to evolve more quickly, and the solutions help the predictors to get more accurate. The exact methods will be described in the following subsections.

3.1.1 Approach by Michal D. Schmidt and Hod Lipson

In this subsection, we talk about the method introduced in [10] by Michal D. Schmidt and Hod Lipson. This method constructs and selects the fitness predictors using the coevolution of solutions and fitness predictors. During evolution, there are three populations present in the algorithm:

1. Solutions - The main population of candidate solutions, the best individual is the one with the best predicted fitness

$$s^* = \arg\min_{s \in S} p_{best}(s),$$

where S is the set of all solutions, p_{best} is the highest-ranked predictor in the current generation, and $p_{best}(s)$ is the predicted fitness of solution s using predictor p_{best}.

2. Trainers[1] - Selected solutions used for training the fitness predictors, the best trainer is the one with the highest variance of predicted fitness among the current generation of predictors.

$$t^* = \arg\max_{s \in S_{cur}} \frac{1}{|P_{cur}|} \sum_{p \in P_{cur}} (p(s) - \overline{p(s)})^2,$$

where S_{cur} is the current solution population, P_{cur} is the current predictor population and $\overline{p(s)}$ is the average predicted fitness of solution s among the predictors from P_{cur}.

3. Predictors - Population of the fitness predictors, each generation of the main population, the best predictor from this population is chosen and used to evaluate the solutions. The predictors are represented by constant-size arrays of pointers to test cases in the training dataset. The best predictor is the one that can predict the fitness of the individuals in the current trainer population the most accurately:

$$p^* = \arg\min_{p \in P} \frac{1}{|T_{cur}|} \sum_{t \in T_{cur}} |fitness(t) - p(t)|,$$

where P is a set of all possible fitness predictors, and T_{cur} is the current population of trainers.

[1] Trainers are an archive or a set, rather than a population because the particular trainers do not evolve, while solutions and predictors do. The reason why we listed them as a population is because authors of the paper did so.

The solutions are evolved using standard GP, and the predictors are evolved using standard GA (as they are not represented by trees). At the beginning of the algorithm, all populations are randomized. It is important that the evolution of predictors does not take too much effort (where effort is measured in point evaluations); otherwise, there would not be any advantage of using them. In [10] the threshold is set to 5% all computation effort measured in point evaluations. Every 100 predictor generations, a new trainer is chosen from the solution population using the above criteria and replaces the oldest trainer. Furthermore, the solution evolution and the predictor evolution are run in separate threads. Pseudocode 2 ilustrates the connection between solutions and predictors during the coevolution.

We will refer to this method as CSP (constant-size predictors).

```
1  Begin Solutions Thread
2      Randomize solution population
3      loop
4          Apply genetic operations on solutions with specified
            probabilities
5          pred ← top-ranked fitness predictor
6          Evaluate the population using pred
7          Perform selection
8          if convergence criterion is met then
9              return top-ranked solution
10         end
11     end
12 end
13
14
15 Begin Predictors Thread
16     Randomize predictor population
17     Randomize trainer population
18     loop
19         if computational effort > 5% then
20             wait
21         end
22         Apply genetic operations on predictors with specified
            probabilities
23         Evaluate predictors using trainers
24         Perform selection
25         if Time to add a new trainer then
26             Compute variances in fitness predictions for all solutions in
                the current population
27             Replace the oldest trainer with a copy of solution with the
                highest variance
28             Calculate the exact fitness of the new trainer
29         end
30     end
31 end
```

Algorithm 2: Pseudocode of coevolution of solutions and fitness predictors

3.1.2 Approach by Michaela Šikulová, Lukáš Sekanina and Michal Wiglasz

The disadvantage of the approach introduced in [10] is that the predictors are of fixed size. Many experiments have to be therefore conducted to find the optimal size to use in each task. [3] introduces adaptive-size fitness predictors. These predictors change their size depending on the *phase of evolution*.

The *phase of evolution* is determined using the evolution speed v, which is computed as

$$v = \frac{\Delta f}{\Delta G},$$

where Δf is the difference of objective fitnesses of the current best solution and best solution last time the size was updated. ΔG is the number of generations since the last size update.

To determine if over-fitting is an issue in the evolution run, inaccuracy I is computed as a ratio of subjective and objective fitness of the current best solution:

$$I = \frac{f_{sub}}{f_{obj}},$$

where f_{sub} is fitness computed on test cases from the current best predictor, and f_{obj} is fitness on all training dataset cases.

Using the following rules, the size of the predictors is updated to ensure accurate fitness prediction and progress of the evolution:

- Increase the predictor size (multiply the size by a constant $C > 1$) if:
  1. Current predictions tend to over-fit ($I > I_{thr}$) or
  2. Objective fitness of the best individual in the population is increasing ($v > 0$)
- Decrease the predictor size if (multiply the size by a constant $C < 1$):
  1. Objective fitness of the best individual in the population is stagnating or decreasing ($v \approx 0$ or $v < 0$)

Constants I_{thr} and C (for each rule) must be found. Even though the predictors are adaptive-size, they are implemented as constant-size arrays with the length of the training set. When reading the test cases of the predictor, only the first unique *read_length* test cases are read; the remaining test cases are ignored. This way, the size can simply be increased or decreased by increasing or decreasing the *read_length* variable.

Notice that these rules assume that the fitness should be maximized. If the goal is to find a solution with minimal fitness (which is our case), these rules have to be modified. To be specific, over-fitting is signaled by $I < I_{thr}$ and improvement is signaled by $v < 0$.

Another difference from the method introduced in the previous section is how new trainers are selected. Here a solution replaces the oldest trainer if

its subjective fitness is better than the subjective fitness of the top-ranked trainer.

We will refer to this method as *ASP* (adaptive-size predictors).

3.2 Adaptive-size deterministic predictors

In this section, we propose our method of constructing the predictors. It does not use coevolution. Instead, it works with only one predictor, whose test cases and size are deterministically changed during the evolution. Deterministically in a sense, that given the same population and state of the predictor, the next state of the predictor will always be the same. This is the opposite of coevolved predictors, that use the randomness in the genetic operations and selection. In the following subsections, we will thoroughly explain how the method works. We will refer to this method as *DP* (deterministic predictors).

3.2.1 Score of a test case

Our method works with a quantity called a *score* of a test case. Suppose we have a training set T and a test case $t \in T$. Let $\boldsymbol{x}$ and y be the input(s) and the output of the test case t respectively. Then the *score* of the test case t is

$$\text{score}(t) = \frac{\sum_{s \in S_c} (|s(\boldsymbol{x}) - y| \cdot \text{fitness}(s)^{-1})}{\sum_{s \in S_c} \text{fitness}(s)^{-1}},$$

where S_c is current population of solutions and $\text{fitness}(s)$ is objective fitness of solution s. In other words, it is a weighted arithmetic mean of absolute errors of the current population on test case t with weights set to inverted values of objective fitnesses of the population.

To decide which case should be chosen to predict the fitness more accurately, we can compare *score* of all the test cases.

We decided to use the mean of absolute errors to compute the *score* because using fitness predictors often runs into the problem of overfitting. This way if a solution's predicted fitness is better than its objective fitness, a test case on which this solution performs poorly, has a higher chance to be included in the next predictor. There is no particular reason not to use some other function than the absolute errors. The only constraint is that the value of the function must quantify how badly the particular solution solves the particular test case. This means that this method, just like the methods described earlier, is not at all limited to be used to solve the symbolic regression. Alternative functions that could be used are for example squared errors or the number of times the robot hits a wall in a simulated maze.

The reason for weights to be set to inverted values of objective fitnesses is that this way, the absolute errors of better solutions are considered more important. Therefore solutions that already perform well can be tuned to perform even better, while solutions that perform poorly have less impact on the test case's *score*.

It is important to note, that this formula assumes that the objective is to minimize the fitness and that the fitness is always positive. If fitness should be maximized, for weights, we would use the values of the fitness, not the inverted values. If a solution with fitness equal to zero is found, it is an ideal solution, and the run is terminated before this formula is evaluated.

Every further use of the term *score* in the context of test cases refers to the score computed by the above formula.

3.2.2 Size of the predictors

As was already mentioned, our method changes the size of the predictor during evolution and tries to find the most suitable one. The procedure of determining the size described in 3.1.2 was used without any modifications, as this method proved to be performing well.

After updating the size of the predictor, two scenarios can occur:

1. Size has to be decreased - if the size has to be decreased by n test cases, we simply remove the last n test cases in the predictor. It would seem natural to remove n test cases with the lowest *score* instead. To do it, however, the score of all test cases would have to be computed. Moreover, the reason to decrease the size of the predictor is to help the evolution leave the local optimum, in which case accuracy of the predictor is not important.

2. Size has to be increased - if the size has to be increased by n test cases, we compute the scores (introduced in 3.2.1) of all test cases in the training set and add n test cases with highest *score*, that are not already present in the predictor.

3.2.3 Periodic update

The predictor is updated periodically. Every T solution generations, we compute the *scores* of all test cases. Then we replace the test case from the predictor with the lowest *score* by the test case with the highest *score*, which is not already present in the predictor. In our experiments we set $T = 100$ generations.

3.3 Random predictors

To justify using sophisticated methods for constructing the fitness predictors, it is important to compare their performance with the most simple ones as well. For this purpose, we use two types of random predictors:

1. Random static - random test cases are selected at the beginning of the run and used for evaluations for the rest of the run. We will refer to them as RP_{static}.

2. Random dynamic - random test cases are selected at the beginning of each generation and only used for the evaluation in that generation. We will refer to them as $RP_{dynamic}$

Both static and dynamic random predictors have a constant size.

Chapter 4

Experiments

In this chapter we compare performance of GP using all test case data (GP_{std}), constant-size coevolving predictors (CSP), adaptive-size coevolving predictors (ASP), random predictors (RP_{static}, $RP_{dynamic}$) and the proposed method (DP). The methods will be evaluated using symbolic regression.

4.1 Implementation

To implement GP solving symbolic regression, we used DEAP (Distributed Evolutionary Algorithms in Python) framework [2] with tools necessary to make GP work, such as tree structures and genetic operations. We implemented the fitness predictors themselves as 1-D `numpy` arrays of `int` type. They contain indexes of test cases that have to be used. Since our data is also stored as a `numpy` array, to use only the cases from a given predictor, we can conveniently use it as an index: `training_set[predictor]`.

To take care of inner mechanisms of fitness predictor construction, `FitnessPredictorManager` class is used. Every method of predictor construction is implemented in a separate class derived from this class. The `get_best_predictor` method is called by the GP run every generation, and it returns the predictor that will be used to evaluate the population of that generation. The `next_generation` method is also called every generation, and its purpose is to allow the predictor manager to update its inner state if needed. It does not return anything, and its keyword arguments contain potentially useful information about the state of the evolution of the solutions, such as the current generation, current population, etc. The whole program runs in one thread. This means that the main evolution has to wait for this method to return.

4.2 Setup

There are many parameters to set that influence the behaviour of GP. In table 4.1 we sum up all the parameters and settings with the corresponding values we used in our experiments.

Table 4.2 contains rules we used to find the constant C to update the size

population size	128
mutation probability	0.1
crossover probability	1.0
function set	{+, -, *, /, sin, cos, exp, log}
terminal set	{inputs, 0, ±1, π}
selection	deterministic crowding
initial population	ramped half-and-half
mutation	uniform mutation (Fig 2.4)
tree height limit	8

Table 4.1: Used GP parameters

condition	C
$I < 0.57$	1.2
$\|v\| \leq 0.001$	0.9
$v > 0$	0.96
$-0.1 \leq v < 0$	1.07
$v < -0.1$	1

Table 4.2: Rules to update the size

of the predictors (for 3.1.2 and 3.2) in the order they were applied. They are the rules used in [3] adjusted to work with our fitness function. The initial size for adaptive-size predictors was set to 5. The size of constant-size predictors was set to 32.

The distance function between two individuals used in deterministic crowding selection was defined as euclidean distance between their error vectors on the testing data:

$$distance(s_1, s_2) = \sqrt{\sum_{i=1}^{N} ((s_1(\boldsymbol{x_i}) - y_i) - (s_2(\boldsymbol{x_i}) - y_i))^2} = \sqrt{\sum_{i=1}^{N} (s_1(\boldsymbol{x_i}) - s_2(\boldsymbol{x_i}))^2},$$

where $\boldsymbol{x_i}$ and y_i are inputs and outputs used to evaluate the individuals - either the entire training set or some subset of it.

4.3 Used data

In the following experiments, we compare methods for finding fitness predictors on several datasets. First five datasets are created using functions from [10] and [3]. The sixth dataset is obtained from UCI Machine Learning Repository [4]. The data contains 6 parameters of the yachts hull and the value to be predicted is yachts residuary resistance. These are the 5 functions we sampled to get our datasets:

$$
\begin{aligned}
f_1(x) &= 1.5x^2 - x^3 && x \in [-5, 5] \\
f_2(x) &= e^{|x|}\sin(2\pi x) && x \in [-3, 3] \\
f_3(x) &= x^2 e^{\sin(x)} + x + \sin(\frac{\pi}{4} - x^3) && x \in [-10, 10] \\
f_4(x) &= e^{-x}x^3\sin(x)\cos(x)(\sin^2(x)\cos(x) - 1) && x \in [0, 10] \\
f_5(x) &= \frac{10}{(x-3)^2 + 5} && x \in [-2, 8]
\end{aligned}
$$

The first three functions were used in [10]. It is important to note that the function f_2 is not exactly the same as the one described in the article. However, because our modified version[1] corresponded to the plots in the article whereas the original one did not, we think that the authors simply made a mistake. Functions f_4 and f_5 were used in [3]. Plots of the used functions can be seen in the figure 4.1.

Our training datasets consist of 200 equally distributed points in presented intervals. The testing datasets contain all these points and 200 additional points randomly selected from the same intervals. This is the way the datasets were created in [10]. The sixth dataset was randomly divided into training and testing data in the ratio of 7:3.

4.4 Measuring potential speed benefits from using fewer point evaluations

The main advantage of using fitness predictors is that every generation of solutions uses fewer point evaluations. Speedup with respect to point evaluations can be thus computed very easily as a ratio between training set size and the predictor size[2]. Speedup with respect to real-time depends on the particular implementation because different tasks may be faster in different implementations, often written in different programming languages. To see the relationship between point evaluations and time in our implementation, we conducted the following experiment:

We ran 1000 generations of evolution and measured how much time they took. Each generation we used 1, 50, 100, 150 and finally 200 test cases to evaluate the solutions. Average values and best fitting linear function can be seen in the figure 4.2. From the figure, it can be seen that the relationship between point evaluations and time is linear, which was expected. Therefore, to compute how much time performing N generations, each using X points of training set will take, we can use the parameters of the linear function:

$$T(N, X) = \frac{N}{1000}(26.4 + 0.64X)\text{sec.}$$

[1] We changed $e^{|x|}\sin(x)$ to $e^{|x|}\sin(2\pi x)$.

[2] for fixed-size predictors

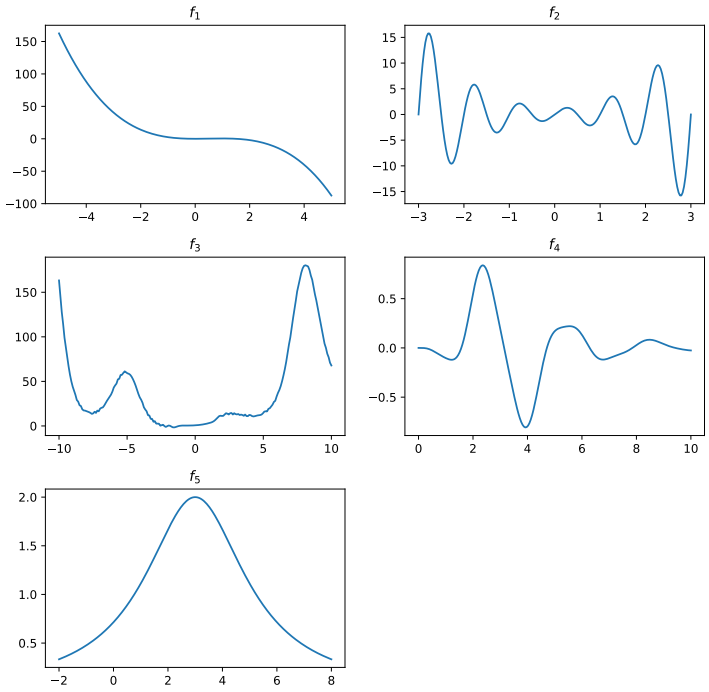


Figure 4.1: Sampled functions

assuming that every generation takes approximately the same amount of time.

An ideal fitness prediction would predict the fitness exactly and use no time to construct the predictor. This method would need the same number of generations as GP that uses all training data and achieve the speedup computed using the above formula. GP using a fitness prediction method that is not this precise should, in theory, achieve smaller speedup, since it will probably need to run for more generations to solve the problem. On the other hand, using the predictor might help the GP in some other ways, for example leaving local optima by allowing objectively worse solutions in the population. In practice, we observed both variants.

Note that speedup with respect to point evaluations does not necessarily imply real-time speedup. That is because a method that uses fewer point evaluations in each generation will naturally manage to perform more generations using the same number of point evaluations. Because of that, it will spend more time doing the tasks linked to the generational process. Figure 4.3 shows how much time it takes to perform 10^7 point evaluations using a various number of test cases each generation. Just like in figure 4.2, these values depend on particular implementation.

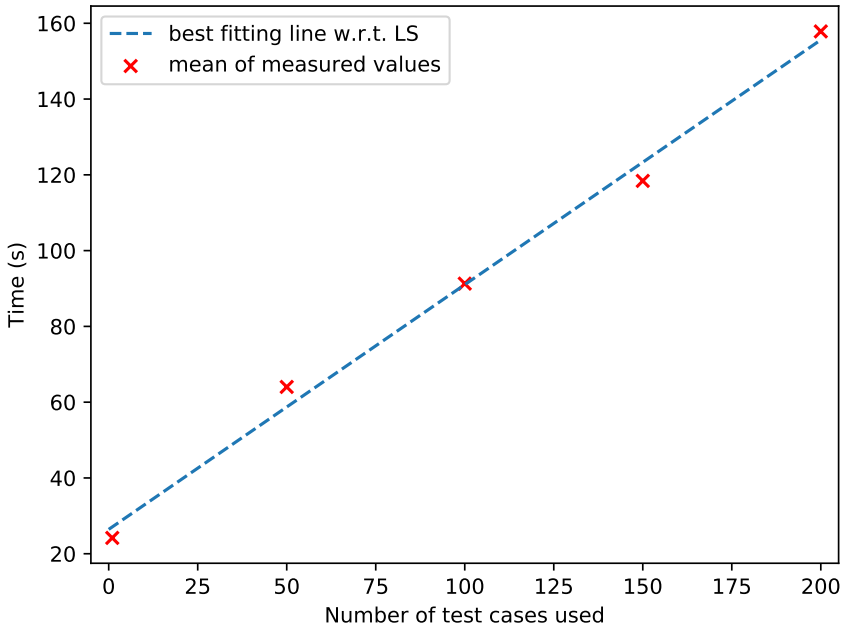


Figure 4.2: Time needed to reach 1000 generations

4.5 Comparison of Time Spent Constructing the Predictors

Even though fitness predictors can potentially speed up the evolution of solutions, constructing the predictors takes some time. In some cases (such as construction using coevolution) this problem can be solved by parallelization. Because our implementation only uses a single thread, it is useful to measure how much time the methods spend constructing the predictors. We ran 1000 generations of evolution and measured the time of GP_{std}, ASP, CSP and DP. In these modified runs, predictors were constructed, but the entire training set was used. Because CSP performs predictor generations based on total effort, we counted the point evaluations as if the predictors were used. This ensured, that the predictors behaved exactly as they would behave if they were used. Figure 4.4 shows how much additional time was needed while using ASP, CSP and DP to reach 1000 solutions generations. We can see, that coevolved predictors require around 6-7% and DP requires around 4% more time. The reason, why DP takes less time constructing the predictors is, that it does not use coevolution. Difference of times between CSP and ASP is most probably caused by their different approaches to solutions-predictors synchronization. (CSP performs a new generation of predictors based on total effort, ASP every T generations of solutions.)

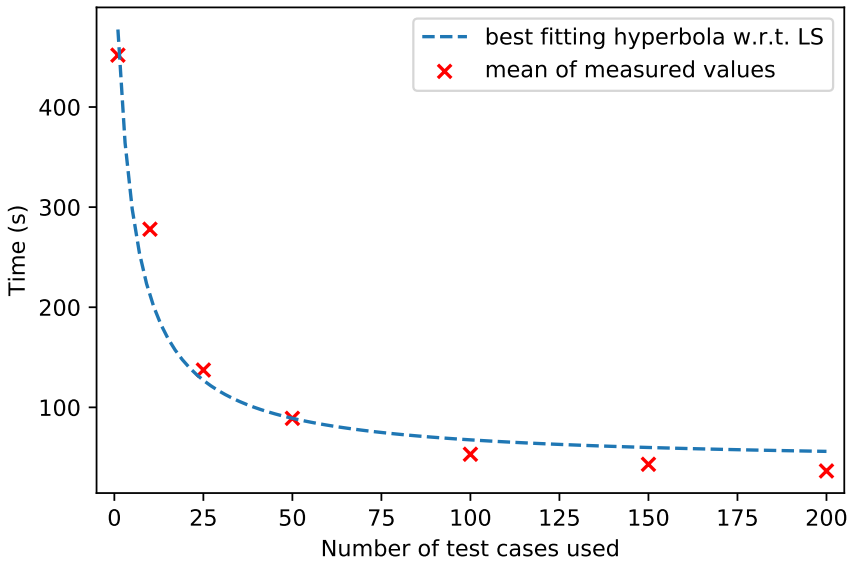


Figure 4.3: Time needed to perform 10^7 point evaluations

4.6 Comparison of Performance During the Evolution

As mentioned earlier, the main advantage of using fitness predictors is to get better results while making less computational effort. We compare how well different methods of finding fitness predictors perform all given the same effort. The effort made is measured in point evaluations as they were defined in chapter 3.

In this experiment, we are interested not only in the final results of the evolution but also in the quality of the solutions in the population during the evolution. We set an effort limit to 10^7 point evaluations and during the evolution logged fitness of the best individual on the test set.

In figure 4.5, we can see the test set fitnesses averaged over 50 runs. These plots do not show any valuable information because arithmetic mean is not robust, and one value can change the result arbitrarily. In our case few runs, mainly from ASP, suffered from over-fitting during evolution. We only show this plot for the sake of completeness. In figure 4.6, we show the same plot, but we compute the mean, excluding 10% worst (highest) values. This time we can see some meaningful results. On f_2, f_4 and f_5 GP_{std} found noticeably worse solutions than other methods. On the rest of the datasets, performance at the end of the runs was comparable, but it can be seen that the fitness was converging much slower than other methods. For example, on f_1, DP found a solution with similar fitness using more than 10 times less effort (note that the scale of the x-axis is logarithmic). On f_2, f_3 and f_4, we can see that at the beginning ASP and DP were performing noticeably worse than other

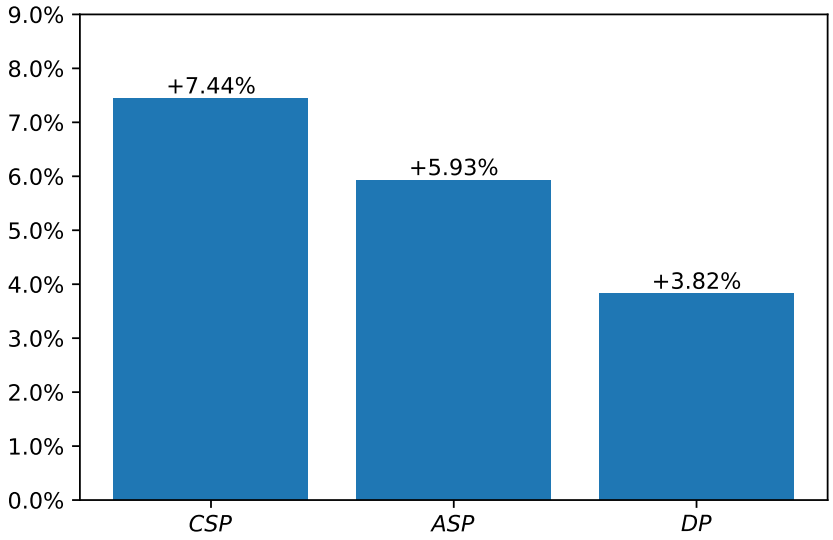


Figure 4.4: Increase of time needed to reach 1000 generations compared to GP_{std}

methods and then caught up later. We think that this happens because both ASP and DP use the same method for finding the predictor size, it took them some time to find the suitable size, and in the process of finding it, the test set fitness suffered. After the suitable size was found, ASP and DP caught up and overtook GP_{std}.

The figure 4.7 shows medians of the test set fitnesses during the evolution. The results are very similar to 4.6, therefore we do not discuss them any further.

Figure 4.8 shows histograms of final test set fitnesses. For the same reason as in figure 4.5, we ignored 10% worst solutions. Again, we can see, that GP_{std} generally produces worse solution than GP using fitness predictors, except for yachts dataset, where it performs comparably well. As for fitness predictors, the histograms show, what was expected from figures 4.6 and 4.7: Generally, ASP and CSP performed similarly well. DP performed slightly worse and except for f_3 better than random predictors. From the histograms it seems, that RP_{static} performs better than $RP_{dynamic}$.

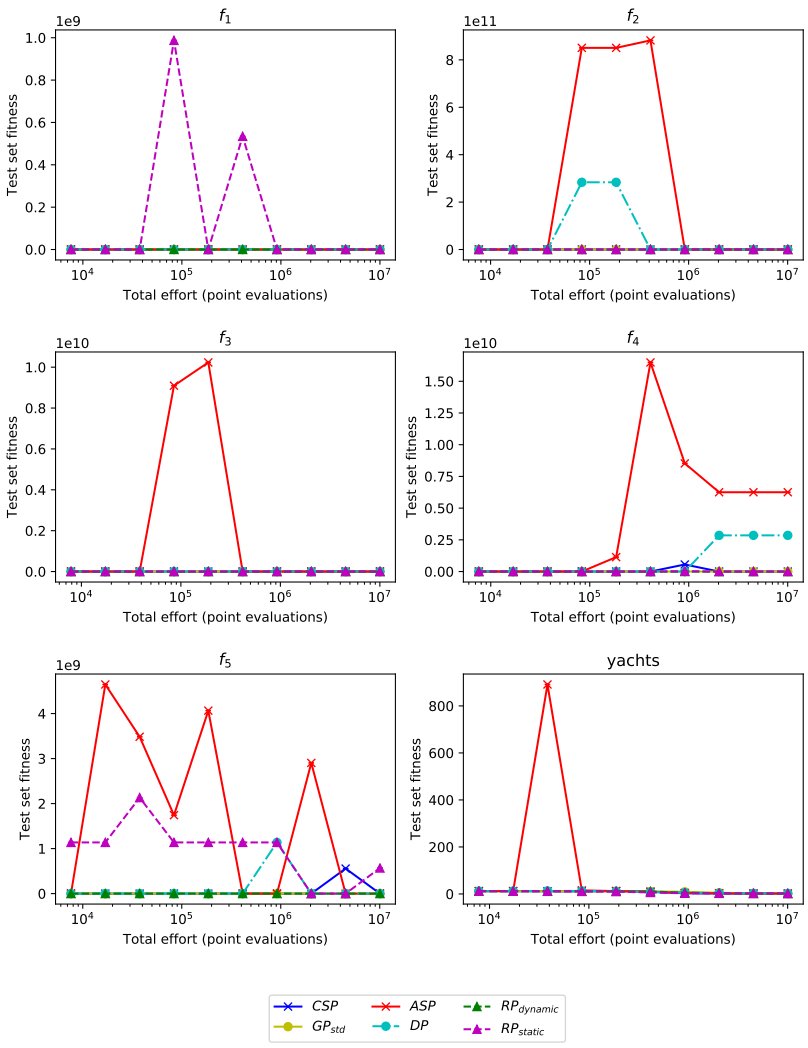


Figure 4.5: Mean test set fitness during the evolution

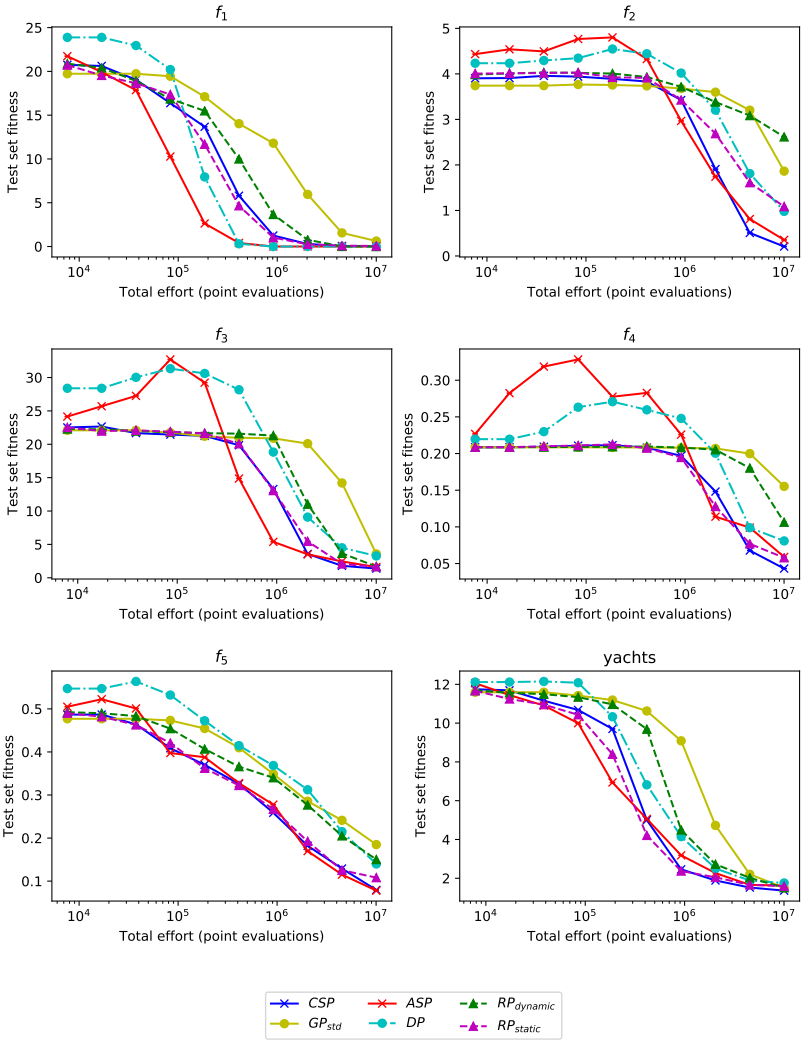


Figure 4.6: Mean test set fitness during the evolution ignoring 10% worst values

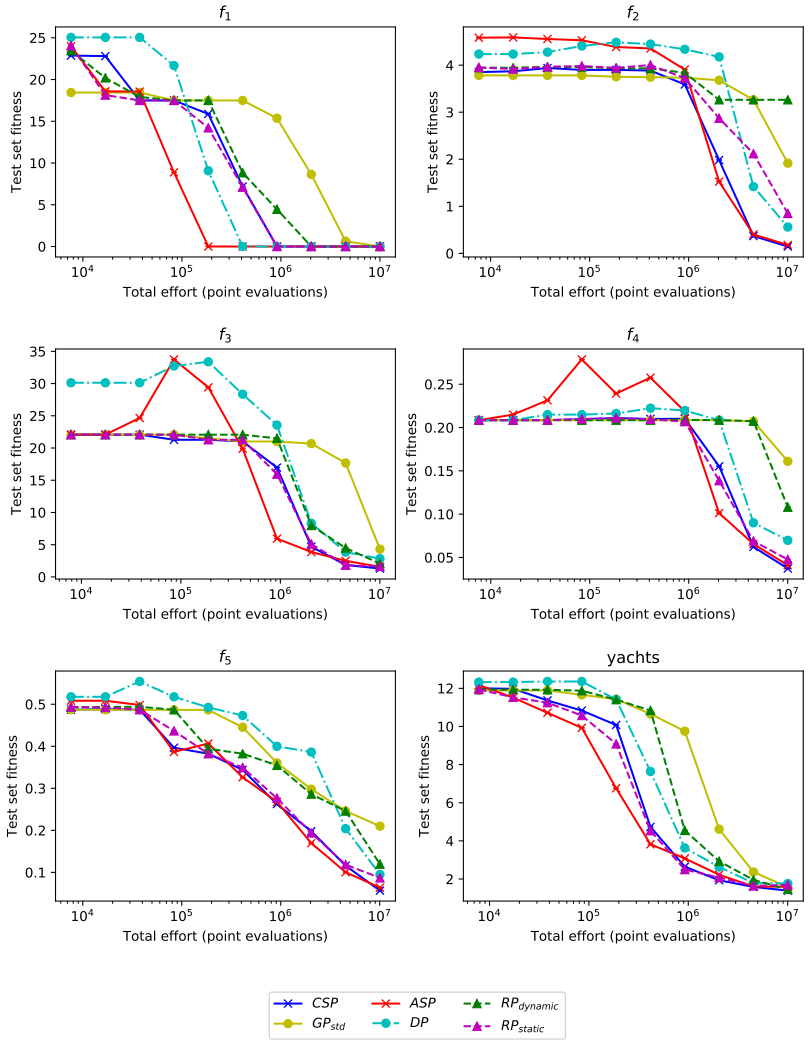


Figure 4.7: Median test set fitness during the evolution

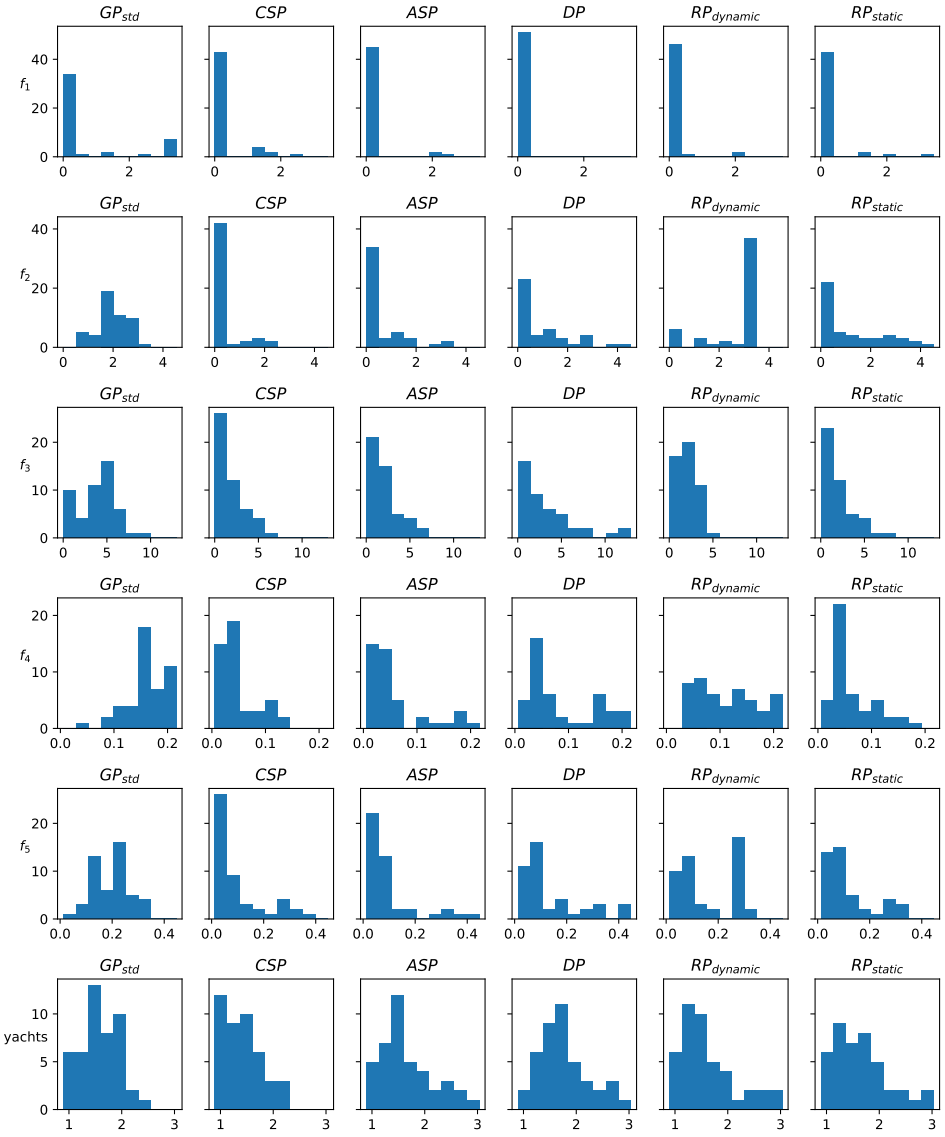


Figure 4.8: Histograms of achieved fitnesses

4.7 Comparison of Expected Cost to Converge

In the previous experiment, we ran the evolution with an effort limit, with no convergence criterion on the quality of the solutions. In this experiment, we ran the evolution, until a solution with test set fitness better than F was found, or until T seconds elapsed. Again, we performed 50 independent runs. We set $T = 300$ and F to values shown in table 4.3. The values of F were set so that the problems are solvable in given time limit, but at the same time competitive.

dataset	$\boldsymbol{F}$
f_1	0.001
f_2	0.5
f_3	1.5
f_4	0.025
f_5	0.025
yachts	1.5

Table 4.3: Convergence thresholds for each dataset

While statistics of successful runs are important, they can be misleading because not all runs converge, and in practice, we need to run multiple runs to get a successful one. To know what will be the cost to get such successful run using a particular method, the expected cost must be computed.

4.7.1 Expected Cost

Let us say that our method has a probability p to converge before the time runs out. Let X be the number of times we run the method until it converges for the first time. It can be proved, that $EX = \frac{1}{p}$. If C is a random variable, whose value is the total cost needed to get a successful run, EC_s is expected cost of a successful run and EC_f is expected cost of a failed run, then the expected cost to converge is

$$EC = (EX - 1)EC_f + EC_s.$$

This equation holds true because the expected value is a linear operator. Because the runs are independent, the number of successful runs follows the binomial distribution, therefore, using maximum likelihood estimate, we can estimate p as $\frac{S}{N}$, where S is the number of observed successful and N is the total number of runs (success rate). Moreover, if we assume that C_s and C_f are distributed normally, we can estimate their expected values as the means of observed values.

4.7.2 Results

The results of the experiment are summarized in table 4.4. Best values are in bold. On each dataset, we measure:

1. success rate - the percentage of successful runs
2. mean time to converge - mean time used by successful runs
3. mean point evaluations to converge - mean point evaluations used by successful runs
4. expected time to converge - expected time to get a successful run
5. expected point evaluations to converge - expected number of point evaluations to get a successful run

We also normalized expected time and point evaluations to $(0, 1]$ by dividing the best value by corresponding value so that the best performing method has the highest score of 1. The normalized values are in brackets.

Our method DP performed best on functions with simple shape f_1 and f_5. On f_2 and f_3 it performed even worse than GP_{std}. On f_4 it performed better than GP_{std} but worse than coevolved predictors (CSP and ASP).

Very interesting are the results from the yachts dataset. Here, the best solutions by far were produced by GP_{std}, having 100% success rate, while fixed size predictors CSP, RP_{static} and $RP_{dynamic}$ only having 17%, 20% and 39% respectively. Adaptive-size predictors ASP and DP achieved success rate of 86% and 57%, which is better than all fixed-size predictors, but still worse than simple GP_{std}. It seems that on this dataset, the fitness of an individual strongly depends on each test case. This may be because the data comes from the real measurements and is multidimensional, both facts contributing to higher distances between the input vectors. Because of that, fixed size predictors performed poorly. Even though adaptive-size predictors are able to increase their size to tackle this problem, it takes time, and that may have been the reason they achieved lower success rates.

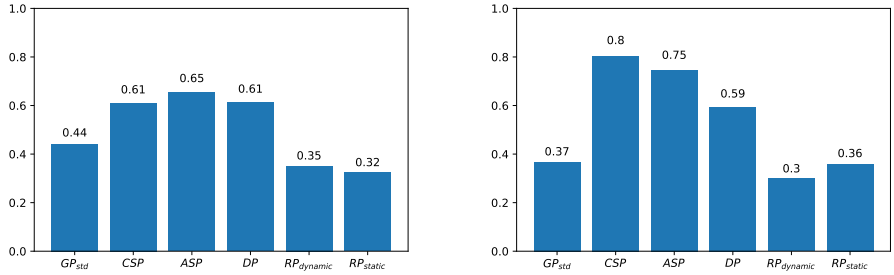
If we take a look at expected point evaluations to converge, except for f_1, where all methods were dominated by DP, ASP is the most efficient method.

Figures 4.9 and 4.10 show means and medians of normalized expected times and numbers of point evaluations to converge. We can see that ranking changes if we compute medians instead of the means. That is because of the dominating performance of DP on f_1 and poor performance of CSP on yachts dataset.

4.7.3 Behaviour of Predictors

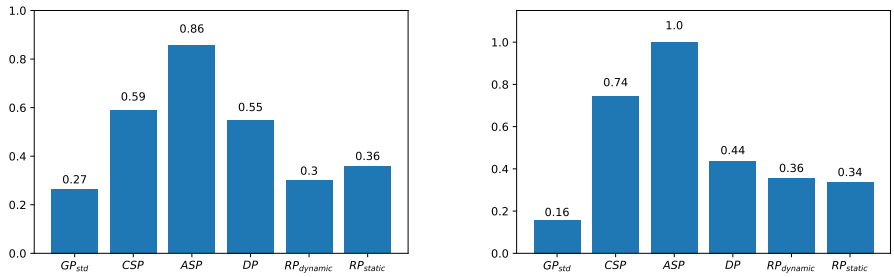
Figure 3.2.1 shows how differently DP and ASP choose which test cases will be used in the predictors. From the histograms, it is very clear that DP prefers points near peaks and valleys and endpoints of the fitted interval. Even though the primary objective of DP is to use the test cases which the solutions struggle to fit, it seems that these particular points lie close to the peaks and valleys. ASP's choice of points seems more uniform, but we can still see that some points get selected more often than the others.

Thanks to the results from the previous experiments, we can conclude that even though exclusively choosing points from peaks and valleys may seem like



(a) : mean **(b) :** median

Figure 4.9: Mean and median normalized expected times to converge



(a) : mean **(b) :** median

Figure 4.10: Mean and median normalized expected numbers of point evaluations to converge

a reasonable approach, it only works well on very simple-shaped functions and on more complex functions it performs considerably worse than coevolved predictors.

		f1	f2	f3	f4	f5	yachts
success rate	GP_{std}	88%	57%	78%	18%	4%	**100%**
	CSP	88%	**76%**	78%	46%	20%	17%
	ASP	94%	67%	**86%**	**52%**	17%	86%
	DP	**100%**	45%	71%	33%	**25%**	57%
	RP_{static}	82%	45%	53%	22%	19%	20%
	$RP_{dynamic}$	92%	31%	78%	39%	11%	39%
mean time to converge (seconds)	GP_{std}	22.95	112.52	68.90	198.72	191.84	**43.83**
	CSP	6.68	55.04	54.68	**132.01**	166.90	50.66
	ASP	10.72	**24.73**	63.54	193.31	144.02	76.80
	DP	**2.48**	60.21	71.33	172.27	164.67	78.03
	RP_{static}	5.86	52.82	**43.61**	146.75	134.43	44.36
	$RP_{dynamic}$	8.01	91.43	87.43	191.97	**120.23**	120.01
mean point evaluations to converge	GP_{std}	$5.01 \cdot 10^6$	$2.31 \cdot 10^7$	$1.28 \cdot 10^7$	$3.37 \cdot 10^7$	$4.28 \cdot 10^7$	$0.86 \cdot 10^7$
	CSP	$0.82 \cdot 10^6$	$0.50 \cdot 10^7$	$0.45 \cdot 10^7$	$\mathbf{1.04 \cdot 10^7}$	$1.56 \cdot 10^7$	$0.18 \cdot 10^7$
	ASP	$0.88 \cdot 10^6$	$\mathbf{0.18 \cdot 10^7}$	$0.39 \cdot 10^7$	$1.25 \cdot 10^7$	$\mathbf{0.85 \cdot 10^7}$	$0.51 \cdot 10^7$
	DP	$\mathbf{0.26 \cdot 10^6}$	$0.59 \cdot 10^7$	$0.58 \cdot 10^7$	$1.59 \cdot 10^7$	$2.13 \cdot 10^7$	$0.47 \cdot 10^7$
	RP_{static}	$0.72 \cdot 10^6$	$0.45 \cdot 10^7$	$\mathbf{0.38 \cdot 10^7}$	$1.10 \cdot 10^7$	$1.28 \cdot 10^7$	$\mathbf{0.13 \cdot 10^7}$
	$RP_{dynamic}$	$1.23 \cdot 10^6$	$1.34 \cdot 10^7$	$0.88 \cdot 10^7$	$1.99 \cdot 10^7$	$1.52 \cdot 10^7$	$0.46 \cdot 10^7$
expected time to converge (seconds)	GP_{std}	63.86 (0.04)	340.11 (0.43)	155.74 (0.72)	1565.39 (0.30)	7391.84 (0.14)	**3.83 (1.0)**
	CSP	47.59 (0.05)	**147.35 (1.00)**	137.18 (0.82)	484.18 (0.97)	1366.90 (0.78)	1483.99 (0.03)
	ASP	29.87 (0.08)	174.73 (0.84)	**113.54 (1.00)**	**470.24 (1.0)**	1644.02 (0.64)	124.53 (0.35)
	DP	**2.48 (1.00)**	428.40 (0.34)	191.33 (0.59)	791.02 (0.59)	**1064.67 (1.0)**	303.03 (0.14)
	RP_{static}	71.71 (0.03)	418.04 (0.36)	308.99 (0.36)	1183.11 (0.39)	1401.10 (0.76)	1244.36 (0.04)
	$RP_{dynamic}$	34.09 (0.07)	747.68 (0.65)	174.28 (0.65)	665.65 (0.70)	2640.23 (0.40)	593.70 (0.07)
expected point evaluations to converge	GP_{std}	$1.05 \cdot 10^7$ (0.02)	$5.91 \cdot 10^7$ (0.19)	$2.50 \cdot 10^7$ (0.27)	$23.14 \cdot 10^7$ (0.12)	$154.52 \cdot 10^7$ (0.06)	$0.86 \cdot 10^7$ (0.91)
	CSP	$0.36 \cdot 10^7$ (0.07)	$1.16 \cdot 10^7$ (0.97)	$1.00 \cdot 10^7$ (0.69)	$3.47 \cdot 10^7$ (0.82)	$12.38 \cdot 10^7$ (0.79)	$4.39 \cdot 10^7$ (0.17)
	ASP	$0.19 \cdot 10^7$ (0.13)	$\mathbf{1.13 \cdot 10^7}$ **(1.00)**	$\mathbf{0.69 \cdot 10^7}$ **(1.00)**	$\mathbf{2.87 \cdot 10^7}$ **(1.00)**	$\mathbf{9.85 \cdot 10^7}$ **(1.00)**	$\mathbf{0.78 \cdot 10^7}$ **(1.00)**
	DP	$\mathbf{0.03 \cdot 10^7}$ **(1.00)**	$4.02 \cdot 10^7$ (0.28)	$1.54 \cdot 10^7$ (0.44)	$7.31 \cdot 10^7$ (0.39)	$13.18 \cdot 10^7$ (0.74)	$1.83 \cdot 10^7$ (0.42)
	RP_{static}	$0.49 \cdot 10^7$ (0.05)	$2.87 \cdot 10^7$ (0.39)	$2.12 \cdot 10^7$ (0.32)	$8.15 \cdot 10^7$ (0.35)	$12.39 \cdot 10^7$ (0.79)	$3.51 \cdot 10^7$ (0.22)
	$RP_{dynamic}$	$0.30 \cdot 10^7$ (0.08)	$9.62 \cdot 10^7$ (0.12)	$1.61 \cdot 10^7$ (0.43)	$6.36 \cdot 10^7$ (0.45)	$29.28 \cdot 10^7$ (0.34)	$2.08 \cdot 10^7$ (0.37)

Table 4.4: Results of 50 independent runs

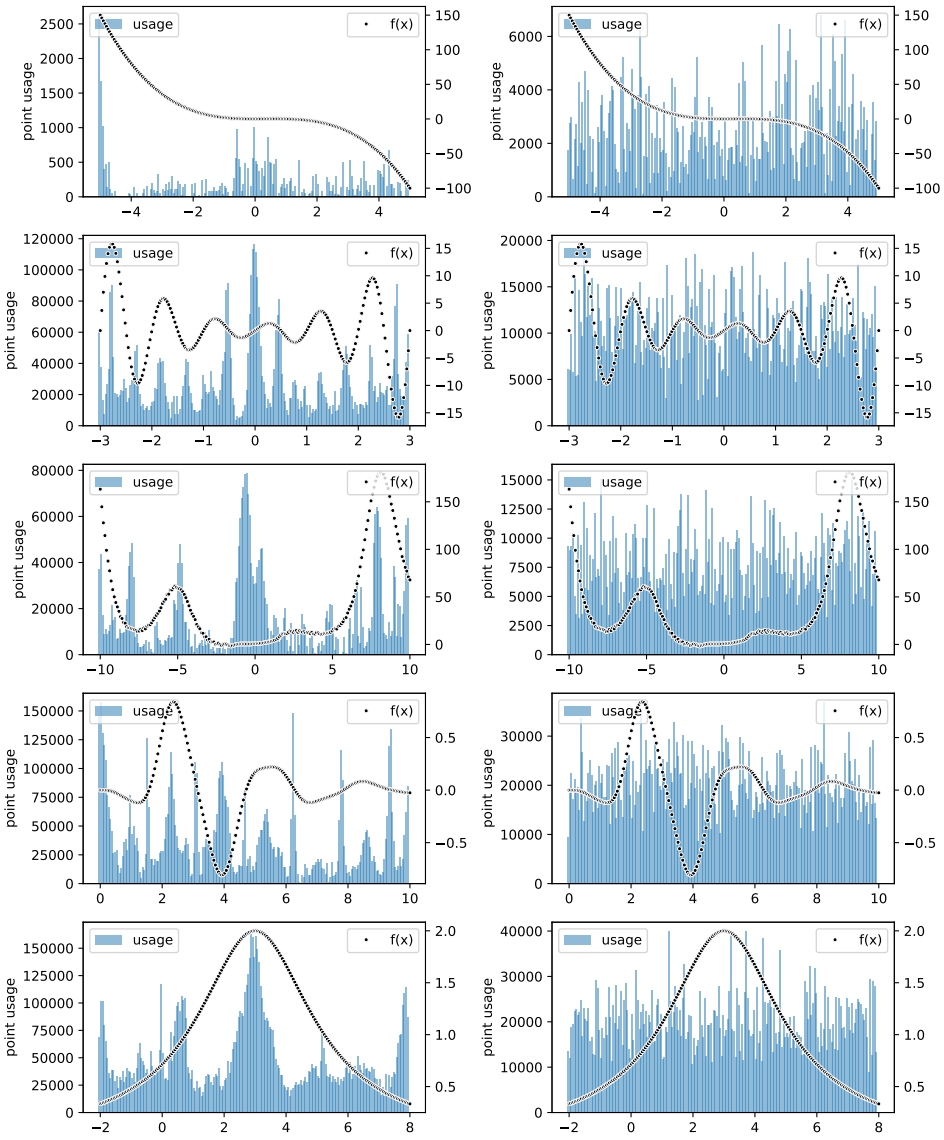


Figure 4.11: Histograms of used test cases by DP (left) and ASP (right)

Chapter 5

Summary and Conclusion

In this thesis, we first briefly summed up the main principles of the genetic programming and defined the problem of fitness predictor construction. Then we described two known methods, that construct the fitness predictors using coevolution. After that, we proposed a new method, that does not use coevolution and constructs the predictors using errors of the individuals in the current population on the training data. We compared the performance of these methods along with the standard GP and random predictors on six symbolic regression problems.

From the two experiments we conducted we can confidently say, that adaptive-size coevolved predictors (3.1.2) use the point evaluations most efficiently from all compared methods. The time requirements did not come out so definite. This may be due to the relationship between predictor size and time to perform a certain number of point evaluations shown in 4.4. Despite this fact, using coevolved predictors, especially ASP, provided a significant real-time speedup as well. Even though our method DP performed considerably well on average, it was due to the fact that it performed exceptionally well on simple-shaped datasets. For that reason, DP cannot be recommended over ASP, apart from some very specific scenarios.

Appendix A

Bibliography

[1] Douglas A. Augusto and Helio J. C. Barbosa. Symbolic regression via genetic programming. In *Proceedings of the VI Brazilian Symposium on Neural Networks (SBRN'00)*, SBRN '00, page 173, USA, 2000. IEEE Computer Society.

[2] François-Michel De Rainville, Félix-Antoine Fortin, M Gardner, Marc Parizeau, and Christian Gagné. Deap: A python framework for evolutionary algorithms. pages 85–92, 07 2012.

[3] Michaela Drahosova, Lukas Sekanina, and Michal Wiglasz. Adaptive fitness predictors in coevolutionary cartesian genetic programming. *Evolutionary Computation*, 27:1–27, 06 2018.

[4] Dheeru Dua and Casey Graff. UCI machine learning repository. `"http://archive.ics.uci.edu/ml"`, 2017.

[5] John Duffy and Jim Engle-Warnick. Using symbolic regression to infer strategies from experimental data. In *"Evolutionary Computation in Economics and Finance"*, pages 61–82. Physica-Verlag HD, Heidelberg, 2002.

[6] W. Daniel Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D Nonlinear Phenomena*, 42(1-3):228–234, June 1990.

[7] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[8] Samir W. Mahfoud. Niching methods for genetic algorithms. Technical report, 1995.

[9] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[10] Michael D. Schmidt and Hod Lipson. Coevolution of fitness predictors. *IEEE Trans. on Evolutionary Computation*, pages 736–749, 2008.

Appendix B

CD contents

/
- `datasets` ..... used datasets
- `documents` ..... digital version of this thesis
- `experiments` ..... config files of the conducted experiments
- `fpgp` ..... Python 3 source files of the project
- `notebooks` ..... used jupyter notebooks
- `README.md` ..... text file with technical instructions for the project
- `requirements.txt` ..... text file with needed dependencies