POZNAN UNIVERSITY OF TECHNOLOGY

Semantic Extensions for Genetic Programming

Bartosz Wieloch

A dissertation submitted to

the Council of the Faculty of Computing and Information Science

in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Supervisor

Krzysztof Krawiec, Ph. D. Dr. Habil.

Poznań, Poland

2013

Acknowledgments

I would like to express my gratitude to my supervisor Krzysztof Krawiec for his enormous effort he put in guiding me through my work. I deeply thank him for his patience and optimism which motivated me.

I am also thankful to my friend Wojciech Jaśkowski, for discussions and useful suggestions.

Last but not least, I am grateful to my family for understanding and supporting me.

This work was supported by National Science Centre grant # DEC-2011/01/B/ST6/07318.

Abstract

Genetic Programming (GP) is a most popular approach to automatic generation of computer programs. Standard methods applied in GP use raw fragments of evolved programs to construct new, hopefully better ones. These methods, except the selection phase, pass over the behavior of the modified programs and operate mainly on their syntax.

In this dissertation we follow alternative, semantic-oriented approach that concentrates on the actual behavior of programs in population to determine how to construct the new ones. This research trend grew up as an attempt to overcome weaknesses of methods that rely only on syntax analysis. Recent contributions suggest that semantic extensions to GP can be a remedy to poor performance of classical, syntactic methods.

Therefore, in this dissertation we firstly present the advantages and disadvantages of several possible descriptions of program's behavior. Then we introduce the concept of semantics used in all semantic extensions presented throughout this thesis.

The first semantic extension presented in this thesis is a method population initialization which forces the individuals in population to be semantically unique. We also show selected semantic-aware variants of crossover and mutation operators. In particular, we test how they perform with and without our initialization method.

Next, we introduce and formalize our novel concept of desired semantics that describes the desired behavior for given part of a program. Then we propose several methods that employ desired semantics to create new programs by combining matching parts. We show that some of these methods significantly outperform other methods, semantic as well as syntactic ones.

The second important proposition of this thesis is the concept of functional modularity. Functional modularity involves defining modules based on their semantic properties instead of syntactical ones, like, e.g., the frequency of occurring some code fragments. Functional modularity can be used to decompose a problem into potentially easier parts (subproblems), and then to solve the subproblems in isolation or together.

All the described methods are illustrated with extensive experimental verification of their performance on a carefully prepared benchmark suite that contains problems from various domains. On this suite, we show the overall advantage of semantic-aware extensions, especially for methods that rely on desired semantics.

Contents

1. Introduction **11**

1.1. Problem Setting . . . 11
1.2. Research Goals . . . 12
1.3. The Scope . . . 13
1.4. Thesis Outline . . . 13

2. Genetic Programming **15**

2.1. Overview . . . 15
2.2. The Canonical GP Algorithm . . . 17
2.3. Review of Selected Applications of GP . . . 23

3. Semantics of Genetic Programs **27**

3.1. Motivations . . . 27
3.2. Overview of Understanding of Semantics in GP . . . 28
3.3. Classification of Semantics . . . 29
3.4. Definitions . . . 33
3.5. Summary . . . 36

4. Algorithm Evaluation Framework **37**

4.1. Introduction . . . 37
4.2. Benchmark Suite . . . 38
4.2.1. Symbolic Regression Domain . . . 38
4.2.2. Boolean Domain . . . 41
4.3. Evolutionary Parameters . . . 42
4.4. The Importance of Fitness Case Assortment . . . 44
4.5. Evaluation Criteria . . . 46
4.5.1. Objectives . . . 46
4.5.2. Optimization of Program Execution . . . 50
4.5.3. Statistical Tools . . . 51

5. Semantically-oriented Search Operators **53**

5.1. Introduction . . . 53

5.2. Population Initialization 54
5.2.1. Overview 54
5.2.2. Semantically Unique Initialization 56
5.2.3. Results 58
5.3. Crossover 63
5.3.1. Overview 63
5.3.2. Experiments 65
5.3.3. Results 66
5.4. Mutation 66
5.4.1. Overview 66
5.4.2. Experiments 69
5.4.3. Results 70
5.5. Summary 70

6. Desired Semantics **75**
6.1. Introduction 75
6.2. Methods 80
6.3. Experiments 84
6.4. Results 85
6.4.1. Qualitative Results 85
6.4.2. Quantitative Results 95
6.5. Discussion and Conclusions 104

7. Functional Modularity **111**
7.1. Introduction 111
7.2. Defining and Exploiting Modularity 112
7.3. Defining Modules for Variable-based Representations 113
7.4. Functional Modularity 114
7.5. Formalization 116
7.6. Experiments 119
7.7. Results 123
7.7.1. Monotonicity Distribution 123
7.7.2. Relation Between Part Quality and Fitness 127
7.7.3. Relation Between Monotonicity and Fitness 130
7.8. Discussion and Conclusions 132

8. Conclusions **137**
8.1. Summary 137
8.2. Contributions 137
8.3. Future Work 139

A. Appendix **141**

1. Introduction

1.1. Problem Setting

Evolutionary Computation is a subfield of Artificial Intelligence that aims at solving optimization problems. It was inspired by the Darwin's theory of evolution by means of natural selection and survival of the fittest. Therefore, it uses an iterative process for incremental improvement of individuals (potential solutions) in a population. One of the subdomains of Evolutionary Computation is Evolutionary Algorithms, which focuses mostly on the bioinspired aspects of evolutions, mainly on the genetic operations. Within this subdomain, there is the field of Genetic Programming (GP) that concerns evolving of computer programs, which is the central topic of this thesis.

In a genetic programming task (GP task), the goal is to create a program that behaves in a desirable way. In general, we are interested in obtaining a symbolic expression (arithmetic expression, Boolean expression, algorithm, etc.) which processes some input data and returns an appropriate result or performs some desired actions. For instance, to such tasks belong symbolic regression problem, logic synthesis, or controlling an arm of a robot. From this point of view, a GP task can be seen as a machine learning task of supervised learning, i.e., generating a function that maps inputs to desired outputs.

For example, GP can be used to evolve an expression that classifies an object basing on its attributes (i.e., returns the proper decision class label given the input data). And indeed, such problems are successfully solved by GP [64, 67, 68, 25]. Moreover, the advantage of GP over many other classifiers that assume certain form of hypothesis representation (e.g., decision trees, neural networks, etc.) is that GP is almost *model free*. This means that we do not limit the search to a narrow space of parameters of an assumed model, but we give the GP algorithm the possibility to construct the model itself (in an imposed language), and possibly parametrize it. However, such generality brings certain drawbacks — the space of potential solutions is huge (and slow to search in consequence). Moreover, such a space can have (and typically has) more local optima in which the whole process may get stuck. Nevertheless, applying such a model free approach is often the only way to solve problems for which we do not know the appropriate model in advance.

These properties of GP tasks, together with other more sophisticated features that we will elaborate on in this thesis, give rise to certain problems that continue troubling this domain. Among them, of particular interest to us are:

- Lack of improvement in consecutive generations (this may manifests in population converging to only a small number of effectively different individuals),

- Difficulties with constructing useful code fragments (modules) and using them to assembly the complete solutions,

- High variation of individual quality between parents and offspring (effect of high epistasis that hinders developing sensible genetic operators).

Up to recent time, researchers were trying to defeat these weaknesses of GP by proposing strictly syntactic extensions of GP that operate on code fragments in isolation from they meaning (e.g., [60, 48, 75, 104]). However, in the last few years, methods have been proposed that take into account *semantics* of programs, meant as a meaning of programs or what are the effects of running them [7, 90, 69, 127]. There are convincing arguments that such semantically founded approaches will lead to a breakthrough in genetic programming theory and in the effectiveness of its practical applications.

1.2. Research Goals

In the context presented in previous section, the overall goal of this thesis is to propose semantically-aware extensions of genetic programing, aimed primarily at improving its efficiency (meant as the computational effort required to find a solution of acceptable quality) and scalability (meant as the capability of producing an acceptable solution altogether). The specific objectives include:

1. To analyze the properties of different types of descriptions used to characterize program behavior (its semantics), particularly with respect of their applicability in the context of GP.

2. To devise methods that exploit the semantic descriptions of evolving programs and enable solving problems that are hard to canonical GP.

3. To experimentally verify the effectiveness of the proposed methods on a wide range of benchmarks of varying difficulty.

4. To compare the proposed methods with other semantic approaches.

5. To propose an approach to defining semantic modules and to give experimental premises of it practical applicability.

1.3. The Scope

For obvious reasons, this thesis cannot embrace all possible scenarios of GP algorithms and its applications to tasks, so it focuses on a certain class of GP programs, algorithm parameters, and GP tasks.

We limit our considerations to canonical, tree-based genetic programming [60]. Other GP variants, like linear genetic programming [6] or cartesian genetic programming [92], are not considered here. However, the proposed methods can be adopted to those and other varieties of GP.

Typical GP algorithms have many parameters that control particular aspects of evolutionary run. It is not feasible to test exhaustively their all possible (or even only the sensible) values. Therefore, in the experiments we use typical settings taken from Nguyen's studies [98, 99, 125] and are based on Koza's works [60, 61].

Historically, GP has been applied to a wide range of problems — from toy examples [60] to designing quantum algorithms [118, 119]. Again, tackling all them in this thesis was unrealistic. However, we made an effort to compare the algorithms studied and proposed here on a representative of problems. Our benchmark suite contains 39 instances of problems from two domains: symbolic regression (19) and binary functions (20). These domains are two most popular in the GP community [87].

1.4. Thesis Outline

This dissertation proposes several, partially independent, extensions of GP, and describes them in separate chapters. Particular chapters present also the outcomes of computational experiments that validate these extensions. This implies certain organization of the text, where the description of the experimental environment precedes the presentation of particular semantic extensions.

More specifically, the dissertation is organized as follows.

Chapter 2 introduces the canonical genetic programming popularized by John Koza in his book [60]. Then it presents selected successful applications of GP to show the wide range of areas GP can be applied in.

In Chapter 3 we discuss the possible definitions and interpretations of semantics in the GP context. Then we propose a classification of types of semantics, define their properties, and compare them.

Chapter 4 presents the problems that form our benchmark suite and shows the impact of the manner in which fitness cases are selected on the problem hardness. Then we present and justify the common parameter settings applied in our experiments and introduce evaluation criteria used further to compare the methods.

In Chapter 5 we propose a method of semantic initialization of population. Then we

present selected variants of semantic crossover and mutation. Finally, we show the influence of these semantic operators on the achieved effectiveness.

Chapter 6 is the main contribution of this thesis. Therein, we introduce a novel concept of *desired semantics* and present a number of methods that exploit it. The results obtained in experimental part show substantial advantage of some of them over the canonical genetic programming.

Chapter 7 presents our concept of *functional modularity* and defines this concept formally. Using this concept, in the experimental part of that chapter we investigate the modular properties of problems from the benchmark suite. We conclude this, mostly theoretical, chapter by demonstrating the possibility of practical applications of the proposed concept in a simple experimental setup.

The last Chapter 8 summarizes this thesis, reviews its main contributions, and points out the future research directions.

2. Genetic Programming

2.1. Overview

Genetic Programming (GP) is a metaheuristic algorithm for solving variety of problems by biologically inspired evolution of population of computer programs. The key feature of GP is that the goal of it is to produce a valid computer program, i.e. a procedure (recipe) for solving a whole *class* of tasks. This is the main conceptual difference between GP and genetic algorithms or other similar metaheuristics which, in contrast, optimize just a particular instance of the problem, for example by tuning some parameters in the solution frame.

This chapter will introduce the main idea of GP and will present those aspects of GP which are directly related to this thesis. To get a wider perspective and a more in-deep understanding of GP mechanism and many variants of this metaheuristic we encourage to familiarize with many introductory books about GP. For example, we highly recommend a very nice book [105] (freely available online[1]) written by Poli, Langdon, and McPhee. It introduces GP but it also describes some advanced techniques and practical applications of GP. A bit older, but also a good book is [4] written by Banzhaf *et al.*

GP belongs to a large family of methods inspired by evolution — to evolutionary algorithms [91]. This group of methods takes inspiration from the biological world, where natural selection causes the fittest individuals to survive and keep on evolving. It is characteristic of evolutionary algorithms that a population of *multiple* potential solutions is simultaneously maintained instead of just modifying a single solution like in local search methods (e.g. hill climbing, tabu search [35, 34], or simulated annealing [56, 24]). Other techniques belonging to the evolutionary algorithms, beside genetic programming and genetic algorithms mentioned above, are evolutionary programming [27], evolution strategy [108], differential evolution [121], learning classifier system [45], and gene expression programming [26] (which in fact is not fundamentally different from the GP philosophy).

The idea of genetic programming, as it is currently known, was firstly presented by Nichael Cramer [18], but extended by John Koza [59] and greatly popularized in the beginning of 1990' thanks to his books [60, 61]. In the classical GP, individuals (potential solutions) are tree-like structures (originally Koza used Lisp s-expressions) which are convenient to encode and ease interpretation. For example, mathematical expressions as well

[1] http://www.gp-field-guide.org.uk/

as programs written in many languages are directly or indirectly represented as trees. However, such structures, despite being very simple and natural, have also certain drawbacks. For example, when a solution requires many copies of some routine (a common subprogram/procedure), the code has to be cloned many times in the individual. This may be, in general, not easy to achieve using standard evolutionary operators like crossover or mutation. Therefore, Koza proposed mechanism of discovering such reusable code fragments — automatic defined functions (ADFs) [61], whereas Lee Spector proposed a similar method — automatically defined macros [117]. Such mechanisms simply extend the tree-based structure to enable encoding of the useful subfunctions. For these and other reasons, other variants of GP have been proposed with different representation of programs, e.g. linear genetic programming [6] or cartesian genetic programming [92], which enable coding graph-like structures (implicitly in the former case, explicitly in the latter).

As these methods have different properties than the canonical tree-base programs, the choice of an appropriate method depends on the problem and experience of a researcher. However, the canonical tree-based genetic programming, as proposed by Koza in his book [60], is probably the most popular variant of GP, most intensely extended and analyzed by researchers. This canonical version of GP became the *ipso facto* standard, thus all methods presented in this thesis are also related to and compared to the canonical GP.

Somewhat similar approach for inducing programs is known as *inductive logic programming* (ILP) [95]. ILP methods induce first-order Horn clauses from positive and negative examples. The programs are generally represented in Prolog language. Even though both ILP and GP construct programs, ILP is strictly logic-oriented and the induction is based on a completely different approach (generalized rule-learning methods). Therefore, in this thesis we will not consider ILP as it is a different paradigm for inducing programs. A comparison of GP and ILP systems, as well as propositions of methods combing both approaches may be found, e.g., in [122, 137, 138].

GP as a computational intelligence paradigm has several unique properties that distinguish it from others. First of all, a solution is constructed from a problem-dependent set of instructions (sometimes called *functions*). An instruction, when executed or interpreted, performs some predefined calculations or takes some actions. The effects of processing carried out by one instruction have usually direct impact on execution of other (typically: subsequent) instructions by providing them with values of arguments or changing some variables.

Another feature quite specific to GP is that execution of a program (a constructed solution) is not a one-step process, but involves an entire *sequence* of actions (executions of single instructions). Therefore, the final result of program execution is not the only observable outcome; we can also observe intermediate states of program execution, i.e., the results of executing parts of the program. This feature is very important and extensively exploited in many semantic-aware approaches which rely on these intermediate states (see

Chapters 5–6 for details and review).

The third property worth mentioning is that, in most cases, the task for a GP algorithm is given in form of a set of *fitness cases*, i.e., pairs composed of input data (that should be processed by programs) and desired output value (correct result or effect) for that data. In this respect, a GP algorithm solves a supervised learning from example task, where each example (corresponding here to fitness case) defines the correct 'behavior' of a classifier (here: evolved program). A potential GP solution is evaluated on several (e.g., 20, but sometimes several thousands or even more) independent *tests*, and the attained fitness reflects how well the program solves all these tasks.

To sum up, a problem for GP is stated by defining three formal objects:

1. A set of functions (instructions) from which the solution should be built.

2. A set of rules telling how instructions can be assembled to form programs (syntax rules). This may include rules that impose certain constraints on the set of 'feasible' programs, like maximal allowed program length (size of a solution).

3. A fitness function — i.e., the function to optimize. It is usually, but not always, given as a set of fitness cases that specify the correct (expected) program outcomes, and a metric measuring the similarity between these correct values and actual program answers.

The first two points relate to the syntax of the allowed solution and are often common among many different problems from the same domain. For instance, for the domain of logic circuits synthesis, one can decide to use the same set of instructions containing logic gates like AND, OR, and NOT. The last point is usually much more specific for the desired program outcome. Although fitness function could also depend on non-behavioral aspects of a program, like its syntax (e.g., costs of used instructions), that is quite a rare case that will not be considered in this thesis.

Other evolutionary parameters that control the behavior of a GP algorithm, like population size, initialization method, presence or absence of elitism, probability of engaging particular search operators (mutation, crossover), etc., can be obviously very important for successful solving of a problem. It is nevertheless essential to emphasize that they are not part of the problem itself. However, when solving specific problems, these parameters often have to be tuned to attain acceptably good outcomes.

2.2. The Canonical GP Algorithm

The overall scheme of all evolutionary algorithms is very similar and consists of three main stages: (1) generating an initial population, (2) assessing fitness, (3) generating a new population. The last two steps are repeated until an ideal solution is found or other

Algorithm 2.1 The *Grow* program initialization algorithm

```
1: procedure GROW(depth, maxDepth)
2:     if depth = maxDepth then
3:         return a randomly chosen terminal from the function set
4:     else
5:         n ← a randomly chosen instruction from the function set
6:         k ← expected number of children for n
7:         for i ← 1...k do
8:             child_i(n) ← GROW(depth + 1, maxDepth)
9:         return n
10: end procedure
```

Algorithm 2.2 The *Full* program initialization algorithm

```
1: procedure FULL(depth, maxDepth)
2:     if depth = maxDepth then
3:         return a randomly chosen terminal from the function set
4:     else
5:         n ← a randomly chosen non-terminal from the function set
6:         k ← expected number of children for n
7:         for i ← 1...k do
8:             child_i(n) ← GROW(depth + 1, maxDepth)
9:         return n
10: end procedure
```

stopping criteria are met (e.g., the algorithm runs out of the allocated resources, typically time). Below, we describe these steps in more detail for tree-based genetic programming.

In canonical GP, individuals are represented as expression trees. Solutions in the initial population are generated randomly from a given set of non-terminal instructions (which may appear as non-leaf nodes in a tree; e.g., functions), and terminals (which may appear as leaf nodes; e.g., constants or input variables). This process should not violate the constraints imposed by task specification (e.g., the limit on the number of nodes per tree/program, maximal tree height, etc.). One common method (proposed by Koza) is the *Grow* algorithm that builds trees in the depth-first manner. In each step, the algorithm

Algorithm 2.3 The *Ramped Half-and-Half* program initialization algorithm

```
1: procedure RAMPED-HALF-AND-HALF(minDepth, maxDepth)
2:     r ← a random real value from range [0; 1)
3:     d ← a random integer value from range [minDepth; maxDepth]
4:     if r < 0.5 then
5:         return GROW(1, d)
6:     else
7:         return FULL(1, d)
8: end procedure
```

Algorithm 2.4 Evaluation of an individual

```
1: procedure EVALUATE(p)
2:     C ← set of fitness cases
3:     value ← 0
4:     for all c ∈ C do
5:         result ← EXECUTE(p, input(c))
6:         value ← value + |result − output(c)|
7:     return value
8: end procedure
```

selects randomly either a terminal instruction, if it is on the maximal allowed depth ($maxDepth$), or any instruction (terminal or non-terminal) otherwise (see Algorithm 2.1). The *Full* algorithm is similar but it enforces building full trees by selecting only non-terminals if the depth limit ($maxDepth$) has not been reached (see Algorithm 2.2).

Depending on the domain and problem, building full or non-full trees may be more or less desirable. Determining the most appropriate population initialization method for a specific problem may be hard, so the method of generating the initial population that is most widely used in practice combines both these practices. Called *Ramped Half-and-Half* (see Algorithm 2.3), it builds a tree by selecting one from the above two algorithms half the times (iterations of recursive tree traversal) and selects the maximal depth randomly from 2 to 6. Other, more complicated methods (like, e.g. PTC2 [82]) are less popular despite the fact that they allow more detailed control of the tree building process.

The goal of the evaluation phase is to assess fitness of each individual in the population. As introduced earlier in this chapter, individuals are usually tested on a set of fitness cases. All of them are supposed to be correctly solved by a good solution. The evaluation procedure executes (or interprets) a given program to obtain its results for each fitness case. Then the procedure computes the disparity between the actually produced results and the desired output values. A sum of these errors is treated as a *minimized* fitness value. Often the returned value is additionally transformed by an equation $1/(1 + value)$ to obtain a *maximized* fitness value. In general, the particular errors can be aggregated in more sophisticated ways but the simple sum is the most common approach. Algorithm 2.4 shows a pseudocode of the most popular variant. The evaluation part of GP algorithm is particularly tightly related to semantic aspect of evolved programs, and will be elaborated on in Section 3.4.

The third phase — generating the new population — is the most complex stage that can take different forms depending on the variant of GP algorithm and application domain. However, there are some generally accepted practices which are most common in use.

A new population is created by breeding new individuals independently, one-by-one. Each individual is bred by applying a breeding operator (reproduction, mutation, crossover), selected with some predefined probabilities. Each of these operators selects a specific num-

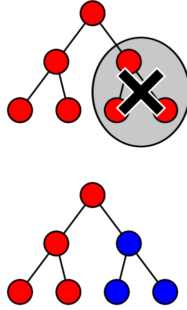


Figure 2.2.1.: The subtree mutation operator.

ber of individuals (parents) from the current population and produces a number of offspring (usually the number of children equals the number of parents). Parents are selected with replacement from the current population by using some *selection method*. The most popular one is the *tournament selection* or *fitness-proportional selection* (also known as *roulette selection*). The first one returns the fittest individual from a small group (usually 3 to 7) of randomly drawn individuals. The roulette selection draws individuals from the population with probabilities proportional to fitness values. The proportional selection is sensitive to the absolute value of fitness, which could be calculated quite arbitrary. Also, a single very fit individual is often selected by this operator and it can easily dominate the population. Therefore, the tournament selection is widely recognized as safer and more robust, and is more often used in practice.

Concerning the breeding operators, the *reproduction operator* is the simplest one — it just makes a copy of the selected single parent and puts it to the newly created population. When used together with a selection procedure, this operator preserves good individuals (i.e. winning the selection procedure) without any modification so the genetic material will survive without risking any potential destruction.

The *mutation operator* accepts a single parent and changes it in some, typically minimal, way. In case of tree-like structures used in GP, a minimal alteration is to modify a single node in the tree. However, this kind of mutation is rarely used in GP practice. Instead, the most popular mutation operator replaces a randomly selected subtree in the parent with a newly generated subtree (see Figure 2.2.1). The new subtree is built using one from the methods used to generate initial population — most often it is the Ramped Half-and-Half algorithm.

The *crossover operator*, as opposed to the previous ones, needs two parents. It randomly selects one subtree in each parent and swaps them, producing so two offspring. Figure 2.2.2 illustrates this process. If only one offspring is needed (because, for instance, there is space for only one individual in the new population), the second child is thrown away.

In contrast to genetic algorithms, the crossover and mutation operators in GP are usually not 'chained', i.e., the mutation operator is usually not applied to the product of

Algorithm 2.5 The Canonical Genetic Programming

```
 1: procedure CanonicalGP
 2:     maxGen ← maximal number of generations
 3:     popSize ← population size
 4:     crossoverProb ← probability of performing crossover
 5:     mutationProb ← probability of performing mutation

 6:     P_0 ← ∅                                          ▷ initialize first population
 7:     while |P_0| < popSize do
 8:         P_0 ← P_0 ∪ { Ramped-Half-and-Half(...) }
 9:     p_best ← ∅
10:     fitness(p_best) ← −∞
11:     i ← 0
12:     loop
13:         for all p ∈ P_i do                           ▷ evaluate population
14:             fitness(p) ←Evaluate(p)
15:             if fitness(p) > fitness(p_best) then
16:                 p_best ← p
                                                         ▷ check stopping condition
17:         if i = maxGen − 1 or p_best is the ideal solution then
18:             return p_best
19:         P_{i+1} ← ∅                                  ▷ create next population
20:         while |P_{i+1}| < popSize do
21:             r ← a random real value from range [0; 1)
22:             if r < crossoverProb then                ▷ perform crossover
23:                 p_1 ← Selection(P_i)
24:                 p_2 ← Selection(P_i)
25:                 c_1, c_2 ← Crossover(Copy(p_1), Copy(p_2))
26:                 P_{i+1} ← P_{i+1} ∪ {c_1}
27:                 if |P_{i+1}| < popSize then
28:                     P_{i+1} ← P_{i+1} ∪ {c_2}
29:             else if r < crossoverProb + mutationProb then   ▷ perform mutation
30:                 p ← Selection(P_i)
31:                 c′ ← Mutation( Copy(p) )
32:                 P_{i+1} ← P_{i+1} ∪ {c}
33:             else                                     ▷ perform direct reproduction
34:                 p ← Selection(P_i)
35:                 P_{i+1} ← P_{i+1} ∪ { Copy(p) }
36:         i ← i + 1
37:     end loop
38: end procedure
```

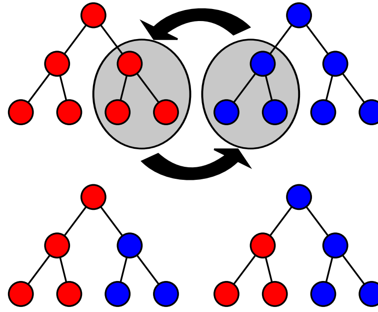


Figure 2.2.2.: The subtree crossover operator.

crossover. Rather than that, mutation is used independently, as one of available genetic operators. It is worth to notice, that the mutation operator was not originally used in Koza's work [60], presumably due to large populations employed there. Only reproduction and crossover operators were applied in that work (with, respectively, 10% and 90% probability). However, most of later studies have used mutation instead of the reproduction operator. Therefore, in this thesis we adopt this standard and use only crossover and mutation operators, without explicitly copying unaltered individuals. However, an implicit copying of unaltered individuals can still occasionally happen because, when crossover or mutation fails in generating children that fulfill given constraints (usually the maximal allowed tree depth), the parent solutions are copied. Nevertheless, this feature is also considered as a standard, and is implemented in, among others, the ECJ package [83] on which the experimental part of this thesis is based on.

Another important remark is that the probabilities of operator engagement (*crossoverProb*, *mutationProb*) does not translate directly to the same proportion of individuals in a new population created by this operator. This happens because a single application of crossover produces *two* individuals at once. Therefore, if the probability of crossover equals 0.5, there will be approximately 66% individuals produced by the crossover in the new population. In general, the expected number of individuals produced by a crossover operator is

$$\frac{2 \cdot crossoverProb}{1 + crossoverProb} \cdot popSize\,,$$

where $crossoverProb$ is the probability of the crossover operator, and $popSize$ is the size of a population.

The pseudocode of the entire canonical GP is shown in Algorithm 2.5. To summarize, the canonical genetic programming has the following main features:

- The evolved programs have form of trees,
- Individuals in the initial population are built using the Ramped Half-and-half procedure,
- Evaluation of a program consists of executing it for each fitness case,

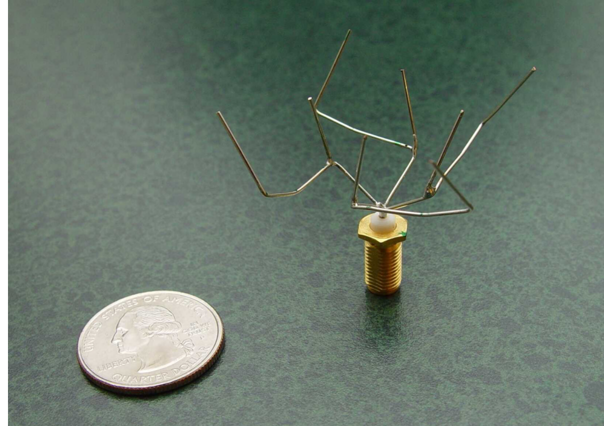


Figure 2.3.1.: An evolved antenna for NASA's Space Technology 5 (ST-5) mission. The image comes from www.nasa.gov website.

- New individuals in the next population are created with either crossover, mutation, or reproduction operators.

2.3. Review of Selected Applications of GP

As we mentioned in the previous section, GP can be considered as a machine learning technique, so no wonder that its application areas, both actual and potential, are diverse. There are many successful application of the genetic programming (and evolutionary computation in general). Some of the most spectacular results are presented at the annual "Humies" competitions organized since 2004 at the Genetic and Evolutionary Computation Conference (GECCO [107]). The goal of "Humies" is to award a human-competitive results produced by a genetic or evolutionary computation. In short, the human-competitiveness is defined as being at least as good as human-created solution that is considered as an achievement in a given field.

One example of such field is quantum computing. It turns out that genetic programming is able to evolve system-size-independent quantum algorithms such as, for example, the Quantum Fourier Transform algorithm presented by Paul Massey *et al.* in [85]. In 2004 Lee Spector won (*ex aequo*) the competition for the work on automatic production of quantum computer programs [118].

Jason Lohn *et al.* won *ex aequo* the second first prize of the same "Humies" edition with an evolved X-band antenna which was deployed on NASA's Space Technology 5 spacecraft [78]. Figure 2.3.1 presents this antenna; despite its unusual, asymmetric shape, it turns out to have excellent properties. One year latter, Preble *et al.* used genetic programming to evolve structures of photonic crystals with maximal band gaps, which enable the control of the flow of light on nanoscale [106]. In [62], Koza *et al.* used GP to automatically synthesize complete designs for optical lens systems which duplicates previously patented

solutions. Most of the evolved systems do not infringe the previous patents and some of them are completely novel designs.

Hod Lipson demonstrated in [77] the ability of GP to synthesize compound 2D kinematic mechanisms. He tested it on classical kinematic challenges of 19th century mechanical design. Latter, Michael Schmidt and Hod Lipson published in Science a paper [113], which shows that automatic discovery of physical laws is possible even without any prior knowledge about physics, kinematics, or geometry. Their GP algorithm successfully discovered Hamiltonians, Lagrangians, and other invariants and laws of geometric and momentum conservation.

Genetic programming was also successfully applied in biology. In [135] Widera *et al.* evolved the protein energy function used in protein structure prediction. The energy function, built by combining a set of expert-designed energy terms, correctly distinguished good and bad candidate structures.

In computer vision, GP was successfully applied e.g. to generate a low-level image feature extractor and visual object recognition. Krawiec in [65, 67, 68] demonstrated competitive performance for real 3D object recognition both in visible spectrum and in radar modality. In 2006 Trujillo and Olague presented an evolved interest point detector [124] which exhibit state-of-the-art performance. Two years letter, Perez and Olague in [103] showed an invariant region descriptor which could be better than solutions applied on e.g. the SIFT descriptor. In 2008, Kadar shows an automatically constructed boundary detector for natural images [52].

Weimer *et al.* in 2009 won first prize in “Humies” for a system to automatically finding and repairing bugs in real software written in C language [134]. Their approach did not require any formal specifications or program annotations, and it worked for off-the-shelf legacy applications. This work had been appreciated by the software engineering community and won both the distinguished paper award and the IFIP TC2 Manfred Paul award on the 31st International Conference on Software Engineering (ICSE is the premier software engineering conference). Orlov and Sipper in [102] presented a methodology for evolving bytecode of unrestricted Java programs. These works, and the paper by Forrest *et al.* [28], clearly show that genetic programming can be used not only to repair real world computer software written in popular languages, but also to construct computer programs from scratch.

Games are another quite popular area of GP applications. In 2007, Hauptman and Sipper [42] presented a GP approach for the Mate-In-N problem in the game of chess (finding such a move that the opponent cannot avoid being mated in N moves). Two years letter, Hauptman *et al.* [41] demonstrated the ability of GP to beat human players as well as the human-designed algorithms in the Rush Hour puzzle.

Genetic programming can be successfully applied also in pure mathematics. In [120], Spector *et al.* described the production of human-competitive results in the discovery of

complex algebraic terms that fulfill certain constraints (mathematicians knew that the terms exist, but could not find them analytically). Alex Fukunaga in [32] applied GP to construct an automated heuristic discovery system: GP evolves local search heuristics that are subsequently used to solve instances of SAT (Boolean satisfiability) problem. KHosraviani *et al.* [55] applied GP techniques to find nearly-optimal design of project organizations.

There are a lot of other successful application of genetic programming. Several dozen results generated automatically by GP are human-competitive[2]. More than twenty of them infringe or duplicate the functionality of previously patented inventions. A reader interested in a more thorough review of most prominent GP achievements is referred to [105, 101, 109, 110].

[2]both [63] and `http://www.genetic-programming.com/humancompetitive.html` (data from December 2003) present 36 results

3. Semantics of Genetic Programs

3.1. Motivations

Before introducing the semantic perspective for GP, we should present the rationale for this particular research direction. In other words, why taking into account the semantic properties of evolving programs and appropriately redesigning the algorithms is worth any attention in the first place?

A short answer to this question is that a typical GP task comprises much more than an optimization task. However, because GP is usually perceived only as a mere variant of evolutionary algorithm, many researchers tend to adopt the "conservative" optimization perspective, in which problem instance is defined by a domain (search space) and a function being optimized within that domain. In particular, it is assumed that the search algorithm knows very little about that function, and querying it is the only allowed action. In this sense, the function being optimized is a black-box oracle.

An important tenet of this thesis is that applying this stance to GP is utterly wrong and leads to unnecessary complexity. In fact, in virtually all GP problems the fitness function is based on a metric that captures the discrepancy between the actual program output and the desired program output (c.f. Section 2.2). Usually, the knowledge about such target output could be utilized to improve the performance of a search process.

Secondly, the solutions considered in GP are programs, and programs are *compositional* by nature. By this we mean that a set of elementary components (instructions) is given, and any solution is a composite of these components. As a consequence, a solution can be decomposed into parts, which can be analyzed independently. It may not make sense to evaluate them using the fitness function, but in most cases they can be independently executed.

The above features make GP tasks quite specific when compared to arbitrary optimization and learning tasks. The methods proposed in Chapters 6 and 7 make extensive use of these features, which, as the experimental part later demonstrates (Sections 6.4), leads to substantial benefits in performance. In particular, the following section provides a broad view of the semantics and especially its forms in the genetic programming. The next section will try to classify semantics in respect to some important properties, and then in following section we present the definition and properties of the concrete semantics in the form as used in this whole thesis.

3.2. Overview of Understanding of Semantics in GP

The word *semantics* is defined in dictionaries as the meaning, or an interpretation of the meaning of a symbol, word, sign, facial expression, etc. Alternatively, in linguistics it is the study of relationships between sentences of words and their meanings [22].

In computer science, the term 'semantics' is mainly used in the context of programming languages and it refers to the meaning of a language construct, as opposed to syntax that deals with the *form* of programs (program code). In theory of formal languages, the meaning is often described using operational or denotational semantics. The operational semantics defines the meaning of a concrete programming language construct as the computation it induces (i.e., what are the operations that get executed when the program is run). Therefore, this notion of semantics describes *how* the effect is produced. In contrast, the denotational semantics models the meanings by mathematical objects representing the *effect* of executing, and is not interested how it is produced.

In the context of genetic programming, *semantics* usually refers to a description of what a program does, i.e. what are the effects of execution of an entire program or its constituent components (subprograms or even individual instructions). However, in a loose form, the semantics could, if reasonable, incorporate some information not strictly related to the behavior but describing the program itself.

Let us notice that there is a nice parallel between the evolutionary aspects of GP and the syntax-semantics divide. Namely, the code (syntax) of a program can be treated as individual's genotype, while its meaning (semantics) can be likened to individual's behavior (phenotype). This convention has been widely used in GP literature ([14, 49, 86]), and seems to be quite broadly accepted by the major researchers in the field, so we will adopt it in this thesis.

In the GP literature, program semantics has been defined in at least three ways:

- as a canonical representation of a program [8],
- a list of program's responses to some input data [125], and
- as fitness value [111].

In following, we describe these alternative approaches.

Using a canonical representation of a program means that all programs giving equivalent results (e.g., functions returning the same value) are encoded in one designated form. In the task of logic functions (circuits) synthesis, this form can be a Reduced Ordered Binary Decision Diagram (ROBDD) that represents a genome itself or is used at a phenotype level [7]. Also for symbolic regression tasks, it is possible to develop appropriate canonical representation (see e.g. [139]). A canonical representation has the advantage that each individual identifies one unique behavior. However, it may be (and in practice mainly is) still hard to analyze such canonical forms thoroughly if they have a complicated structure

like trees or graphs (e.g. like ROBDDs). Therefore, usually only a simple equivalence checking is performed.

Another representation of semantics might be a list of pairs consisting of input data and a result computed by a program in effect of processing of these data. Such semantics is called *sampling semantics* because it describes the behavior of a program only for a finite number of cases (different program inputs). As the set of used input data is usually common among computed semantics, this representation reduces just to a list of produced outputs. For instance, in symbolic regression tasks it will be a vector of numbers. Comparing such semantics boils down to comparison of two vectors, which can be defined much simpler than comparing lists of pairs with arbitrary input data.

The input data used to calculate such sampling semantics may be randomly chosen from an appropriate problem domain (like in e.g. [98]) or may be fixed and come with a problem, as a part of problem definition (e.g. [69]). The key feature of this representation is that the list has generally constant length and, more importantly, its elements have simple structure and thus the entire semantics are quite easy to analyze.

The elements of sampling semantics do not necessary have to be scalars — they may be also matrices etc. For example, in an image processing task a single element of such semantics may be an image produced by a program. Moreover, in general the elements of sampling semantics may have even different types: some of them may be integers, some reals, other matrices or nominal values. However, this would be a very exotic scenario with limited applicability, so for the sake of this thesis, we will assume that all elements of sampling semantics are of the same type.

The third interpretation of program semantics present in the GP literature is just a single scalar. In this simplest case, the semantics may be identified by, for example, a fitness value, or a phenotype identifier. Thus, from this point of view, even the canonical GP (described in Section 2.2) use extremely degenerated semantics as the fitness value is used, e.g., in selection phase. However, such exceedingly reduced description makes more thorough analysis of programs behavior impossible, and therefore such kind of extraordinary meaning of semantics will not be considered in further part of this thesis.

3.3. Classification of Semantics

Before formally pinning down the definition of semantics to be used in this thesis, it is worth considering this concept in a more abstract form. To this aim, in this section we discuss the forms of semantics and their properties.

Firstly, semantics could fully describe a behavior of a program in such a way that it explains how a program will work in any admissible conditions, for any possible input data, etc. This would be the case for, e.g., operational and denotational semantics mentioned in the previous section. Such *complete semantics* may seem very desired, however it has

some disadvantages too. If the space of all possible program behaviors is huge, then a complete semantic description may also be very long as it has to be able to explain any possible program from this space. In a consequence, such exact semantics may be used only for a reasonably small space of phenotypes, otherwise it will be useless in practice. It should be emphasized that analyzing the complete semantics is generally as complex as analyzing a raw phenotype and therefore has hardly any advantages. This statement is true despite the fact that in some situation a description of a sophisticated program could be very simple and succinct.

Let us illustrate the above observations with exemplary programs that accept a vector of real numbers. Denotational semantics of any sorting algorithm applied to a vector $\mathbf{x}$ of numbers may be written as $\forall i < j : x_i \leq x_j$, no matter how many numbers are being sorted. However, it should be noticed that even though this particular program (an algorithm which sorts numbers) has so simple and easy to encode semantics, many other programs will have much more complex semantic description. An example of such program is a program solving a kind of a 'needle in a haystack' problem, where the task of the program is output the logical value *true* only for a unique combination of input values $\mathbf{v}$. Its semantics requires enumerating individually all variables: $x_1 = v_1 \wedge x_2 = v_2 \wedge \ldots \wedge x_n = v_n$, and it is very unlikely that this description can be reduced to any simpler form.

In contrast to the complete semantics, it may be worth considering also several types of *incomplete semantics* which may describe a program behavior in an non-exhaustive manner. We say that semantics is *partial* if it describes behavior of a program only for a subset of all possible conditions (inputs) and says nothing about how it works in others. Semantics can be *inexact* when a description itself is not precise due to, e.g., value quantization or using linguistic terms like 'small' or 'big'. If this is not the case, i.e., the description is precise, we say that such semantics is *exact*.

In this thesis, we are interested in semantics as a means for guiding the evolution in the right direction. This means that an adopted form of semantics has to make it possible to construct a function to evaluate or to compare such semantics. In evolutionary perspective, useful semantics has to contain enough information, meaning that it should well characterize at least those aspects of a program which are evaluated and used to calculate its fitness value. Semantics that meets this requirement will be called *sufficient* in this thesis. For example, a sampling semantics constructed on a basis of fitness cases, i.e., examples that come as a part of problem specification (such sampling semantics is formally defined in the following section, see Definition 2), is definitely sufficient, because it allows to calculate the fitness value directly. It is worth to notice that an incomplete semantics may be sufficient, which is also the case here (in general, sampling semantics is partial by definition).

In contrast, a sampling semantics calculated for a random set of input data (e.g., random function arguments in the case of symbolic regression) is usually an example of *insufficient*

semantics because the correct output values for these points are usually not known. In general, all forms of semantics which do not contain all aspects taken in evaluation procedure are insufficient. Of course, the terms *sufficient semantics* and *insufficient semantics* do not prejudge whether a particular form of semantics will be useful for evolutionary search — for instance, insufficient semantics may be used to increase diversity in a population, or to make a crossover operator which works more locally (e.g., see Section 5.3).

A semantic representation that is sufficient may be enriched with more information that is not explicitly used to evaluate the program. Such additional information may be helpful for, e.g., capturing differences between individuals, and exploited by evolutionary algorithms in various genetic operators, e.g. to diversify a population. Such semantics enriched with additional program's description is *augmented* in the sense that it allows deeper investigation of the individual, not only focusing on properties strictly optimized in the given problem, which is the case in standard GP.

For example, when a programming language involves memory, e.g., registers or variables, semantics may be enriched with state of memory. This means that program's behavior may be described not only by the single final output but also by the final values of all other auxiliary variables created/used by a program, or even some intermediate (i.e. historical) values of them, to reflect the process of calculation the final value (see [66] for an example of such approach).

In case of symbolic regression, the roles of augmented semantics could be played by derivatives calculated at given points or program's responses to supplementary (i.e., not included in the original training set) input data. Such sensible additional information may be useful because many quite different programs (i.e., evolved mathematical equations) may have similar values for given fitness cases and, in consequence, they are hardly distinguishable without augmented semantics. For instance, the expression $f_1(x) = x$ and $f_2(x) = x^3$ are indistinguishable for a set of fitness cases $x \in \{-1, 0, +1\}$ because both functions have equal values for these three inputs. However, functions values for other x ($x \notin \{-1, 0, +1\}$) and derivatives for any x are different for f_1 and f_2 function. For another example, when the task is to evolve a controller for the fastest race car [123, 79, 1], it seems reasonable to include in semantics not only the total lap time but also information about number of collisions, number of overtook cars, or whether a car got into a skid, etc.

To sum up, additional elements of augmented semantics may include general properties such as length of a program, or some expert designed properties suitable in given problem. Gustafson *et al.* in work [37] proposed some other elements which could be included in such kind of semantics.

Exploiting any semantics more sophisticated than a single scalar value seems a bit like a kind of *multi-objectivization* (see Knowles *et al.* [57]), where each element of used semantics corresponds somehow to a different criterion. In particular *sufficient* semantics is similar to multi-objectivization via the decomposition of the original objective [39], whereas the

	Canonical form	*Sampling semantics*	*Fitness value*
behavior description	perfect	fair	extremely rough
complete	✓	✗	✗
partial	✗	✓	✓
exact	✓	✓	✗
sufficient	✓	✓[69], ✗[98]	✓
augmented	✓	✗ (but easily extendable)	✗
equivalence testing	✓	✓	✓
similarity measure	difficult to define	standard norms (Euclidean, city-block, etc.)	✓

Table 3.1.: Characterization of different forms of semantics present in literature in terms of the properties proposed in this thesis.

augmented semantics may be perceived as multi-objectivization via the addition of new objectives [13]. However, in this thesis we will use neither any specialized multi-objective optimization algorithms nor any method of dynamically changing selection pressure (e.g., as in *implicit fitness sharing* proposed by Smith *et al.* in [115]). Moreover, all methods described in this thesis (Chapters 5–6) focus on semantics as a whole, treating evenly all elements of semantics representation, and they use semantics in the process of program construction (initialization and breeding operators — see Section 2.2), not in the selection phase.

To summarize, we proposed here a set of properties that seem to be helpful for characterizing different aspects of semantic description of programs. According to this, a semantic of a program can be:

complete — fully and precisely describes behavior of a program in any admissible conditions,

partial — describes a program results only in some subset of all possible situations,

exact — accurately describes behavior for presented situations,

sufficient — allows to guide an evolution process in direction defined by the problem itself (i.e., is consistent with used fitness function),

augmented — contains additional information not necessary connected with aspects of program evaluated by fitness function.

Table 3.1 compares the three types of semantics presented in the previous section in terms of the features discussed above and the features defined in the following section (equivalence testing and similarity measure). It is clearly visible that the canonical representation perfectly describes behavior of any program. However, it has one fundamental

drawback — it is hard to define a meaningful similarity measure for it (i.e., such that it correlates with the similarity of behavior/output of compared programs). On the other extreme, the fitness value is easy to compare, but it describes the program behavior very roughly, preventing a deeper investigation of program properties.

Between these two extremes lies the sampling semantics, which offers a reasonable compromise. On the one hand, sampling semantics are easy to compare, on the other hand, the precision of description can be adopted by simply increasing or decreasing the number of elements. Therefore, in following we concentrate only on this kind of semantics.

3.4. Definitions

In this thesis, we employ the variant of sampling semantics that is calculated from the fitness cases (i.e., training data that come with problem specification). This might suggest that we constrain our considerations only to problems which define a set of fitness cases, but this is not true. Indeed, this thesis presents only results obtained on such problems (all used benchmarks are presented in Chapter 4.2), however all proposed algorithms are generic and nothing stands in the way of applying these methods to other problems and, after defining appropriate metrics, with other forms of semantics.

In this section we present the necessary definitions related to the semantics in the form used in this thesis. Lets start with the general definition of fitness cases:

Definition 1. A *set of fitness cases* is a set of pairs

$$C = \{(x_i, y_i) : i = 1, \dots, N\},$$

where x_i and y_i are, respectively, the ith input datum and the correct (desired) result for these datum, and N is the number of fitness cases. In statistical terms, the former correspond to independent variables, while the latter to the dependent variables.

In general, both x_i and y_i may be either scalars, vectors, matrices, or anything else, whatever is appropriate for a particular problem.

Definition 2. A *sampling semantics* of a program p based on a set of fitness cases C is a list of actual results calculated by p:

$$s(p) = [s_i : s_i = p(x_i), (x_i, y_i) \in C],$$

where $p(x_i)$ is the result calculated by program p for input data x_i.

Alternatively, we could consider a form of sampling semantics that is not based on a set of fitness cases. In such a case, the values of x_i could be randomly drawn from an appropriate domain, and the number of elements of such semantics would not necessary

	Fitness case		*Program output*
i	x_i	y_i	s_i
1	-0.5	-0.125	**0.5**
2	1.0	1.000	**2.0**
3	1.5	3.375	**4.5**
4	2.0	8.000	**8.0**

$\Longleftarrow \quad p(x) = 2x^2$

$\Longrightarrow \quad \boldsymbol{s(p) = [0.5, 2.0, 4.5, 8.0]}$

Figure 3.4.1.: An example of symbolic regression problem (defined by a given set of fitness cases) and the calculated sampling semantics for program $p(x) = 2x^2$.

be the same as the size of C. However, in this thesis we will use only sampling semantics based on fitness cases, so $|s(p)| = |C|$.

For brevity, in following we will use terms 'semantics' and 'sampling semantics' interchangeably, but always in the meaning 'sampling semantics based on the fitness cases'.

We want to emphasize that sampling semantics as defined above is restricted to describe only the final output of a program, and it does not tell anything about *how* the result was obtained. Therefore, two programs calculating the same results but in completely different way cannot be distinguished by means of this form of semantics (see discussion about augmented semantics in Section 3.3).

Figure 3.4.1 shows an example of symbolic regression problem, an exemplary program, and the sampling semantics of that program. The problem is given by the set of fitness cases (determined by function $y = x^3$, which is however irrelevant here). The shown sampling semantics is calculated for program $p(x) = 2x^2$.

Definition 3. Two semantics s_1 and s_2 are considered equal, iff

$$\forall i,\ |s_{1,i} - s_{2,i}| < \epsilon,$$

where $s_{1,i}$ and $s_{2,i}$ are i^{th} elements of semantics s_1 and s_2, respectively, and ϵ is a small tolerance threshold.

An important consequence of this definition is that equality of sampling semantics is reflexive, symmetric, but not transitive. However, such approximate equality relation has been already proposed in literature [98, 99, 125] because it is very useful and sufficient in practice. In all our experiments we set $\epsilon = 1.11 \cdot 10^{-15}$ (the same value as used in ECJ [83] package) to eliminate the floating point accuracy problems. Thanks to this, we can safely assume that mathematically equivalent expressions have equal semantics.

To avoid separate definitions for Boolean domain, in following we will identify the logical values *true* and *false* with numerical values 1.0 and 0.0, respectively. In this way, the above equality definition is valid also for Boolean problems, and it says in practice that two Boolean semantics s_1 and s_2 are equal iff $\forall i,\ s_{1,i} = s_{2,i}$.

Definition 4. A *distance* $d(s_1, s_2)$ between two semantics s_1 and s_2 is

$$d(s_1, s_2) = \|s_1 - s_2\| = \sum_{i=1}^{N} |s_{1,i} - s_{2,i}| \,, \tag{3.4.1}$$

where $s_{1,i}$ and $s_{2,i}$ are i^{th} elements of semantics s_1 and s_2 respectively, and $|| \cdot ||$ is an assumed L_1 (city-block/Manhattan) distance metric.

Again, replacing logical values *true* and *false* with numerical constants 1.0 and 0.0, respectively, the distance defined above is valid for Boolean domain, and it is equivalent to the Hamming distance.

It is often convenient to describe algorithms using the concept of *similarity* between two semantics rather than distance between them. Founding a similarity measure on distance obviously requires an appropriate transformation. For the sake of the argument, we can assume that these two quantities are related in the following way:

$$similarity(s_1, s_2) = \frac{1}{1 + d(s_1, s_2)}. \tag{3.4.2}$$

However, the particular form of mapping distance to similarity is of secondary importance, as it is not explicitly used in the algorithms proposed in this thesis.

It is important to notice that in most genetic programming problems that define the set of fitness cases, the task may be defined in the form of *target semantics*. The target semantics may be defined as our sampling semantics, but it contains expected (correct) results instead of outputs from some real program. In other words, the target semantics is the sampling semantics of an ideal program (potentially only a hypothetical one) which solves a given problem perfectly.

For such problems, the *minimized* fitness of an individual is usually calculated as a difference between the semantics of the individual to evaluate and the target semantics t (cf. Algorithm 2.4):

$$f(p) = d(s(p), t). \tag{3.4.3}$$

Alternatively, if it is more reasonable, the *maximized* fitness value can be calculated as:

$$f(p) = similarity(s(p), t). \tag{3.4.4}$$

Therefore, it seems reasonable to use a definition of distance between two sampling semantics which is the same as the one used in a fitness function imposed by a problem. The methods proposed in this thesis neither require nor exploit this property, nevertheless, all benchmark problems have such compatible fitness function.

3.5. Summary

In this chapter we presented the concept of semantics in genetic programming. Possible formalizations of this notion and their properties have been discussed. From the wide range of possibilities we decided to concentrate on the sampling semantics, which is a compromise between precision (meant as the extent to which semantics reflects the behavior of a program) and the capability to analyze and compare such semantics.

The sampling semantics based on fitness cases, which will be used throughout this thesis, might be characterized as:

- partial — because the description might be incomplete if the set of fitness cases does not cover the whole domain (sampling semantics does not describe actual program output for other values of independent variables); however, for Boolean problems from our benchmark suite (see Section 4.2.2), the set of fitness cases contains all possible combinations of input variables, therefore, the sampling semantics for this case will be complete,
- accurate — values for all given fitness cases are calculated precisely (strictly speaking, up to the floating point precision provided by computer implementation),
- sufficient — it describes all aspects evaluated by a fitness function.

For sampling semantic, we can easily define two essential formalisms:

- equivalence relation — checked with an ϵ tolerance threshold,
- similarity function — measured as a sum of absolute differences between corresponding semantics elements.

Importantly, for any pair of semantics, verification/calculation of the above quantities can be done at low computational expense, linear with respect to the number of fitness cases.

4. Algorithm Evaluation Framework

4.1. Introduction

In the subsequent chapters of this dissertation, we propose a family of semantically extended GP algorithms that are expected to outperform standard GP methods. To verify this hypothesis, the algorithms will be compared on a representative sample of problem instances. Because the experimental framework we built for that purpose is common for all the proposed approaches, it is presented in this separate chapter, to avoid unnecessary repetitions.

This practice of benchmarking, meant as comparing algorithms on a representative sample of problem instances, is common in computational intelligence, and, as a matter of fact, the only reasonable one. Testing the algorithms on *all* problem instances is not only computationally infeasible, but also, according to the No Free Lunch theorem [136], has to be inconclusive, as the expected outcome of all optimization/learning algorithms on the entire universe of problems is exactly the same.

Unfortunately, there is a lack of strict standardization and appropriate rigor of benchmarking [88], and in consequence it is often difficult to compare different approaches. Therefore, within the last two decades, the community of GP researchers informally elaborated a set of benchmarks that became a *de facto* standard gauge of algorithm quality. These benchmarks can be grouped into several classes:

- Symbolic Regression — the aim is to fit a (generally real-valued) mathematical formula to a set of given measurements; the most commonly used problems are synthetic functions, especially polynomials like quartic,
- Predictive Modeling — tasks similar to symbolic regression, but here the goal is to create a function which will predict future values of a data series basing on historical values; *Mackey-Glass Chaotic Time Series* [76] is one of the more popular one,
- Classification — essentially equivalent to the paradigm of supervised classification via learning from examples in machine learning; the task is to classify objects into a predefined set of (two or more) decision classes; this is usually done by evolving an expression whose output value (possibly after thresholding) indicates the resultant class identifier; tasks from the UCI repository [29] are often used here,

- Boolean (Logic) Function Synthesis — the aim is to synthesize a logic expression (Boolean circuit, usually acyclic) consistent with a given truth table; the *even parity* or *multiplexer* [60] are the most popular benchmarks,
- Path Finding and Planning — the goal is to plan a path and to control some agent in a way which maximize some criteria; the most often used problem is the *Artificial Ant* [60].

Less popular, but also found in literature, benchmarks include: a class connected with traditional programming like sorting or automatic bug fixing, specially constructed problems testing some aspects of investigated methods like *Max* problem.

In this thesis, presented algorithms are evaluated on 39 different problems belonging to the two most popular classes of benchmarks: symbolic regression and Boolean functions. According to McDermott *et al.*, these two classes of benchmarks are used, respectively, in about 33% and 15% of survived 172 articles in [87]. Following sections describe those problems in details and present evolutionary parameters which are commonly used in our experiments. In particular, in Section 4.4 we provide experimental results which suggest that selecting fitness cases randomly for each evolutionary run is not a good practice, although such a procedure is common in the GP community. Based on these results, we decide to use a fixed set of fitness cases in the subsequent chapters. The last section of this chapter introduces criteria used to evaluate performance of proposed algorithms and presents statistical tools used to compare algorithms with each other.

4.2. Benchmark Suite

4.2.1. Symbolic Regression Domain

The set of symbolic regression problems used in this thesis is presented in Table 4.1. Problems F01–F12 are borrowed from Nguyen paper [126] (half of them Nguyen took from [44, 54, 50]), and the rest, except the problem R0 introduced here, come from Krawiec and Wieloch work [72]. Within these, the F02–F04 problems are the most widely studied within GP, so-called *quartic*, *quintic*, and *sextic* polynomials [60, 61]. The table shows the 'hidden' equation to discover (*Target program*), the number of independent variables (*Vars*), the range from which they are chosen (*Range*), and the number of fitness cases (points) both for training and testing sets.

A program is treated as an ideal solution if it returns correct values for each fitness case from the training set with a $1.11 \cdot 10^{-15}$ tolerance (i.e., if its sampling semantics is equal to the target semantics; *cf.* Definition 3 on page 34 of sampling semantics equality). This tolerance threshold is introduced to deal with the floating point imprecision. Without this assumption, even an expression mathematically equivalent to the target program could be found non-optimal (i.e., having a non-zero value of minimized fitness function).

	Target program (expression)	*Vars*	*Range*	*Training*	*Testing*
F01	$x^3 + x^2 + x$	1	$[-1; 1]$	20	200
F02	$x^4 + x^3 + x^2 + x$	1	$[-1; 1]$	20	200
F03	$x^5 + x^4 + x^3 + x^2 + x$	1	$[-1; 1]$	20	200
F04	$x^6 + x^5 + x^4 + x^3 + x^2 + x$	1	$[-1; 1]$	20	200
F05	$\sin(x^2)\cos(x) - 1$	1	$[-1; 1]$	20	200
F06	$\sin(x) + \sin(x + x^2)$	1	$[-1; 1]$	20	200
F07	$\log(x + 1) + \log(x^2 + 1)$	1	$[0; 2]$	20	200
F08	$\sqrt{x}$	1	$[0; 4]$	20	200
F09	$\sin(x) + \sin(y^2)$	2	$[0.01; 0.99]$	100	10000
F10	$2\sin(x)\cos(y)$	2	$[0.01; 0.99]$	100	10000
F11	x^y	2	$[0.01; 0.99]$	100	10000
F12	$x^4 - x^3 + y^2/2 - y$	2	$[0.01; 0.99]$	100	10000
P1	$x^6 - 2x^4 + x^2$	1	$[-1; 1]$	20	200
P2	$x^7 - 2x^6 + x^5 - x^4 + x^3 - 2x^2 + x$	1	$[-1; 1]$	20	200
P3	$x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x$	1	$[-1; 1]$	20	200
R0	$(x - 1)/(x^2 + 1)$	1	$[-1; 1]$	20	200
R1	$(x + 1)^3/(x^2 - x + 1)$	1	$[-1; 1]$	20	200
R2	$(x^5 - 3x^3 + 1)/(x^2 + 1)$	1	$[-1; 1]$	20	200
R3	$(x^6 + x^5)/(x^4 + x^3 + x^2 + x + 1)$	1	$[-1; 1]$	20	200

Table 4.1.: Symbolic regression functions used in experimental part of this thesis. The columns present the number of independent variables with ranges of allowed values, and numbers of fitness cases in both training and testing set.

The points of fitness cases are evenly distributed in appropriate domain. This means that the values are evenly spaced in a given closed interval shown in Table 4.1, with the extreme values placed on the interval boundaries. For univariate problems, this implies that the difference between any two consecutive points equals $(b-a)/(k-1)$ for k points in range $[a;b]$.

In case of bivariate functions the values of both variables lie on an evenly spaced square lattice. For instance, in case of 100 points, both x and y have one from 10 possible, equally spaced, values. This, however, may cause problems. For that instance, if the variable ranges were $[0;1]$ for F11, then a substantial number of fitness cases (nearly 40%!) would hit on special arguments of the objective functions (i.e. 0^y, 1^y, x^0, or x^1). This may render evolution unable to escape from even very simple local optima. Therefore we slightly narrowed the original $[0;1]$ interval to $[0.01;0.99]$.

The problem mentioned above does not exists in the original problem formulation with $[0;1]$ interval as in paper [126] because Nguyen *et al.* (as most researchers) have used randomly selected points uniformly distributed in this range. However, as shown in Section 4.4, we have strong evidence that such selection of fitness cases is not a good practice and therefore we decided to use evenly distributed points.

Testing set is created analogously to training set, i.e., the points are also evenly, but are much more densely distributed in the same domain. For this reason, both training and testing sets are not disjoint. From the specific ranges and the number of fitness cases (presented in Table 4.1) it appears that for all univariate problems exactly two points (the extreme values from interval boundaries) are common for training and testing sets. However, in case of bivariate problems, the testing set contains all training fitness cases (and 9900 others).

For univariate problems, the terminal set contains two elements: x — the independent variable, and a constant 1.0. For bivariate problems there are two terminals: x and y — the independent variables, without the constant 1.0. Though the lack of constants for bivariate problems may seem surprising, let us note that it has been shown many time that GP fares pretty well without any constants at all, as evolution can easily come up with the idea of using subexpressions like x/x or $x-x$ instead of constants.

The set of non-terminal instructions consists of eight functions: $+$, $-$, $\times$, $/$ (protected), sin, cos, exp, log (protected). The protected version of division returns 1.0 if the denominator equals zero, irrespective of the numerator. The protected version of logarithm returns 0.0 if its argument equals zero, otherwise it returns the logarithm of the absolute value of its argument.

Let us note that the provided set of terminal and non-terminal instructions allows expressing all target functions presented in Table 4.1. In other words, for every benchmark problem, an optimal solution is present in the considered solution space.

4.2.2. Boolean Domain

In the Boolean domain, four different problems will be used: even parity, multiplexer, majority, and comparator. First three of them come from Koza's book [60], and the last one is a simplified version of a digital comparator proposed in Walker and Miller paper [128].

In following, for Boolean problems, we will use the term 'argument' or 'bit' in the same meaning as 'independent variable' was used in the symbolic regression context in the previous section.

The objective of *even parity* (PAR) problem is to synthesize a function which returns *true* if and only if an even number of its arguments are *true*. PAR can be alternatively seen as a generalization of the not-exclusive-or function to more than two arguments. In this thesis, specific instances of this benchmark problems will use 4, 5, and 6 bits (i.e. with 4, 5, or 6 input arguments), and will be denoted as PAR4, PAR5, and PAR6.

In *multiplexer* problem, program arguments are divided into two blocks: address bits and data bits. The goal is to interpret the address bits as a binary number and use that number to index and return an appropriate data bit. We consider two variants of this problem — 6-bit (MUX6) and 11-bit (MUX11). In the former we have 2 address bits and 4 data bits. In the latter — 3 and 8 bits, respectively.

The task in *majority* problem is to create a function that returns *true* if more than half of input arguments are *true*. Note that for even number of arguments, the function should return *false* if exactly half (or less) of them are *true*. We consider three variants of this problem: with 5 bits (MAJ5), 6 bits (MAJ6), and 7 bits (MAJ7).

The last Boolean problem used in this thesis is *comparator*. The objective here is to bisect arguments into two equally-sized subsets, interpret them as two binary integer numbers, and return *true* only if the first number is greater than the second one. We used six (CMP6) and eight (CMP8) bits variants, which means that we compare 3-bits numbers (CMP6) or 4-bits numbers (CMP8).

In Table 4.2, all ten problems are shown together with the number of fitness cases on which solutions will be evaluated. Because the training set contains all possible combinations ($2^{numberOfBits}$), there is no testing set. A solution is considered as an ideal only if it returns correct result for all fitness cases.

A set of terminals used in Boolean domain experiments contains one terminal for each input bit. For this domain, we consider two function sets (non-terminal instructions): {AND, OR, NOT, IF} and {AND, OR, NAND, NOR}. If the first set is used, the name of a problem is prefixed with "I", if the second set — with "N". In this way, we have 20 different problems in total:

- Using AND, OR, NOT, and IF functions: IPAR4, IPAR5, IPAR6, IMUX6, IMUX11, IMAJ5, IMAJ6, IMAJ7, ICMP6, ICMP8,

Problem	*Instance*	*Bits*	*Fitness cases*
even parity	PAR4	4	16
	PAR5	5	32
	PAR6	6	64
multiplexer	MUX6	6	64
	MUX11	11	2048
majority	MAJ5	5	32
	MAJ6	6	64
	MAJ7	7	128
comparator	CMP6	6	64
	CMP8	8	256

Table 4.2.: Boolean problems.

- Using AND, OR, NAND, and NOR functions: NPAR4, NPAR5, NPAR6, NMUX6, NMUX11, NMAJ5, NMAJ6, NMAJ7, NCMP6, NCMP8.

Considering two function sets has historical origins. In Koza's book [60], only the multiplexer benchmark uses the first function set containing the IF function, and the others use the second set with negation of AND and OR. However, for completeness, we use both sets for all problems.

Similarly to regression problems, also for this domain solutions to all aforementioned problems can be found in the assumed solution space.

4.3. Evolutionary Parameters

Experiments shown in this thesis, are based on the canonical genetic programming with appropriate modifications introduced by the proposed methods. When not stated otherwise, parameters of the evolution used in these experiments are shown in Table 4.3. These parameters are based on Nguyen's work [98, 99, 125]. Parameters not mentioned by Nguyen, are taken from ECJ [83] package and are based on parameters originally used by John Koza [60, 61].

Most experiments presented in the remaining part of this thesis will aim at assessing the influence of various search operators on the conduct and final outcome of evolutionary search. Therefore, different proportions of the operator in question and one of the standard crossover or mutation operators (playing the role of auxiliary operator) will be examined. These proportions will be controlled by corresponding probabilities of operator engagement. The proportions of the two operators will vary from 0.0 (the auxiliary operator will not be used) to 1.0 (the analyzed operator will not be used) with step 0.1. The setup with proportion 1.0 will usually correspond to standard GP, as only crossover or mutation will be used.

Parameter	*Value*
Generations	100
Population size	500
Initialization method	Ramped Half-and-Half (Algorithm 2.3)
Initial minimal depth	2
Initial maximal depth	6
Duplicate retries	100 (before accepting a syntactic duplicated individual)
Selection method	tournament
Tournament size	3
Operators probability	varying from 0 to 1 with step 0.1
Maximal program depth	15
Node selection	probability of terminal nodes: 10% probability of non-terminal nodes: 90%
Mutation method	subtree mutation
Subtree builder	Grow (Algorithm 2.1)
Subtree depth	5
Crossover method	subtrees swapping
Instructions	*symbolic regression*: see Section 4.2.1 *Boolean domain*: see Section 4.2.2
Successful run	*symbolic regression*: error on each fitness case $< 1.11 \cdot 10^{-15}$ *Boolean domain*: perfect reproduction of all fitness cases
Number of runs	200 or 1000 (depending on experiment)

Table 4.3.: Evolutionary parameters.

Problem	*min*	*median*	*max*	*IQR*	*pvalue*
F01	0.1450	0.9300	1.0000	0.2350	0.0000
F02	0.2350	0.6500	0.8950	0.3000	0.0000
F03	0.0200	0.4750	0.6900	0.2825	0.0000
F04	0.0400	0.1675	0.4150	0.1113	0.0000
F05	0.0000	0.0200	0.0700	0.0163	0.0000
F06	0.0850	0.3350	0.7850	0.3425	0.0000
F07	0.0100	0.0400	0.1300	0.0250	0.0000
F08	0.0300	0.0900	0.1600	0.0350	0.0015
F09	0.1150	0.2200	0.3450	0.0700	0.0000
F10	0.0900	0.1550	0.2350	0.0350	0.0336
F11	0.1400	0.2075	0.2900	0.0363	0.0874
F12	0.0000	0.0000	0.0000	0.0000	—
P1	0.0000	0.0050	0.0250	0.0100	0.5039
P2	0.0000	0.0000	0.0200	0.0050	0.0005
P3	0.0000	0.0100	0.0750	0.0150	0.0000
R0	0.0000	0.0050	0.0250	0.0100	0.0939
R1	0.0000	0.0000	0.0000	0.0000	—
R2	0.0000	0.0000	0.0000	0.0000	—
R3	0.0000	0.0000	0.0100	0.0000	0.0094

Table 4.4.: Statistics on the success rates for each problem and for different sets of fitness cases. IQR is the interquartile range, and the last column shows the p-value of performed statistical test for testing the hypothesis that the probability of success for each set are the same.

4.4. The Importance of Fitness Case Assortment

Especially for the real-valued symbolic regression problems, the practice commonly adopted in GP is to sample the fitness cases uniformly from a given range for each of independent variables. However, we will show in the following that the choice of concrete values is crucial — two tasks with the same target function but different fitness cases, even chosen from the same range, may lead to extremely different hardness of such problems. Obviously, this does not concern our Boolean benchmarks for which, as it is usually done in practice, all possible fitness cases are used for fitness assessment.

We demonstrate that problem hardness depends heavily on the chosen set of fitness cases by testing how the choice of fitness cases impacts the odds of standard GP algorithm finding the optimal solution. For this purpose we generate randomly 100 alternative sets of sampled values (i.e., 100 different sets, each with 20 random fitness cases) from appropriate ranges (see Table 4.1), and then for each such set and for each symbolic regression target function (Table 4.1) we perform 200 independent evolutionary runs to estimate the success rate (thus $19 \times 100 \times 200 = 380000$ runs in total). The evolutionary parameters are set according to Table 4.3. In this experiment, the probability of standard crossover and mutation operators was fixed to 0.9 and 0.1, respectively, which is a common setting, also used in the ECJ library.

Table 4.4, for each symbolic regression target function, presents the range of achieved

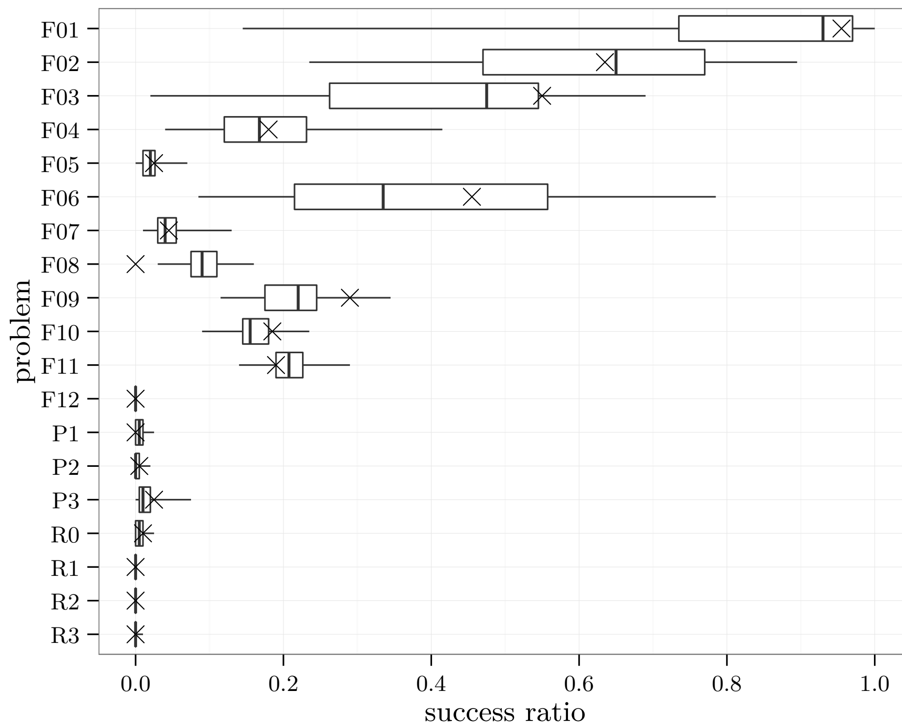


Figure 4.4.1.: Five-number summary of success rates for different sets of independent values (i.e., the smallest observed success rate, first quartile, the median, the third quartile, and the maximum). Additionally, the cross mark denote a success rate for evenly distributed fitness cases.

success rates depending on the chosen set of fitness cases. Additionally, we test statistically (a special test of equal binomial proportion [97]) the null hypothesis that the probabilities of success are the same regardless of the used set of fitness cases — the calculated p-values are also shown in the table. Figure 4.4.1 presents graphically the ranges of success rates varying with different random set of points.

This experiment demonstrates that for 12 problems out of 19 the choice of fitness cases influences the success rate in a statistically important degree (with $\alpha = 0.01$). For example, the problem F01 can be very easy (with success rate equal 100%) or much harder (with success rate only about 15%), depending just on how the used pseudorandom number generator happened to behave in the process of generating the training fitness cases. For two problems (F10 and F11) we cannot reject the null hypothesis despite the fact that the range of success rate is about 15 percent points wide. It appears that seven problems (F12, P1, P2, R0, R1, R2, R3) are quite difficult to solve by the standard genetic programming (best success ratio is less than 3%). For five of them (F12, P1, R0, R1, R2), we also cannot reject the null hypothesis (at least for the used experimental setup). For problems F12, R1, and R2, evolution does not find even a single ideal solution in 200 runs for any of the

100 variants of those problems. Note however, that this does not invalidate GP as a useful technique for *approximating* solutions for these problems.

Table 4.5 shows influence of proportion between crossover and mutation operators on the obtained success ratio (in contrast to the previous experiment carried out for crossover and mutation probabilities fixed at 0.9 and 0.1, respectively). As we can see, most of the presented symbolic regression problems are easier to solve when the probability of crossover operator is very high. An exception is problem F08, for which the success rate is similar irrespective to this proportion, and problem F11 which seems to be the easiest when mutation is applied with about 50% probability.

In the GP community, it is common to overcome the above problem of varying task difficulty by selecting fitness cases randomly for each independent run of an examined algorithm. However, this implies that each evolutionary run solves in fact a completely different problem and, what is more, this makes it incorrect to aggregate some statistics from such runs. For example, the fitness of the same individual can substantially vary when tested on different variants (sets of fitness cases) of the same problem.

It is even easier to notice that such approach is flawed when the target function cannot be constructed from the available functions and terminals (i.e., the solution space does not contain the exact solution to a problem). In this case, the fitness values obtained by the best possible solution (which approximates the target function best) vary depending on the selected fitness cases. In other words, the fitness values assigned to the same individuals by the fitness functions that use different sets of fitness cases are incomparable. By this token, it does not make sense to compare fitness values of, e.g., the best-of-run individuals resulting from different runs, not mentioning aggregating their performance.

Therefore, to avoid this conundrum and make problem definition unambiguous, in this thesis, instead of drawing fitness cases at random, we decided to evenly distribute the points. In this way, we also make reproduction of presented experiments much easier.

A practical conclusion from the above observations is that in GP, a task should be identified with a target function *and* an associated set of fitness cases. Defining a benchmark by target function alone leads to large variance of algorithm performance and incomparability of performance across runs.

4.5. Evaluation Criteria

4.5.1. Objectives

This section describes statistics which we employ to evaluate and compare algorithms tested on the benchmark suite presented in Section 4.2. The experiments included in following chapters employ, where appropriate, most of these measures. Next section presents statistical tests and procedures used to analyze these data.

In following, as a final result obtained from an evolution we will treat the *best-of-run*

	P(xover)	*100*	*90*	*80*	*70*	*60*	*50*	*40*	*30*	*20*	*10*	*0*
	min	12	14	21	20	23	22	24	20	20	18	12
F01	median	93	93	90	86	79	70	60	53	45	38	29
	max	100	100	99	98	93	86	80	69	64	50	41
	min	24	24	16	16	11	11	9	7	4	4	2
F02	median	71	65	59	46	38	29	22	17	14	10	5
	max	94	90	84	73	60	44	38	28	21	18	11
	min	2	2	3	4	2	2	2	0	0	0	0
F03	median	60	48	40	25	16	10	7	4	3	2	0
	max	80	69	60	45	30	21	16	8	6	4	2
	min	4	4	1	1	1	0	0	0	0	0	0
F04	median	22	17	13	8	5	3	2	1	0	0	0
	max	52	42	31	20	12	8	6	5	2	2	1
	min	0	0	0	0	0	0	0	0	0	0	0
F05	median	2	2	2	2	1	1	2	0	1	1	1
	max	4	7	4	6	4	5	4	3	4	4	3
	min	10	8	7	6	4	4	2	2	1	1	0
F06	median	35	34	29	26	22	16	12	9	6	5	3
	max	80	78	67	60	42	32	22	18	14	10	8
	min	1	1	0	0	0	0	0	0	0	0	0
F07	median	5	4	3	2	2	2	1	1	1	0	0
	max	14	13	10	6	4	4	4	3	2	2	2
	min	6	3	5	6	4	3	4	2	4	3	3
F08	median	11	9	10	10	10	9	9	8	10	8	8
	max	16	16	16	18	17	16	20	16	18	16	14
	min	11	12	10	6	6	5	4	4	4	2	1
F09	median	23	22	19	17	14	12	11	8	8	6	4
	max	36	34	36	24	22	24	20	14	15	12	10
	min	10	9	8	9	6	4	4	4	2	2	2
F10	median	18	16	14	13	11	8	8	7	6	5	4
	max	27	24	21	18	18	14	13	11	10	8	8
	min	11	14	17	16	16	20	24	20	18	16	16
F11	median	18	21	25	24	26	28	30	27	26	22	24
	max	27	29	34	32	36	42	36	38	34	32	35
	min	0	0	0	0	0	0	0	0	0	0	0
F12	median	0	0	0	0	0	0	0	0	0	0	0
	max	0	0	0	0	0	0	0	0	0	0	0
	min	0	0	0	0	0	0	0	0	0	0	0
P1	median	0	0	0	0	0	0	0	0	0	0	0
	max	4	2	2	2	1	2	0	2	1	0	0
	min	0	0	0	0	0	0	0	0	0	0	0
P2	median	0	0	0	0	0	0	0	0	0	0	0
	max	3	2	2	1	1	1	0	0	0	0	0
	min	0	0	0	0	0	0	0	0	0	0	0
P3	median	2	1	0	0	0	0	0	0	0	0	0
	max	16	8	7	4	2	1	1	0	0	0	0
	min	0	0	0	0	0	0	0	0	0	0	0
R0	median	1	0	1	0	1	0	0	0	0	0	1
	max	4	2	3	3	3	2	2	3	3	3	3
	min	0	0	0	0	0	0	0	0	0	0	0
R1	median	0	0	0	0	0	0	0	0	0	0	0
	max	0	0	0	0	0	0	0	0	0	0	0
	min	0	0	0	0	0	0	0	0	0	0	0
R2	median	0	0	0	0	0	0	0	0	0	0	0
	max	0	0	0	0	0	0	0	0	0	0	0
	min	0	0	0	0	0	0	0	0	0	0	0
R3	median	0	0	0	0	0	0	0	0	0	0	0
	max	0	1	0	0	0	0	0	0	0	0	0

Table 4.5.: Comparison of success rates for different proportion of crossover and mutation operators ($P(mutation) = 100\% - P(crossover)$). All shown values are in percentage (%).

individual. In other words, the algorithm returns a single program with the best fitness value, whenever it appears in the whole evolutionary process. This is quite important notice, because sometimes only the best individual from the last generation (*best-of-last-generation*) is deemed as a final outcome of evolution. However, in this thesis, it does not matter when the returned program appears. Therefore, there is no risk that the best evolved individual will be forgotten.

Success Ratio is the ratio of successful runs to all performed runs. It is a natural estimate of the probability of finding the ideal solution in a single evolutionary run. Because the stochastic nature of an evolution, each methods will be tested in many (at least 200) independent evolutionary runs to estimate this statistics.

The success ratio is the objective of high practical importance, as it tells about algorithm's ability of successfully solving a given problem, i.e., finding a program which fulfills the problem's requirements. In the case of symbolic regression, this means that the evolved program has to give an ideal result within a $1.11 \cdot 10^{-15}$ tolerance for each fitness case from the training set. In case of Boolean problems, the program has to return perfect output for all fitness cases.

Error is the sum of disparity between a program actual result and the proper output for each fitness case from a *training set*. It is calculated according to Equation 3.4.1 as a distance between semantics of a program and the target semantics. The error shows how well a given algorithm approximates the ideal solution.

Again, if the difference on a single fitness case is less than $1.11 \cdot 10^{-15}$, it is supposed to be caused by floating point rounding errors, and such difference is interpreted as zero. Therefore, programs recognized as ideal always have *error* (as defined here, i.e., calculated on training set of fitness cases) equal to zero. The error value of an individual is treated as its minimized fitness used by the evolution.

Testing error is the error calculated in analogous way (i.e., also by means of Equation 3.4.1), however on an additional *testing set*. This measure is used to check whether a program is overfitted.

The testing error is not applicable for Boolean problems, because for them the training set contains all possible fitness cases. In this thesis, the testing sets have 10 times more testing cases for all univariate symbolic regression problems and 100 times more for bivariate problems (10 times more values for each variable). The testing sets are generated analogously to training sets (see Section 4.2.1). Therefore, due to the even distribution of test cases, the training and testing sets are not disjoint (exactly 1% of testing cases belongs to the training set too).

Number of hits counts the number of fitness cases (either from *training* or *testing* set) which are solved 'well enough'.

This auxiliary performance measure has been originally introduced by Koza [60] and used in GP for a long time. It is justified by the practical perspective, where, depending on application context, it may be sometimes better to have a solution that performs reasonably well on all fitness cases than a solution that behaves perfectly on some of them while committing large errors on the others. For symbolic regression problem, a fitness case is considered to be correct (a 'hit') if the value of the dependent variable produced by a program comes within 0.001 of target value. For Boolean domain, *hits* counts the number of correct results — the dependent variable has only two possible values (*false* and *true*).

Mean size is the average number of nodes (both functions and terminals) of individuals in a population.

The tendency of inexorably growing programs generated by GP without corresponding increase of fitness is known as *code bloat effect* and has been intensively studied in the GP literature [11, 100, 89, 116, 73, 23]. This process is generally perceived as having negative impact on the search (genetic modifications of large programs are likely to be ineffective, and large programs are more costly to execute) and making the resultant programs difficult to comprehend. Observing the mean size enables to monitor the growth of evolved programs and detect code bloat.

Best-of-run size is the size of the best-of-run program.

This is an important factor indicating the quality of achieved results because a small solution, in the sense of program length, is in general highly desired. A smaller program is cheaper to produce and runs faster in comparison to larger and more complicated one.

Time required to execute the evolutionary process (run) is an important criterion saying about method performance and computational efficiency.

In all experiments, the computations were performed on 16 PCs equipped in identical hardware. However, the measured times may be inaccurate as the computations were run on many computers simultaneously with other processes requiring the processor time and other system resources. Despite these not fully reliable conditions, the times averaged from many runs are still informative and give some sense of practical algorithms efficiency. However, due to certain optimization techniques employed in our implementation, this performance indicator requires additional comments which we provide in the subsequent Section 4.5.2.

Success per hour says how many successful runs are expected if a GP using given setup would be allowed to run for one hour, starting a new evolutionary run after the previous one has been completed.

This is much better measure of efficiency than the mean execution time, because a

method that executes very fast does not necessary score many successes. However, if a method is faster, then, given a fixed budget of computation time (one hour in this case), it can be run more times (which can be alternatively seen as restarting the evolutionary process with different random number generators). This could potentially improve an assessment of such method, but if the method is poor then the successive runs do not help much.

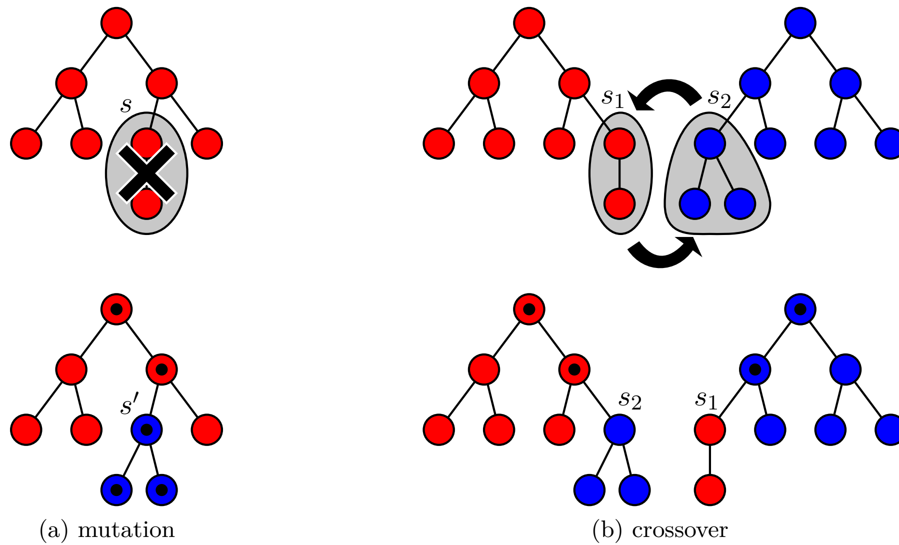
Diversity — the number of individuals in a population either with distinct semantics or distinct fitness values. Diversity shows which method promotes higher variety in population. Observing how the diversity is changing during the evolution could help to explain the performance of compared methods [15, 14].

4.5.2. Optimization of Program Execution

It is necessary to remember that measured evolution runtime (c.f. criterion *Time* above) depends not only on method performance, but also on its implementation. In this context, it is important to mention that our implementation, to avoid repeating the same computations, intensively uses a cache to save the output of each internal node of a program once it has been calculated. This means that each subprogram is processed only once during the entire evolutionary run, no matter in how many solutions it appeared as a result of genetic modifications (as long as it has not been modified). This gives extreme speedup in computation time (evaluation is even six times faster, however the overall evolution runtime has shorten up to three times). This happens, however, at the expense of much higher memory usage, which sometimes may have negative impact on the computation time, especially when the available memory runs out.

Let us illustrate the benefits of this optimization with two simple examples. Firstly, consider a new individual created by the standard, subtree mutation operator, in which a randomly selected subtree s has been replaced by a newly created random subtree s' (see Figure 4.5.1a). Normally (i.e., when using standard GP implementations like the one offered by ECJ), such individual would have to be executed entirely, on all fitness cases, to assess its fitness. However, within the proposed optimized scheme, only the subtree s' and nodes on the path from the location of s' to the root node are executed.

In case of crossover operator, a randomly chosen subtrees s_1 and s_2 are swapped. Because we remember output of all unmodified subtrees, each created offspring requires computation only on the path from the crossover point up to the root (see Figure 4.5.1b). This means, that the number of nodes which require recalculations is only $O(log\, n)$ order comparing to not optimized $O(n)$ (where n is the number of nodes in an individual). Hence, a control experiments which are using only mutation and crossover operators without any other mechanism have a big advantage to other methods performing some more complicated computations which are not so prone to such simple optimization.



(a) mutation (b) crossover

Figure 4.5.1.: Optimized version of standard breeding operators. In each offspring, only the nodes marked with black dots need to be executed.

4.5.3. Statistical Tools

To determine whether one method is better than others, an appropriate statistical tools should be used. The well-known procedure for testing differences between many related samples is the one-way Repeated Measures ANOVA (or within-subjects ANOVA). However, ANOVA assumes that the dependent variable is both continuous and normally distributed, and that the sphericity condition is not violated (i.e., variances of the differences between all combinations of tested algorithms must be equal).

Because, we want to run statistical tests for data violating these conditions, we will use the Friedman test [31, 30], which is a non-parametric alternative to the one-way ANOVA with repeated measures. For a set of k samples, this nonparametric test checks if at least two of them represent populations with different median values. The null hypothesis says that medians of all samples are equal, and the alternative hypothesis says that this is not true. These k groups of independent variable, corresponding to methods/algorithms in our case, are traditionally referred as treatments (also conditions or within-subjects factor), and the set of benchmark problems corresponds to subjects in the design of Friedman test.

In this test, a ranking of all compared algorithms has to be prepared. The procedure of calculating ranks has two steps:

1. For each problem independently, create a ranking of algorithms on this problem with respect to the assumed quality indicator (see Section 4.5.1). The best method has rank one; for tied values assign the average ranks of all tied algorithms.

2. For each algorithm, average the ranks obtained in the previous step for all considered problems.

It is important to be aware, that the produced ranks are averaged over all problems. Moreover, in case of ties, i.e., if several algorithms have equal values for a given problem then the assigned ranks are averaged and all of them get the same rank. Therefore, the best algorithm (not worse on any problem than other algorithms) might have the rank greater than one. For example, if two algorithms A and B perform equally on one problem but algorithm A is better than B on a second problem, then eventually method A will get rank $1.25 = (1.5+1)/2$ and method B — $1.75 = (1.5+2)/2$. Such ties happen very often, especially for easier problems where most algorithms are able to find an ideal solution, or for hard problems where most of them fail.

Nevertheless, such ranking can be used to sort methods from the best one to the worst. However, even if the Friedman test says that there is a statistical difference between some of the algorithms, an additional analysis is necessary to determine which one is significantly better than another.

To compare a single method with all other algorithms, Holm's post-hoc statistical procedure [46] will be used as suggested by Derrac *et al.* in [21]. This procedure calculates the adjusted p-values, allowing the comparison between an algorithm in question and all others. However, to check *all* pairs of methods, another post-hoc procedure should be used. For this purpose, Garcıa *et al.* [33] strongly encourage the use of Shaffer's static procedure [114], which benefits from using information about logically related hypothesis.

In a simpler case, when only two methods are compared with each other on several benchmarks, the paired Wilcoxon test will be performed instead of the Friedman test.

The Friedman test or paired Wilcoxon test will be used to compare algorithms performance (e.g. success ratio or error) tested on *all* benchmarks. However, to check statistical significant difference between two methods on the same problem, other tests are more appropriate. In such situation, criteria like error will be compared using the Mann-Whitney U test (also known as Wilcoxon rank-sum test).

However, for comparison of two success ratios, a more suitable, special test of equal proportion [97] will be performed. This test has the null hypothesis that the probabilities of success in several groups are the same. This test is used to compare success ratio in Section 4.4 and 5.2.

5. Semantically-oriented Search Operators

5.1. Introduction

Chapter 3 of this thesis provided rationale for using semantic information in GP. In this chapter, we review the past work on semantically-aware GP and present results of experiments performed on our benchmark suite. The purpose of this part of dissertation is threefold: (i) to verify the results published in related methods, using our experimental methodology and software framework, (ii) to determine optimal probability of applying those methods, and (iii) to provide reference results for the methods presented in subsequent chapters.

The semantic information may be used in many different ways. In this chapter, methods will be presented that replace various components of the standard evolutionary scheme of genetic programming, described in Section 2.2. The components that potentially can be modified to utilize semantic information include: population initialization, selection procedure, and search operators, i.e., crossover and mutation. Because the scope of modifications required by any of the methods proposed in this chapter is limited to a single component, these methods can be quite simply incorporated into existing evolutionary frameworks. Other, more sophisticated methods employing semantics, which engage multiple components of evolutionary algorithm, are described in Chapters 6 and 7.

To remind, the semantics used in further part of this thesis, if not stated otherwise, will have the form of sampling semantics described earlier (Section 3.4). Sampling semantics is a vector of outputs produced by a program in response to a predefined set of inputs (fitness cases). Such semantics is partial, so two different programs with the same sampling semantics may produce different outputs for some other input data not included in the set of training fitness cases. We will not extend these semantics with any additional information to keep the whole framework as simple as possible and, more importantly, to investigate the influence of semantics as such, apart from other elements which potentially may be very specific for a particular problem.

5.2. Population Initialization

5.2.1. Overview

Genetic programming, as most population-based search/optimization algorithms, belongs to the category of global search algorithms. In contrast to local methods, such algorithms are assumed to perform simultaneously exploration and exploitation of the search space. There are two prerequisites for exploration: appropriate initialization of the search process, and effective variation mechanisms during the search process. This section focuses on the former one.

In more straightforward problems of optimization and learning, population initialization is not an issue. For instance, evolutionary strategies that perform search in continuous vector spaces do not require sophisticated initialization procedures. With GP however, providing sufficient diversification in the initial population can be challenging. The reason is the many-to-one genotype-phenotype mapping, which causes some semantics to occur much more often than others when using standard initialization procedures like Ramped Half-and-Half (RHH) (see Algorithm 2.3 on page 18). For instance, Langdon showed [74] for the Boolean domain that, particularly for large program trees, some semantics are very easy to generate while others are close to impossible to generate.

Our experiment confirms Langdon's observations. We generated one million random programs using the RHH procedure with depths limit set from 2 to 15, and counted how many times each unique semantics appeared. We repeated this procedure for several different instruction sets used by different problems from the benchmark suite presented in Section 4.2. The calculated numbers for the first 1000 most frequent semantics are shown in Figure 5.2.1. It is important to notice, that the frequencies are presented on a logarithmic scale. As it is easily visible, generating one from the 100 most frequent semantics is several orders more probable than others. About 28%–65% of all generated programs, depending on the instruction set, have one of the 100 most frequent semantics (and 18%–52% — one from only 10 most frequent!). Moreover, most semantics have hardly any chance to be generated by RHH procedure (more than 90% of all generated semantics, for each instruction set, appeared only once for one million programs).

Population initialization is then an important and nontrivial stage of any GP algorithm, which deserves attention and offers potential benefits. For these reasons, we examine it thoroughly in this section.

There are several works devoted to the process of creating an initial population in genetic programming. Most of them concentrate on building syntactically diversified individuals in general [60], or on their particular aspects like diversity of program structure or shape [16, 12, 84].

Relatively few papers bring up behavioral aspects of created individuals. Looks in his study [81] analyzes behavioral distribution of randomly generated programs, and points

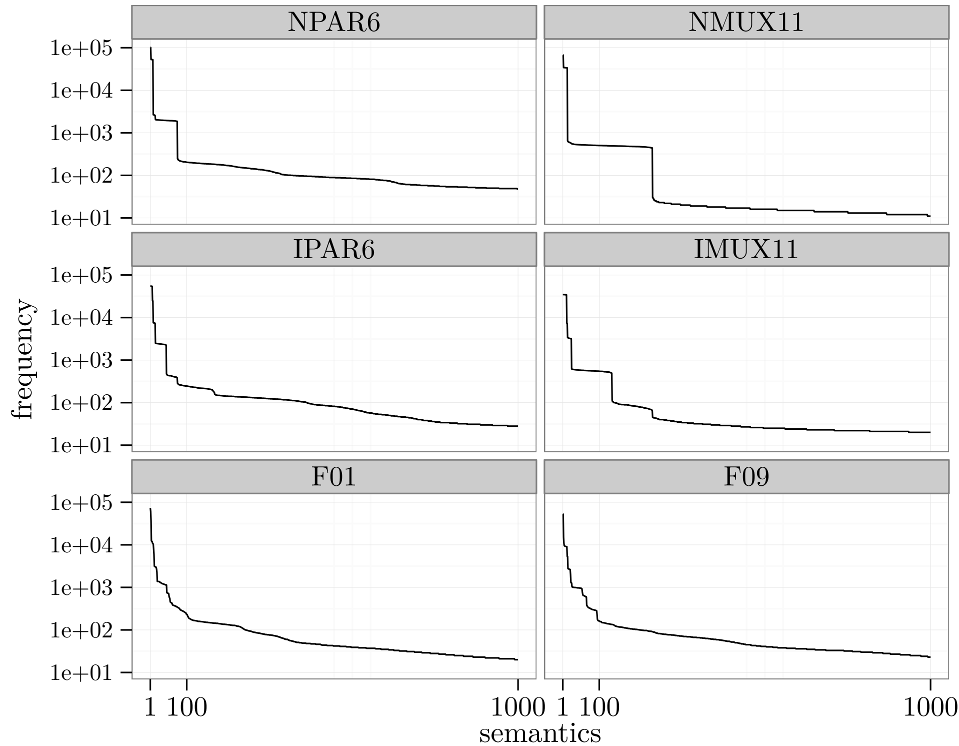


Figure 5.2.1.: Frequencies of the most 1000 common semantics of one million programs generated by the Ramped Half-and-Half procedure with depths limit set from 2 to 15. The set of used instructions depends on the problem.

out that, because of a highly skewed distribution of sampled programs, probability of generating some program decreases exponentially with the growing complexity of its behavior (defined as a minimal required program length).

Motivated by this observation, Looks proposed a heuristic for sampling programs that tries to approximate a uniform distribution of minimal programs by length. His algorithm uses a special reduction procedure which transform any program to its equivalent reduced (syntactically simplified) version of minimal length. This simplification procedure depends on a program domain and is not trivial in general. Results presented in [81] show that the semantic sampling procedure leads to statistical significant improvement of mean best fitness after the first 10 generations, when compared to the ramped half-and-half (RHH) initialization procedure. These advantages diminish in later generations and the differences become generally insignificant. These observations were consistent with the results presented in [84]. However, Looks showed that his heuristics eventually leads to better success ratio and lower computational effort for most problems.

Beadle and Johnson in [8] make an attempt to attain the same goal, i.e., to increase the behavioral diversity in the initial population. They present detailed analysis of program initialization, particularly the influence of behavioral diversity, tree size and shape on the performance of GP algorithm. Their semantically driven initialization (SDI) algorithm and

its hybridization with RHH called HSDI, in contrast to [81], operate on reduced ordered binary decision diagrams (ROBDD), which are translated into problem specific language in the process of creating the initial population. Results obtained by Beadle and Johnson are similar to those of Looks. They conclude that the performance improvement depends on the problem and report that, for the seven tested problems, HSDI performs best for three, SDI for two, and the traditional RHH is the best in two cases.

5.2.2. Semantically Unique Initialization

The algorithms proposed by Looks [81] or Beadle and Johnson in [8] are quite complicated and their implementation depends very much on the problem domain. Therefore, we propose and experimentally verify here a very simple method that rejects programs with semantics identical to any already created individual in the population. The drawbacks of this method mentioned in [81], like possibly quite large number of programs which needs to be generated, are compensated with trivial implementation.

As Koza noticed in [60], duplicated individuals are unproductive deadwood, and therefore it is desirable to avoid such duplicates. Therefore, some evolutionary packages, like ECJ, have population initialization procedure which, by default, tries to avoid syntactically identical individuals. In contrast, we propose an analogous initialization procedure analyzing semantics. Our procedure operates as follows. When a candidate individual generated during population initialization turns out to be semantically equivalent (see Definition 3 on page 34) to an individual already in the population, such an individual is discarded and a new candidate is randomly generated. The procedure tries to produce a unique individual several times (up to 100 by default — analogously to the syntactic procedure implemented in ECJ) before giving up (in this case the population will contain semantic duplicates).

In the control experiments we perform only the standard rejecting procedure of syntactic duplicates which is applied by default. This means that two individuals are considered equal if they are exactly the same, i.e., both have exactly the same nodes in the same places. In contrast, we also perform experiments with our candidate rejection when two individuals have the same, measured semantics, i.e., they behave equally, no matter how they are built. In both cases, a ramped half-and-half algorithm is used to generate new candidate individuals, and the rejecting procedure retries to generate a unique program maximal 100 times. Algorithm 5.1 presents both procedures to create initial population — with the standard syntactic duplicate rejection and with our rejection of semantically equivalent individuals.

We compare the GP performance between setups with syntactic and semantic duplicates rejection for several proportion between crossover and mutation probability. The tested probability of crossover varied from 0.0 to 1.0 with step 0.1, thus eleven setups with syntactic and eleven setups with semantic initialization are executed. Setups with both

Algorithm 5.1 Syntactic and semantic methods of population initialization.

```
1: maxTries ← maximal number of tries                          ▷ default 100
2: popSize ← population size

3: procedure Init-Syntactically
4:     P ← ∅                                                   ▷ P is a multiset
5:     for i ← 1 ... popSize do
6:         t ← maxTries
7:         repeat
8:             p ← Ramped-Half-and-Half(...)
9:             t ← t − 1
10:        until p ∉ P or t=0
11:        P ← P ∪ {p}
12:    return P
13: end procedure

14: procedure Init-Semantically
15:    P ← ∅, S ← ∅                                            ▷ P and S are multisets
16:    for i ← 1 ... popSize do
17:        t ← maxTries
18:        repeat
19:            p ← Ramped-Half-and-Half(...)
20:            s ← Semantics(p)                                ▷ get semantics of individual p
21:            t ← t − 1
22:        until s ∉ S or t = 0
23:        P ← P ∪ {p}
24:        S ← S ∪ {s}
25:    return P
26: end procedure
```

Setup	*Rank*	*Setup*	*Rank*
SEM|X 1.0	**6.29**	X+M 0.5	11.46
SEM|X+M 0.2	**6.91**	SEM|X+M 0.6	11.73
X 1.0	**6.96**	SEM|X+M 0.7	12.59
SEM|X+M 0.1	7.17	X+M 0.6	13.27
X+M 0.1	**7.55**	X+M 0.7	13.63
X+M 0.2	7.69	SEM|X+M 0.8	14.79
SEM|X+M 0.4	8.54	X+M 0.8	15.29
X+M 0.3	8.77	SEM|X+M 0.9	15.79
SEM|X+M 0.3	9.60	X+M 0.9	17.04
X+M 0.4	10.58	**M 1.0**	**17.60**
SEM|X+M 0.5	11.19	**SEM|M 1.0**	**18.54**

Table 5.1.: Friedman ranks of success ratio performance on all 39 problems (both symbolic regression and Boolean domain).

crossover and mutation operators are denoted as *X+M* β, where β is the probability of the second operator — here mutation. Prefix *SEM* denotes initialization with rejecting semantic duplicates, lack of it means the regular, syntactic rejection. We perform one thousand of independent runs of each setup. Other parameters of the evolutionary methods, are the same as used in the rest of experiments presented in this thesis (see Table 4.3 on page 43).

5.2.3. Results

Table 5.1 presents ranks from the Friedman test applied to success rates obtained by each setup. Bold font is used to emphasize the best setups from each combination of operators and the initialization procedure. The first observation is that setups without the mutation operator or with small proportion of it behave much better than setups with high mutation probability. This experiment shows that the semantic initialization gives slightly better results, however the difference is minimal.

The errors committed by the best-of-run individuals rank the methods in a very similar way to Table 5.1, so we do not report them here (see Appendix A). However, all observations and conclusions presented in this section are true also with reference to errors made by the best obtained individuals.

If we analyze the results in more detail, it turns out that for the symbolic regression problems the standard syntactic initialization leads generally to higher success ratio than the semantic initialization. Conversely, for the Boolean problems the situation is inverted. Tables 5.2 and 5.3 show results for each setup separately for both these domains. Numbers in bold font denote statistically significant superiority of a given initialization method.

As an example, let us compare the results obtained by the best setups from the ranking that do not use mutation (i.e., X 1.0 and SEM|X 1.0). For the symbolic regression problems, the semantic initialization gives better (statistically insignificant) results on

	P(xover)	*100*	*90*	*80*	*70*	*60*	*50*	*40*	*30*	*20*	*10*	*0*
F01	syntactic	96	95	92	**89**	82	73	**66**	56	48	39	30
	semantic	95	93	90	86	79	72	61	54	44	39	32
F02	syntactic	**71**	**65**	**57**	**49**	40	34	27	22	17	11	8
	semantic	66	58	52	44	39	30	24	19	16	9	8
F03	syntactic	**68**	**57**	**49**	**32**	**24**	14	9	**6**	3	2	1
	semantic	56	44	36	26	17	12	7	4	4	2	1
F04	syntactic	**29**	**21**	**16**	**12**	**8**	4	3	2	2	0	0
	semantic	22	16	12	8	5	5	2	2	1	1	0
F05	syntactic	2	1	1	1	2	1	1	1	1	1	1
	semantic	2	1	2	1	1	1	2	1	2	1	1
F06	syntactic	43	44	40	34	27	23	16	11	9	6	4
	semantic	46	42	37	34	27	21	15	11	9	7	4
F07	syntactic	5	4	2	2	3	2	1	1	2	1	1
	semantic	5	4	3	4	3	1	2	**2**	1	1	0
F08	syntactic	1	1	0	**1**	0	1	1	1	0	0	1
	semantic	0	1	1	0	1	0	1	0	1	0	0
F09	syntactic	25	25	23	20	20	16	14	10	10	8	6
	semantic	25	29	24	22	19	17	11	12	11	9	6
F10	syntactic	21	21	16	14	13	10	10	6	6	5	4
	semantic	22	20	18	**18**	14	**13**	9	6	6	5	4
F11	syntactic	21	22	**30**	30	30	33	33	35	31	28	28
	semantic	19	24	26	28	31	29	35	32	31	31	29
F12	syntactic	0	0	0	0	0	0	0	0	0	0	0
	semantic	0	0	0	0	0	0	0	0	0	0	0
P1	syntactic	0	0	0	0	0	0	0	0	0	0	0
	semantic	0	1	1	0	0	0	0	0	0	0	0
P2	syntactic	1	1	0	0	0	0	0	0	0	0	0
	semantic	0	0	0	0	0	0	0	0	0	0	0
P3	syntactic	**8**	3	2	1	0	0	0	0	0	0	0
	semantic	3	2	1	1	0	0	0	0	0	0	0
R0	syntactic	1	1	1	1	1	1	1	1	1	1	1
	semantic	1	1	1	0	1	1	1	1	1	2	0
R1	syntactic	0	0	0	0	0	0	0	0	0	0	0
	semantic	0	0	0	0	0	0	0	0	0	0	0
R2	syntactic	0	0	0	0	0	0	0	0	0	0	0
	semantic	0	0	0	0	0	0	0	0	0	0	0
R3	syntactic	0	0	0	0	0	0	0	0	0	0	0
	semantic	0	0	0	0	0	0	0	0	0	0	0

Table 5.2.: Comparison of success rates for 19 symbolic regression problems for different proportion of crossover and mutation operators ($P(mutation) = 100\% - P(crossover)$) — statistically better results (p-value < 0.05) in each pair are printed in bold. All values are in percentage (%).

	P(xover)	*100*	*90*	*80*	*70*	*60*	*50*	*40*	*30*	*20*	*10*	*0*
ICMP6	syntactic	15	19	18	21	19	18	17	15	13	8	1
	semantic	**20**	22	**22**	23	22	21	19	**20**	15	8	2
ICMP8	syntactic	0	0	0	0	0	0	0	0	0	0	0
	semantic	0	0	0	0	0	0	0	0	0	0	0
IMAJ5	syntactic	98	98	98	97	96	96	97	95	93	89	76
	semantic	98	98	99	98	98	96	96	95	93	**92**	**81**
IMAJ6	syntactic	71	68	68	65	61	56	51	44	37	22	1
	semantic	70	69	70	64	61	59	53	48	37	24	1
IMAJ7	syntactic	4	4	3	3	1	1	1	0	0	0	0
	semantic	3	4	3	2	2	1	1	0	0	0	0
IMUX11	syntactic	1	1	2	1	1	1	1	0	**1**	0	0
	semantic	1	2	1	**2**	**2**	1	1	1	0	0	0
IMUX6	syntactic	94	96	96	95	95	94	92	92	91	85	73
	semantic	96	96	95	94	94	95	94	92	91	87	75
IPAR4	syntactic	89	91	91	91	90	91	90	91	89	86	82
	semantic	**94**	**94**	92	92	92	92	91	91	91	**89**	**86**
IPAR5	syntactic	19	18	19	18	18	16	15	13	12	8	7
	semantic	21	19	21	19	19	16	17	16	13	11	7
IPAR6	syntactic	0	0	0	1	0	0	0	1	0	0	0
	semantic	1	0	0	0	1	0	0	1	0	1	0
NCMP6	syntactic	11	11	14	13	13	14	12	11	9	5	0
	semantic	13	14	16	15	16	17	**16**	**14**	11	5	0
NCMP8	syntactic	0	0	0	0	0	0	0	0	0	0	0
	semantic	0	0	0	0	0	0	0	0	0	0	0
NMAJ5	syntactic	88	87	83	85	81	81	79	77	73	59	33
	semantic	89	87	86	84	84	**84**	81	**81**	74	**67**	**41**
NMAJ6	syntactic	37	37	33	29	27	26	19	14	8	4	0
	semantic	37	33	33	31	30	26	**24**	**17**	9	4	0
NMAJ7	syntactic	1	0	0	0	0	0	0	0	0	0	0
	semantic	0	0	0	0	0	0	0	0	0	0	0
NMUX11	syntactic	0	0	0	0	0	0	0	0	0	0	0
	semantic	0	0	0	0	0	0	0	0	0	0	0
NMUX6	syntactic	75	76	78	79	78	76	76	76	71	63	30
	semantic	78	80	81	82	**82**	80	**80**	78	74	**67**	34
NPAR4	syntactic	41	41	41	37	36	35	31	26	23	13	5
	semantic	**48**	**47**	45	**44**	40	37	34	**31**	**27**	**17**	5
NPAR5	syntactic	0	0	0	0	0	0	0	0	0	0	0
	semantic	0	0	0	0	0	0	0	0	0	0	0
NPAR6	syntactic	0	0	0	0	0	0	0	0	0	0	0
	semantic	0	0	0	0	0	0	0	0	0	0	0

Table 5.3.: Comparison of success rates for 20 Boolean problems for different proportion of crossover and mutation operators ($P(mutation) = 100\% - P(crossover)$) — statistically better results (p-value < 0.05) in each pair are printed in bold. All values are in percentage (%).

two benchmarks, but in 8 cases it is worse (statistically significant in 4 cases). On the other hand, for the Boolean domain, it appears that semantic initialization is better for 9 problems (statistically significantly for 3) and (insignificantly) worse for 3 other problems.

These results clearly show that effectiveness of the semantic initialization, in the form proposed in this section, strongly depends not only on particular problem definition, but much more on the problem domain itself — especially on the used set of functions and terminals. We hypothesize, that the reason for this might be the fact that for symbolic regression problems the initial diversity in genetic material is not as important as in Boolean programs, because it is much easier achievable in the former domain.

Figure 5.2.2 shows the influence of initialization methods to the diversity in all generations. In this visualization, we use two alternative measures to quantify the diversification of population: the number of individuals with unique semantics and the number of individuals with unique fitness. Unsurprisingly, for regression problems, the number of semantically unique individuals is only slightly higher than the number of unique fitness values in the population. This is due to the fact that, for continuously valued semantics, it is very unlikely to have different semantics that have exactly the same fitness values. For discrete semantics used in the Booolean problems, this is much more frequent.

The graphs demonstrate that semantic duplicate rejection provides higher semantic diversity in the first generations, however later the difference becomes smaller. Note also that the difference in number of unique semantics for the later generations is higher for the two chosen symbolic regression problems than for the Boolean ones where the difference is negligible and hard to notice. We can then state that for the former domain the effects of semantically unique initialization are more likely to impact an entire evolutionary run than for the later.

Finally, Table 5.4 shows how many random programs need to be generated to ensure that the initial population (of size 500) will contain only unique individuals (syntactically or semantically, respectively). In other words, it is the total number of calls of the *Ramped Half-and-Half* function in appropriate procedure shown in Algorithm 5.1. For the reasons discussed earlier, the reported numbers are much lower for regression problems than for the Boolean problems, and more computational effort is required to provide semantic uniqueness than syntactic uniqueness. The varying number for semantic initialization for symbolic regression problems comes from a different range of input variables (for F07 and F08) or from different set of terminals (F09–F12 have two independent variables). For Boolean problems, the variation comes from different numbers of fitness cases (an implementation issue which influences on both syntactic and semantic procedure) and different function sets (other for problems with I and N prefixes).

Presented results demonstrate that increasing semantic diversity in the initial population is beneficial to achieved success ratio. Results not presented in this section (for more statistics see Appendix A) additionally show that setup SEM|X 1.0 produces programs

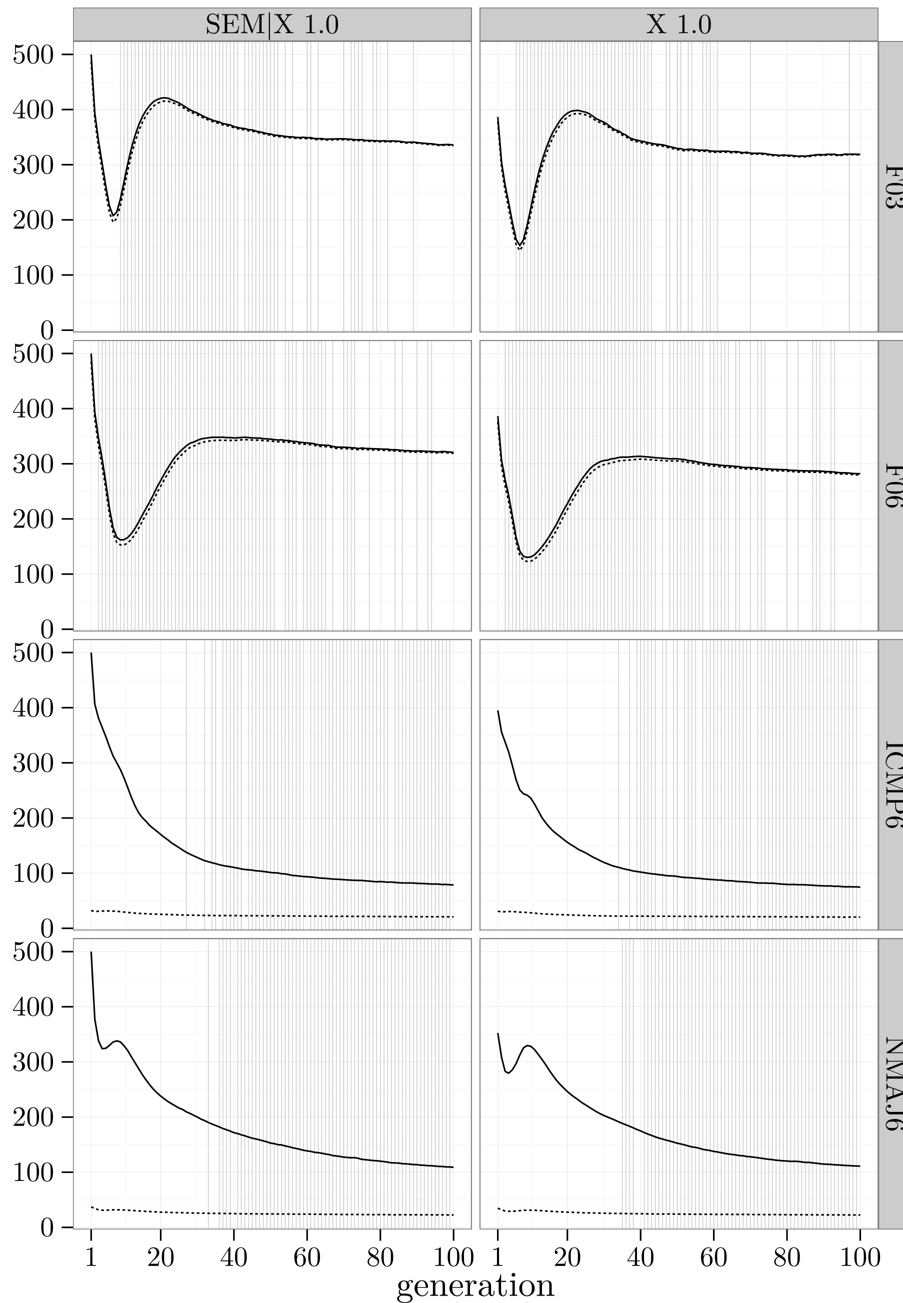


Figure 5.2.2.: Number of individuals with unique semantics (solid line) and unique fitness values (dotted line), for a crossover-only setup without (left column) and with (right column) semantically unique population initialization. Vertical lines show generations in which some runs finished with an ideal solution found, therefore values in subsequent generations are averaged from fewer runs.

Problem	*syntactic*	*semantic*	*Problem*	*syntactic*	*semantic*
F01	732	977	ICMP6	825	1081
F02	732	977	ICMP8	834	989
F03	732	977	IMAJ5	819	1197
F04	732	977	IMAJ6	825	1081
F05	732	977	IMAJ7	829	1024
F06	732	977	IMUX11	845	937
F07	732	972	IMUX6	825	1081
F08	732	972	IPAR4	818	1679
F09	732	832	IPAR5	819	1197
F10	732	832	IPAR6	825	1081
F11	732	832	NCMP6	793	1211
F12	732	832	NCMP8	796	1038
P1	732	977	NMAJ5	795	1442
P2	732	977	NMAJ6	793	1211
P3	732	977	NMAJ7	795	1105
R0	732	977	NMUX11	804	944
R1	732	977	NMUX6	793	1211
R2	732	977	NPAR4	797	2624
R3	732	977	NPAR5	795	1442
			NPAR6	793	1211

Table 5.4.: Mean number of randomly generated programs needed to ensure the syntactic and semantic uniqueness of all individuals in an initial population of the assumed size 500.

often making smaller errors than the best control setup (for 15 problems SEM|X 1.0 gives better error and for 5 problems worse error on the training set, and is better for 11 and worse for 5 problems on the testing set). Therefore, we can conclude that the semantic initialization is advantages in most cases, even though such initialization procedure requires a bit more time.

5.3. Crossover

5.3.1. Overview

As already signaled in Chapter 3, the major weakness of most crossover operators developed for GP is that they operate exclusively on syntactic level. For instance, for the standard tree-swapping crossover (cf. Section 2.2), neither the parent programs, nor the subtrees swapped between them, have anything in common semantically. As a result, such operators act in quite a haphazard way. No wonder then that with the dawn of semantically-aware approaches in GP, some attempts have been made to harness semantics for designing more effective crossover operators. This section is another contribution to that trend.

In the literature, several methods incorporating semantics into a crossover operator have been described.

Beadle and Johnson in [7] proposed the Semantically Driven Crossover. This modification to Koza standard crossover [60] adds the offspring to the new population only if it is semantically different (i.e., non-equivalent) to any of its parents. To check two programs

for semantic equivalence, Beadle and Johnson transform them to a canonical representation. As they apply their technique to Boolean problems, they used Reduces Ordered Binary Decision Diagrams (ROBDD) as the canonical form. In this case, two programs are semantically equivalent if and only if they reduce to the same ROBDD.

Quang Uy Nguyen *et al.* in [98] proposed a quite similar approach — Semantically Aware Crossover. However, it does not check the semantics of the whole resultant program, but instead ensures that the swapped subtrees under the crossover points have different semantics. Moreover, Nguyen *et al.* use sampling semantics to represent behavior of the subtrees, so the equivalence is only approximately checked.

The two studies cited above test the considered programs or subprograms only for equivalence. However, for sampling semantics it is easy to measure the distance between them (see Equation 3.4.1 on page 35). This property was exploited by Nguyen *et al.* in [127] in a Semantic Similarity Crossover (SSC) method. This operator tries to select as crossover points such subtrees in parent solutions that have similar but not identical semantics. Technically, this is realized by selecting at random candidate crossover points and checking whether the semantic distance between the corresponding subtrees falls within an assumed interval of upper and lower bound of semantic similarity. In this way, the operator increases the search locality (understood as leading to smaller semantics changes) and improves its overall performance.

This method has, however, one drawback — authors claim that SSC is very sensitive to its parameters, i.e., the lower and upper bound of an acceptable semantics distance. Therefore, in [126] the same team proposed several procedures for self-adapting these parameters. The best one, Self-Adaptation based on Successful Execution (SASE), adjusts the lower and upper bound depending on the proportion of SSC which successfully found the acceptable crossover points in the previous generation. If the matching points were found too easily then the range is narrowed, if too hard — the range is widened.

Krawiec and Lichocki proposed in [69] a crossover operator which is approximately geometric in the semantic space. The operator produces a fixed-size pool of candidate offspring using the standard crossover [60]. From this pool, only the child most similar to both parents is appointed as the 'true' offspring and is propagated to the new population. Formally, such offspring has minimal sum of distances between its semantics and the semantics of its first and second parents. Assuming that the fitness function in GP is usually also based on distance from a predefined target (see Formula 3.4.1), and thus the fitness landscape is a unimodal cone, such operator is expected to exploit the fitness–distance correlation in the semantic fitness landscape. Authors show that their approach is not worse than the conventional GP but, unfortunately, in the presented version does not outperform the reference method.

Algorithm 5.2 Semantic Similarity based Crossover (SSC) [127]

1: **procedure** SSC(p_1, p_2, A, B)
2: $maxTries \leftarrow$ maximal number of tries ▷ default 12
3: $t \leftarrow maxTries$
4: **repeat**
5: $subtree_1 \leftarrow$ random crossover point in p_1
6: $subtree_2 \leftarrow$ random crossover point in p_2
7: $t \leftarrow t - 1$
8: **until** $A < \|\text{SEMANTICS}(subtree_1) - \text{SEMANTICS}(subtree_2)\| < B$ **or** $t = 0$ ▷ see Equation 5.3.1
9: $c_1, c_2 \leftarrow$ children produced by swapping $subtree_1$ with $subtree_2$ in p_1 and p_2
10: **return** $\{c_1, c_2\}$
11: **end procedure**

5.3.2. Experiments

To show capabilities of semantic crossover we test the performance of Nguyen's SASES approach which, according to [126], is the best variant of SASE type of SSC method. In SASES, the required proportion of successful execution of SSC is not fixed like in SASE, but changes linearly during an evolution. Nguyen and coauthors claim that this method not only gives the best results but also reduces the number of tuning parameters.

Algorithm 5.2 shows a pseudocode of SSC method. The calculated distance (line 8 of the pseudcode) between two sampling semantics (lets call them **u** and **v**) is defined as

$$\|\mathbf{u} - \mathbf{v}\| = \frac{1}{N}\sum_{i=1}^{N} |u_i - v_i| \tag{5.3.1}$$

i.e., it equals to the mean absolute difference of each semantics elements. Note that this formula is equivalent (up to scaling) to the semantic similarity as defined in Formula 3.4.1. Let us remind that for Boolean domain, as mentioned in Section 3.4, we treat *true* and *false* values as 1.0 and 0.0, respectively. Therefore, the above equation in this case is equivalent to the Hamming distance normalized to an interval $[0;1]$. SSC tries to find a pair of crossover points that meet the required constraints (see line 8 of Algorithm 5.2) up to 12 times before giving up.

In the SASES approach the required proportion of successful executions of SSC in the beginning of evolution is set to 65%, and is continuously increased to 85% in the last (in our experiments — 100th) generation. If, in a given generation, the number of successful executions achieves this level, the $[A, B]$ range is narrowed, otherwise it is widened. In the first generation the upper (B) and lower (A) bounds are set to 0.4 and 0.004 respectively, and then they are both multiplied or divided by 0.9 to narrow or widen the bounds. These settings follow the ones used by Nguyen in [126].

To test the performance of SASES we run a series of experiments — 11 setups with

Setup	*Rank*	*Setup*	*Rank*	*Setup*	*Rank*
SASES+M 0.2	**5.51**	SASES+M 0.7	9.99	X+M 0.5	13.56
SASES+M 0.1	5.73	**X+M 0.2**	**10.27**	SASES+M 0.9	14.28
SASES 1.0	**5.87**	**X 1.0**	**10.47**	X+M 0.6	14.32
SASES+M 0.3	6.18	X+M 0.1	10.55	X+M 0.7	14.90
SASES+M 0.4	6.58	X+M 0.3	11.87	X+M 0.8	15.81
SASES+M 0.5	7.71	SASES+M 0.8	12.23	X+M 0.9	16.53
SASES+M 0.6	9.29	X+M 0.4	12.36	**M 1.0**	**16.99**

Table 5.5.: Friedman ranks of success ratio on all 39 problems (both symbolic regression and Boolean domain).

SASES and mutation operators, and 11 control setups with standard crossover and mutations. Setups are denoted as SASES+M β or X+M β, where β is the probability of choosing mutation and it varies from 0 to 1 with step 0.1 (therefore we have 21 distinct setups in total). All other parameters are exactly the same as in previous experiments. Two hundred independent runs of each of the above setups were performed.

5.3.3. Results

Table 5.5 presents ranks from the Friedman test applied to success rates obtained by each setup. Again, bold font is used to emphasize the best setups from each combination of operators. This ranking clearly shows that the setups with SASES outperform canonical crossover, which confirms conclusions formulated in [126]. The best setup, SASES+M 0.2, statistically significantly (p-value < 0.05) outperforms SASES+M 0.6 and all further setups. The best control experiment (X+M 0.2) is statistically worse than the first four setups (SASES+M 0.2 – SASES+M 0.3).

Tables 5.6 and 5.7 show detailed values of success rates for each setup and problem. Success rates that are statistically better values than the performance of GP with standard crossover are marked with bold font. The main observation resulting from these tables is that the setups with the standard crossover never perform significantly better than SASES. The latter method, for at least one proportion of mutation, is statistically better on eleven symbolic regression problems and thirteen Boolean problems.

5.4. Mutation

5.4.1. Overview

In [9], Beadle and Johnson proposed Semantic Driven Mutation operator. This algorithm proceeds as the standard subtree mutation, but after producing the candidate offspring checks if it is semantically different from its parent. If both programs are semantically equivalent, the mutation is reverted and the process is repeated. If the algorithm fails several times to produce a semantically distinct child, then the parent is copied to a new

	β	*0.0*	*0.1*	*0.2*	*0.3*	*0.4*	*0.5*	*0.6*	*0.7*	*0.8*	*0.9*	*1.0*
F01	SASES+M	0.96	0.94	0.94	0.93	**0.93**	0.81	**0.80**	0.61	0.57	0.44	0.29
	X+M	0.96	0.96	0.92	0.90	0.83	0.75	0.63	0.59	0.50	0.39	0.29
F02	SASES+M	0.74	**0.74**	**0.73**	**0.67**	**0.52**	**0.49**	**0.45**	0.31	0.21	0.19	0.07
	X+M	0.73	0.64	0.60	0.45	0.38	0.29	0.25	0.24	0.15	0.14	0.07
F03	SASES+M	0.68	**0.66**	0.53	**0.45**	0.29	0.20	0.09	0.07	0.04	0.04	0.00
	X+M	0.68	0.55	0.54	0.30	0.25	0.19	0.11	0.07	0.03	0.04	0.00
F04	SASES+M	0.36	**0.31**	**0.28**	**0.22**	0.14	**0.11**	0.07	0.02	0.04	0.00	0.00
	X+M	0.27	0.18	0.14	0.13	0.08	0.04	0.04	0.02	0.03	0.00	0.00
F05	SASES+M	0.03	0.05	0.02	**0.06**	0.03	0.04	0.05	**0.05**	0.01	0.02	0.01
	X+M	0.02	0.03	0.00	0.01	0.01	0.02	0.02	0.01	0.01	0.01	0.01
F06	SASES+M	0.56	**0.58**	0.53	0.49	**0.54**	**0.41**	**0.25**	**0.23**	0.13	0.08	0.06
	X+M	0.46	0.46	0.43	0.41	0.24	0.20	0.16	0.11	0.10	0.07	0.06
F07	SASES+M	0.04	0.04	0.02	0.05	0.03	0.02	0.04	0.01	0.01	0.01	0.02
	X+M	0.07	0.05	0.02	0.02	0.04	0.01	0.02	0.00	0.03	0.01	0.02
F08	SASES+M	**0.03**	0.02	0.01	0.01	0.02	0.02	0.00	0.01	0.02	0.02	0.01
	X+M	0.00	0.00	0.01	0.01	0.00	0.01	0.01	0.00	0.01	0.00	0.01
F09	SASES+M	**0.85**	**0.84**	**0.84**	**0.80**	**0.70**	**0.51**	**0.26**	**0.30**	**0.20**	0.11	0.07
	X+M	0.25	0.29	0.23	0.17	0.20	0.15	0.12	0.11	0.10	0.06	0.07
F10	SASES+M	**0.75**	**0.71**	**0.63**	**0.62**	**0.51**	**0.31**	**0.26**	**0.17**	**0.16**	0.09	0.04
	X+M	0.19	0.19	0.13	0.14	0.16	0.10	0.08	0.07	0.05	0.07	0.04
F11	SASES+M	**0.35**	**0.44**	**0.43**	**0.42**	**0.46**	**0.52**	**0.53**	**0.47**	0.41	**0.40**	0.29
	X+M	0.17	0.19	0.29	0.26	0.29	0.37	0.35	0.32	0.32	0.26	0.29
F12	SASES+M	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	X+M	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
P1	SASES+M	0.00	0.01	0.01	0.01	0.01	0.00	0.00	0.00	0.01	0.00	0.00
	X+M	0.00	0.00	0.01	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00
P2	SASES+M	0.02	0.00	0.01	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	X+M	0.00	0.01	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
P3	SASES+M	0.11	0.07	0.04	0.01	0.00	0.01	0.00	0.01	0.00	0.00	0.00
	X+M	0.07	0.03	0.01	0.01	0.01	0.00	0.00	0.00	0.00	0.00	0.00
R0	SASES+M	0.03	0.02	0.01	0.01	**0.04**	0.02	0.01	0.04	0.01	0.01	0.02
	X+M	0.01	0.01	0.02	0.01	0.01	0.02	0.01	0.01	0.00	0.03	0.02
R1	SASES+M	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00
	X+M	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R2	SASES+M	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	X+M	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R3	SASES+M	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	X+M	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 5.6.: Comparison of success rates for 19 symbolic regression problems for SASES+M β and X+M β for different values of β (i.e., probability of mutation). Statistically better results (p-value < 0.05) in each pair are printed in bold.

	β	*0.0*	*0.1*	*0.2*	*0.3*	*0.4*	*0.5*	*0.6*	*0.7*	*0.8*	*0.9*	*1.0*
ICMP6	SASES+M	**0.29**	**0.33**	**0.45**	**0.42**	**0.42**	**0.35**	**0.35**	**0.35**	**0.31**	**0.15**	0.01
	X+M	0.19	0.22	0.23	0.21	0.20	0.18	0.15	0.13	0.16	0.06	0.01
ICMP8	SASES+M	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	X+M	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
IMAJ5	SASES+M	1.00	1.00	**1.00**	0.98	**1.00**	**1.00**	0.98	0.96	0.97	**0.95**	0.78
	X+M	0.99	0.98	0.96	0.97	0.97	0.95	0.97	0.97	0.93	0.88	0.78
IMAJ6	SASES+M	**0.88**	**0.91**	**0.88**	**0.89**	**0.84**	**0.80**	**0.80**	**0.71**	**0.62**	**0.45**	0.01
	X+M	0.74	0.65	0.75	0.67	0.64	0.57	0.47	0.45	0.35	0.24	0.01
IMAJ7	SASES+M	**0.29**	**0.26**	**0.18**	**0.15**	**0.07**	**0.09**	0.02	0.01	0.01	0.00	0.00
	X+M	0.05	0.05	0.03	0.02	0.02	0.01	0.01	0.00	0.00	0.00	0.00
IMUX11	SASES+M	**0.06**	**0.07**	**0.07**	**0.09**	**0.11**	**0.08**	**0.04**	0.04	0.01	0.01	0.00
	X+M	0.01	0.02	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.00	0.00
IMUX6	SASES+M	0.99	0.99	**1.00**	0.99	**1.00**	**0.99**	**0.98**	**1.00**	0.96	**0.95**	0.73
	X+M	0.96	0.96	0.96	0.95	0.94	0.94	0.89	0.93	0.92	0.84	0.73
IPAR4	SASES+M	0.91	0.91	0.96	0.93	0.94	0.92	0.93	0.92	0.90	0.89	0.84
	X+M	0.89	0.90	0.93	0.91	0.94	0.91	0.90	0.92	0.89	0.91	0.84
IPAR5	SASES+M	0.23	**0.29**	0.29	**0.27**	0.25	**0.28**	0.18	0.19	0.20	0.14	0.09
	X+M	0.17	0.16	0.21	0.18	0.17	0.16	0.16	0.16	0.15	0.09	0.09
IPAR6	SASES+M	0.01	0.01	0.01	0.02	0.01	0.01	0.01	0.02	0.01	0.02	0.01
	X+M	0.01	0.00	0.01	0.00	0.01	0.00	0.00	0.02	0.01	0.01	0.01
NCMP6	SASES+M	**0.34**	**0.38**	**0.45**	**0.37**	**0.47**	**0.39**	**0.39**	**0.30**	**0.29**	**0.18**	0.00
	X+M	0.10	0.15	0.11	0.12	0.13	0.15	0.10	0.13	0.09	0.06	0.00
NCMP8	SASES+M	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	X+M	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
NMAJ5	SASES+M	**0.98**	**1.00**	**0.99**	**0.98**	**0.96**	**0.96**	**0.93**	**0.90**	**0.85**	**0.76**	0.34
	X+M	0.87	0.85	0.80	0.87	0.81	0.81	0.80	0.77	0.73	0.59	0.34
NMAJ6	SASES+M	**0.85**	**0.86**	**0.79**	**0.76**	**0.78**	**0.70**	**0.69**	**0.53**	**0.42**	**0.18**	0.00
	X+M	0.36	0.36	0.32	0.31	0.25	0.21	0.18	0.16	0.09	0.03	0.00
NMAJ7	SASES+M	**0.15**	**0.18**	**0.10**	**0.09**	**0.05**	**0.04**	0.01	0.01	0.00	0.00	0.00
	X+M	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00
NMUX11	SASES+M	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	X+M	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
NMUX6	SASES+M	**0.94**	**0.90**	**0.94**	**0.97**	**0.96**	**0.95**	**0.96**	**0.92**	**0.90**	**0.84**	0.30
	X+M	0.73	0.80	0.74	0.80	0.79	0.80	0.86	0.80	0.71	0.68	0.30
NPAR4	SASES+M	**0.57**	**0.60**	**0.53**	**0.56**	**0.52**	**0.51**	**0.42**	**0.43**	**0.40**	0.20	0.06
	X+M	0.37	0.43	0.40	0.39	0.37	0.33	0.30	0.30	0.21	0.18	0.06
NPAR5	SASES+M	0.01	0.01	0.01	0.01	0.01	0.01	0.00	0.00	0.00	0.00	0.00
	X+M	0.01	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
NPAR6	SASES+M	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	X+M	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 5.7.: Comparison of success rates for 20 Boolean problems for SASES+M β and X+M β for different values of β (i.e., probability of mutation). Statistically better results (p-value < 0.05) in each pair are printed in bold.

Algorithm 5.3 Semantic Similarity based Mutation (SSM) [99]

1: **procedure** SSM(p, A, B)
2: $maxTries \leftarrow$ maximal number of tries ▷ default 12
3: $t \leftarrow maxTries$
4: **repeat**
5: $subtree_1 \leftarrow$ random mutation point in p
6: $subtree_2 \leftarrow$ GROW(1, $maxDepth$) ▷ see Algorithm 2.1
7: $t \leftarrow t - 1$
8: **until** A $<$ $\|$SEMANTICS($subtree_1$) $-$ SEMANTICS($subtree_2$)$\|$ $<$ B **or** $t = 0$ ▷ see Equation 5.3.1
9: **return** child produced by replacing $subtree_1$ with $subtree_2$ in p
10: **end procedure**

population. In this work, Beadle and Johnson also used Reduced Ordered Binary Decision Diagrams (ROBDD) to test the candidate solutions for behavioral equivalence — exactly as in Semantically Driven Crossover (see Section 5.3).

In line of the Semantically Aware Crossover and Semantic Similarity Crossover (described in Section 5.3), Nguyen *et al.* in [99] presented two analogous methods: Semantically Aware Mutation (SAM) and Semantic Similarity Mutation (SSM). The first operator chooses randomly a mutation point (a subtree in the parent program) and generates at random a new subtree. If the distance between semantics of this two subtrees (calculated according to Equation 5.3.1) is sufficiently large, i.e., exceeds certain threshold (the only parameter of the method), then the subtree at the mutation point is replaced by the new subtree. Otherwise, the procedure may be repeated (but originally, in [99], it is not) several times, and in case of failure a standard subtree mutation can eventually be applied. In this way, SAM tries to force the new subtree to behave differently enough than the old replaced subtree.

The SSM method works very similarly to SAM. However, instead of testing semantic equivalence, SSM checks if the semantic similarity are in a given range (see Algorithm 5.3). Therefore, the SSM operator has two parameters — lower and upper bound of acceptable distance between semantics. Both SAM and SSM methods use sampling semantics as all previously mentioned Nguyen *et al.*'s algorithms.

5.4.2. Experiments

In this section we conduct experiments to test SSM on our benchmark problems. The lower and upper bound of semantic sensitivity are set to B = 0.4 and A = 0.004, respectively (like for SASES in the previous section). SSM makes up to 12 attempts to generate an appropriate subtree before applying the standard subtree mutation.

Again, we tested different proportion between pairs of operators — standard crossover and SSM (X+SSM β) for the SSM setups and standard crossover with standard mutation

Setup	*Rank*	*Setup*	*Rank*	*Setup*	*Rank*
X+SSM 0.1	**7.01**	X+SSM 0.4	9.09	X+SSM 0.7	12.97
X+M 0.2	**7.10**	X+M 0.4	9.50	X+M 0.8	13.77
X+M 0.1	7.15	X+SSM 0.5	10.14	X+SSM 0.8	13.96
X 1.0	**7.40**	X+M 0.5	10.46	X+SSM 0.9	14.62
X+SSM 0.2	7.82	X+M 0.6	11.51	X+M 0.9	15.15
X+SSM 0.3	7.88	X+SSM 0.6	12.00	**M 1.0**	**15.94**
X+M 0.3	8.63	X+M 0.7	12.55	**SSM 1.0**	**16.33**

Table 5.8.: Friedman ranks of success ratio performance on all 39 problems (both symbolic regression and Boolean domain).

(X+M β) as control experiments. Other parameters are set as in previous experiments in this thesis.

5.4.3. Results

The ranking of all examined setups is presented in Table 5.8. Bold font is used to emphasize the best setups from each combination of operators. It appears that the setup X+SSM 0.1 is slightly better than the second setup X+M 0.2, but this difference is negligible. In fact the best setup, X+SSM 0.1, statistically significantly outperforms only the twelfth (X+M 0.6) and the following items from this ranking. The best control experiment (X+M 0.2) significantly outperforms exactly the same setups.

One explanation for these results could be that SSM is overall not a very good mutation operator. However, its not necessary true because the best setups in Table 5.8 use mutation very rarely or not at all. Therefore, even if SSM was very advantageous comparing to the standard mutation, this operator would have little chance to demonstrate that. On the other hand, in the setups with large mutation probability, SSM could act too intensely, causing exaggerated exploration while neglecting exploitation of the search space.

When analyzing individual success rates for particular problems, it appears that there are only a few statistical significant differences between SSM and standard mutation. For all problems, SSM demonstrates its advantage only in two cases from all considered proportions of crossover operator, and the standard mutation outperforms SSM only in three cases out of 390 cases (39 problems times 10 setups with active mutation operators). For this reason, the detailed tables presenting those results are omitted here (but they are attached in Appendix A).

5.5. Summary

In this chapter we described selected past works on the semantic methods applied to GP. We also conducted a series of extensive computational experiments, in which we tested the most promising of these methods on a benchmark suite. The results obtained in this way

Setup	*Rank*	*Setup*	*Rank*
SEM|SASES+SSM 0.1	**13.19**	SEM|SASES+SSM 0.8	32.06
SEM|SASES 1.0	**13.96**	SASES+SSM 0.8	32.17
SASES+M 0.2	**14.23**	SEM|X+M 0.4	32.28
SEM|SASES+SSM 0.3	15.15	X+SSM 0.3	32.37
SASES+M 0.1	15.35	SEM|X+M 0.3	33.17
SASES+SSM 0.1	**15.42**	X+M 0.3	34.08
SASES 1.0	**15.62**	X+SSM 0.4	34.85
SASES+M 0.3	16.73	SASES+M 0.8	35.06
SEM|SASES+SSM 0.2	17.45	X+M 0.4	36.01
SASES+SSM 0.3	17.69	SEM|X+M 0.5	37.54
SASES+M 0.4	18.01	SEM|X+M 0.6	37.91
SASES+SSM 0.2	19.13	X+SSM 0.5	38.33
SASES+SSM 0.4	19.91	SASES+SSM 0.9	38.71
SEM|SASES+SSM 0.4	20.01	X+M 0.5	39.15
SASES+SSM 0.5	21.06	SEM|X+M 0.7	40.40
SASES+M 0.5	21.40	SASES+M 0.9	41.14
SEM|SASES+SSM 0.6	21.74	X+M 0.6	41.14
SEM|SASES+SSM 0.5	23.63	SEM|SASES+SSM 0.9	41.81
SASES+SSM 0.6	25.65	X+SSM 0.6	41.87
SEM|X 1.0	**26.26**	X+M 0.7	43.51
SEM|SASES+SSM 0.7	26.56	X+SSM 0.7	43.83
SASES+M 0.6	26.85	SEM|X+M 0.8	44.42
SEM|X+M 0.1	**28.04**	X+M 0.8	45.27
SEM|X+M 0.2	28.38	X+SSM 0.8	45.94
SASES+SSM 0.7	28.63	X+SSM 0.9	46.55
SASES+M 0.7	28.69	SEM|X+M 0.9	46.82
X+M 0.2	**29.05**	X+M 0.9	47.87
X+SSM 0.1	**29.42**	**M 1.0**	**49.13**
X 1.0	**29.76**	**SEM|SSM 1.0**	**49.81**
X+M 0.1	30.05	**SSM 1.0**	**50.41**
X+SSM 0.2	31.56	**SEM|M 1.0**	**50.78**

Table 5.9.: Friedman ranks of success ratio performance on all 39 problems (both symbolic regression and Boolean domain) for all semantic operators. There are 62 setups in total.

form a base for comparison with more advanced methods shown in following chapters.

The experiments from this chapter demonstrate that semantically oriented search operators proposed recently for GP, when compared to the standard crossover and mutation, are generally profitable, however to a greater or lesser extent, depending on the setup and problem. Especially, the SASES appears to evidently outperform the standard crossover, whereas the advantage of semantic population initialization procedure proposed in Section 5.2 or SSM tested in Section 5.4 is hardly visible.

Finally, in this section we compare all setups from previous sections together. In this comparison, we include also two additional setups that combine SASES and SSM operators (setups SASES+SSM β), and setups that contain all three semantic operators, i.e., semantic initialization, SASES, and SSM, at the same time (setups SEM|SASES+SSM β).

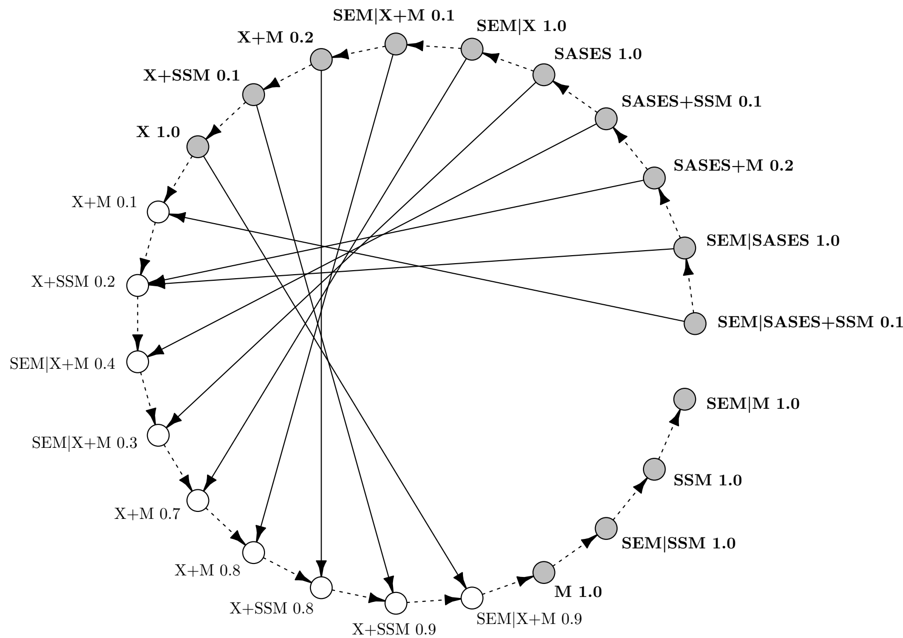


Figure 5.5.1.: Subset of the ranking presented in Table 5.9. Dotted arrows show order of ranks, solid arrows shows the first setup in the ranking which is statistically significantly worse (Friedman test with Shaffer's post-hoc procedure, p-value < 0.05).

The parameters of both operators and the whole evolution are the same as in previous sections.

The obtained ranking is presented in Table 5.9 (as practiced in this thesis, we mark the best setup of a given type in bold). This ranking shows that synergy of all semantic operators is profitable — the best nineteen setups (out of 62) incorporate the SASES operator, but the best setup is a fusion of all three semantic operators. This suggests that it may be worthy to design GP variants that implement semantic-aware mechanisms on *all* stages of evolutionary search: population initialization, recombination, and mutation. It is also worth emphasizing that the gap between the performance of most semantic-aware methods, particularly SEM|SASES+SSM β, and the best non-semantic setups is big: the best setup based on standard GP, X+M 0.2, has rank around 29, while the leader around 13.

To conclude, Figure 5.5.1 visualize graphically which setups are statistically better than other (p-value < 0.05) in the sense of achieving success ratios (*cf.* Table 5.9). The results are obtained by Friedman test with Shaffer's post-hoc procedure which compares all 62 possible pairs of tested setups, but the graph shows the 14 best setups from each pair of used operators (denoted by bold font in Table 6.1) and 9 setups directly surpassed by them. The dotted arrows show order of setups in the ranking. Solid arrows point the first, statistically significantly worse setup from the ranking. This means that if some method

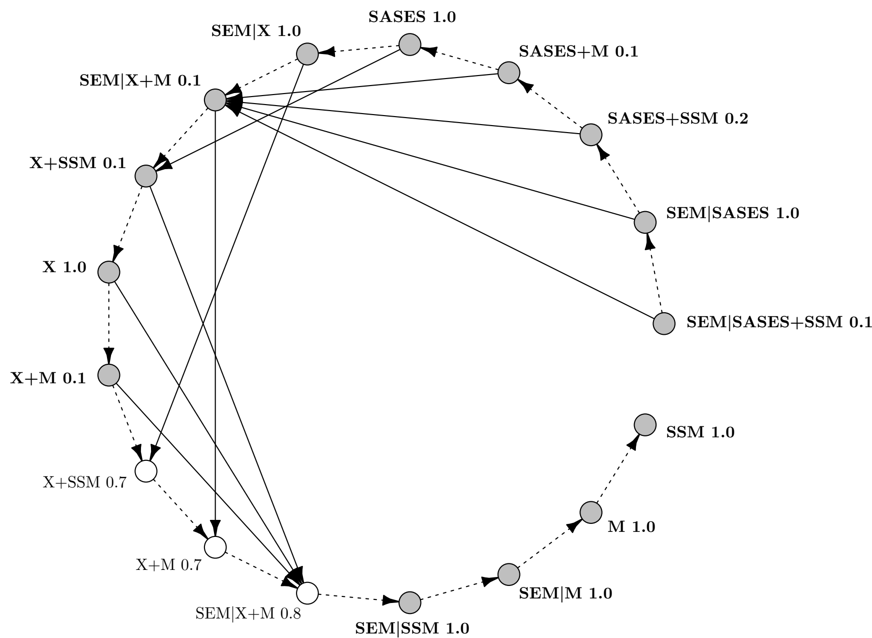


Figure 5.5.2.: Subset of Friedman ranking of *median error* on training set performed on all 39 problems.. Dotted arrows show order of ranks, solid arrows shows the first setup in the ranking which is statistically significantly worse (Friedman test with Shaffer's post-hoc procedure, p-value < 0.05).

A statistically outperforms some other method B, then it simultaneously outperforms all methods following B in the ranking (dotted arrows).

It appears, that the performed test (Shaffer's post-hoc procedure that compares all setups together) does not show that the best control setup (X+M 0.2) is statistically worse than any of the semantic operators. However, the best setup is statistically better than X+M 0.1 which has only one worse rank than the best control setup (X+M 0.1 — 30.05, X+M_0.2 — 29.05, see Table 5.9). On the other hand, the Holm's post-hoc procedure (comparing selected method with all others) shows that the best setup (SEM|SASES+SSM 0.1) is statistically better than SASES+SSM 0.6 and following setups. Moreover, using this procedure, the best control setup (X+M 0.1) is statistically worse than the first 7 best setups from the ranking.

The advantage of semantics setups is much more visible when comparing errors (see Section 4.5.1). Figure 5.5.2 shows a similar graph visualizing which setups are statistically better (Shaffer's post-hoc procedure, p-value < 0.05) in the sense of committed errors on training set. This chart shows, that five best setups (SEM|SASES+SSM 0.1, SEM|SASES 1.0, SASES+SSM 0.2, SASES+M 0.1, and SASES 1.0) are statistically better than the best control setup (X 1.0). However, the Holm's post-hoc procedure says that the first 17 best setups are statistically better than X 1.0 (see Appendix A for full ranking of median errors).

6. Desired Semantics

6.1. Introduction

In this chapter we introduce the concept of *desired semantics,* and propose and experimentally investigate a few approaches that rely on it. Informally, desired semantic, calculated for a location in a program, expresses the semantics of a code fragment (*desired building block*) which, when implanted at that location, maximizes program's fitness. To calculate the desired semantics, a property of (at least partial) instruction (function) inversibility is required. In other words, the proposed approach requires all instructions (functions) given by a problem definition to be reversible (at least to some extent, which we will explain further in this chapter).

To explain how the proposed idea of desired building blocks works, we first introduce the term 'context' (see also Chapter 7). The *context* is a fragment of a GP program bereft of some *part* of it. Because, we constrain this thesis only to the tree-based GP (see Section 2.2), we will further identify the *context* with an incomplete tree that misses a single branch (*part*). Thus, an entire program can be assembled by combining a context with a part (subtree).

Definition 5. *The desired semantics of a context* (*desired semantics* for short) is semantics of such a part, which combined with the given context causes the resultant program to have the target semantics (i.e., a program which is an ideal solution). A desired semantics of an empty context is equivalent to the target semantics (see Section 3.4) as in this case the missing part has to be the entire solution itself.

Formally, the desired semantics of a context c for problem with target semantics t may be expressed as a set $D(c,t)$ (D in short):

$$D(c,t) = \{x : s(c \circ x) = t\}$$

where $s(c \circ x)$ denotes semantics of context c fed with (composed with) semantics x, i.e., semantics of a composition of a context c and a hypothetical part with semantics x (i.e., not necessarily constructable from an available set of instructions). In other words, desired semantics of a context describes semantics of all such parts that the semantics of a program assembled from one of these parts and the given context equals the target semantics.

It is important to notice that for particular context c and target t, the desired semantic

may not exist (and thus $D = \emptyset$). In such a case, no value fed into the context at the missing branch location (denoted by '#') can make c return t. Note also however, that this situation is different from a scenario in which $D \neq \emptyset$, but there is no part that has the desired semantics, which we will discuss later in this chapter.

The *desired* semantics of a context should not be confused with a semantics of a (sub)program (represented by a (sub)tree). Semantics is associated with a root node of a (sub)tree and describes the behavior of this (sub)program. In contrast, the desired semantics describes the desired behavior of a hypothetical part which does not have to exists. Therefore, it pertains to a missing part of a context.

In following, by 'desired semantics of a node' we mean the desired semantics of a context created by removing a subtree rooted at that node from the original program. Desired semantics always refers to some context.

The desired semantics says what, in an ideal case, should be put in a given place to get an optimal solution of a problem (i.e., cause the semantics of the program to equal the target). Obviously, such desired subtree semantics might not exist at all or be ambiguous. For this reason the desired semantics requires richer means of expression than subtree semantics, which enable describing the following five possible situations:

1. There exists exactly one semantics of a subtree which causes the context to have the target semantics.

2. There exists more than one such semantics, but the set of all of them is finite.

3. There is an infinite number of such semantics, so the only way to express them concisely is to resort to a more complex notation, e.g. to formulas.

4. Any semantics is accepted in D because whatever is fed into the context, the resultant tree will always have the target semantics. In other words, the missing tree in this context is an intron and does not have any influence on the final behavior of the program.

5. No matter what is fed into the context, the resultant tree will not have the target semantics anyway. In such a case, the context is not able to achieve the target semantics in any situation (to do this, the context has to be changed itself).

To illustrate all of the above situations let us assume that the problem is to evolve a real-valued mathematical expression which always (identically) equals zero, i.e., its target semantics $t = 0$. For this problem we can easily design contexts that represent the above categories. Examples of such contexts are shown in Figure 6.1.1. The first context encoding expression $1 \cdot (\# - 1)$, where # is the missing subtree, accepts exactly one semantics — only by replacing # with a subtree returning 1 one can obtain an expression which equals 0. The second expression $1 - \#^2$ accepts two possible semantics: -1 and 1. In the next

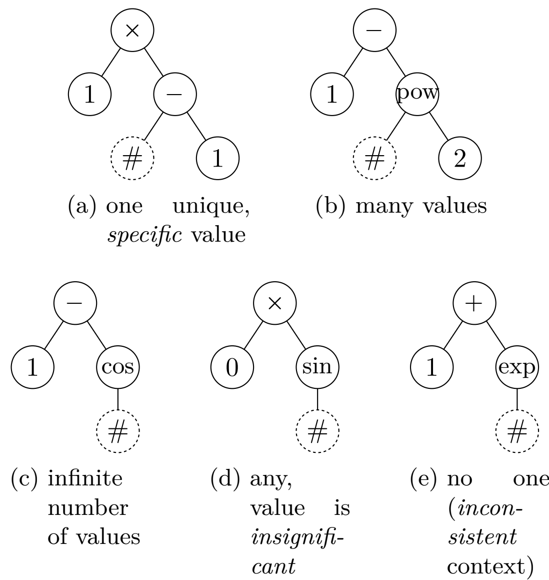


Figure 6.1.1.: Examples of five qualitatively different situations concerning desired semantics for zero-valued target (i.e., $t = 0$). The subtitles report the number of semantics accepted in D. The dotted circle (node '#') represents a missing subtree of the context.

example, $1 - cos(\#)$, a subtree returning any value equal to $2\pi n$, $n \in \mathbb{Z}$ is acceptable, so the cardinality of D is infinite (but countable). Next, when the context is $0 \cdot \sin(\#)$, then whatever is put in the missing place, the whole expression will always equal zero anyway (D has an infinite and uncountable number of elements). Finally, the last example shows a situation in which the resultant expression is always greater than 1 disregard the semantic substituted in place of #, so this context cannot be used to construct a solution to our problem (such desired semantics will be called *inconsistent* in following).

In this simple and extreme example, the five above-mentioned situations occurred globally for the whole semantics. However, they can occur also locally, independently for each component of semantics. For instance, when the used semantics has a form of *sampling semantics* (see Definition 2 on page 33), i.e., it is a list of responses for each fitness case, then each element of this list falls into one of the above five categories individually and independently from others. This feature is very important because it allows practical application of methods using desired semantics, even if some elements of it are insignificant or inconsistent.

Further, in the experimental part, we simplify the space of desired semantics, so each element can express only one from three cases (instead of five):

- One, concrete value is acceptable (see Case 1 on the facing page).
- Any value is acceptable (i.e., 'don't care') — such elements will be called *insignificant*

in following (Case 4).

- No value is acceptable — it is *inconsistent* (Case 5).

Such *simplified desired semantics* can be expressed as a single list (similarly to sampling semantics defined in Section 3.4) with two special values encoding undefined values — both insignificant ('don't care') and inconsistent elements. In following we will identify desired semantics just with this simplified version of it, which does not require set notation any more.

In our implementation, when more values are acceptable (see Cases 2 and 3 on page 76), only one of them, arbitrary chosen, will be stored and considered in further processing (instead of a full set of all possible values, or an expression defining all of them). We choose such proceeding to simplify both the computations and the representation complexity, even though this can introduce some bias and punish good subtrees. Otherwise, the complexity of desired semantics would increase exponentially with the length of the path from the root node to the missing subtree of context.

To illustrate this, let us continue the example presented on page 76 where the goal was to evolve an expression returning zero. For the context $1 - \#^2$ (see Figure 6.1.1b) our simplified desired semantics will equal either -1 or 1, and any further computation will assume that only one of them is correct. Similarly, for the context $1 - cos(\#)$ (see Figure 6.1.1c), the simplified desired semantics will be 0 (although in theory it could be any multiplicity of 2π, our implementation prefers smaller values among all valid). As a result, a part returning, e.g., 2π, will be treated as committing some error, even though this value is formally acceptable too.

To calculate the desired semantics of a context, all instructions should be invertible, as mentioned at the beginning of this section. When we have perfectly invertible instruction, the inversion of it will have the same properties as an inverse function in mathematics. This means, that it must be possible to calculate the desired input for each of used functions (instructions), given the values all the other inputs (remaining arguments of a function) and the expected output (the result of the function). For functions that are not fully invertible, some elements of the calculated desired semantics can be ambiguous or inconsistent. It should be also noticed that the invertibility requirement means that the used functions cannot be just black boxes given by a problem definition — we must know how to calculate the desired argument to get an expected function value (i.e., we must know the inverse functions).

When a method calculating the desired input value for each function is given, the algorithm to calculate the desired semantics of a whole context is straightforward. It starts from the root node and goes along the path to the missing subtree of the context. In each step, the desired value of an argument on the path is calculated. Thus, at the beginning, the desired argument for the instruction located at the root node is calculated

Algorithm 6.1 Calculating *desired semantics of a context*

```
1: procedure DesiredSemantics(c, t)          ▷ for context c and target semantics t
2:     L ← list of nodes on the path from root of c to the missing subtree
3:     d ← t                                  ▷ desired semantics of an empty context
4:     for all n ∈ L do
5:         S ← semantics of all children x (subtrees rooted in x) of node n such that x ∉ L
6:         d ← n^{-1}(d, S)       ▷ calculate desired values using inverse of function n
7:     return d
8: end procedure
```

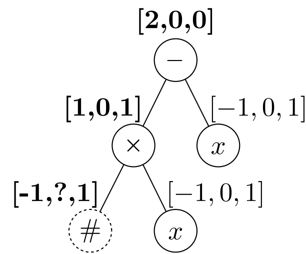


Figure 6.1.2.: Exemplary context with calculated desired semantics (in bold). The target semantics of a task is $[2, 0, 0]$, semantics of an independent variable is $[-1, 0, 1]$, and the context desired semantics is $[-1, ?, 1]$.

(the given target semantics is simultaneously a desired semantics of the root). Then, recursively, all consecutive nodes on the path are processed. For each node, the semantics of all its subtrees (arguments) not belonging to the path are directly accessible, therefore the only unknown for each node is the quested desired semantics. The last calculated value forms the unknown desired semantics of the context. Algorithm 6.1 shows the pseudocode of this procedure.

To give an example, let us suppose that a given problem is a symbolic regression task to evolve expression $x^2 - x$. The only input variable is x, and there are three fitness cases, for which it assumes values $-1, 0, 1$, respectively. Thus, the semantics of the terminal node x equals $[-1, 0, 1]$ and the sampling semantics of the target is $[2, 0, 0]$. Figure 6.1.2 shows an exemplary context $(\# \cdot x - x)$ with semantics of all subtrees denoted in plain text (here only terminals). In bold there are shown all desired semantics computed from the root node till the missing part of the context (i.e., also the context which arises by removing the subtree rooted in the multiplying ($\times$) node). The $[2, 0, 0]$ semantics presented in the figure is both the desired semantics of a completely empty context and the target semantics of problem.

In this example the desired semantics of the context is $[-1, ?, 1]$, with the question mark denoting insignificant value (‘don’t care’). It does not matter what is the second element of the missing subtree semantics, because the program will be always correct anyway. In this concrete example, any value of this element is multiplied by zero. As the result of the

subsequent processing at the root node (subtraction), second element of the semantics of entire program is zero, which is the target value.

6.2. Methods

Using the concept of desired semantics a new family of methods for solving problems may be proposed. The common high-level idea of them is to maintain two groups: one of contexts and one of parts, and then to construct complete programs by matching the elements from both of these sets, where the choice of contexts and parts is driven by analysis of desired semantics. This meta-algorithm may be implemented in many ways, especially in an evolutionary manner. For example, one may evolve separately two populations, one of contexts and one of subtrees, or just a single population of full trees (individuals), which would be virtually divided into contexts and subtrees. In this thesis, we propose several variants of the latter concept (i.e., the evolution of a single population), which are quite simple extensions to the classical GP algorithm described in Section 2.5.

In our proposition, we simply enable the evolution to use, in addition to mutation or crossover, a new breeding operator which combines a selected context extracted from a single parent individual and the best matching subtree from a library of available subtrees. All variants of this operator, presented below in this section, differ only in how the context from a parent is chosen.

In the proposed approach, the library of available subtrees contains all subtrees extracted from all individuals from the present population. This means, that in every generation the set of all subtrees existing in current individuals is created from scratch. However, if two or more subtrees have the same semantics, then only the one with the minimal subtree depth[1] is stored in the set. Such restriction to minimal subtrees with unique behaviors drastically reduces the size of this library. There are two reasons to it. Firstly, the majority of genome fragments in the whole evolved population exists in many copies. Secondly, different genotypes often map to equivalent phenotypes, i.e., have the same semantics.

Below we describe all five variants of the proposed breeding operator. All of them work somehow similarly to the standard subtree-replacing mutation operator [60]. However, instead of generating a new random subtree in place of the old one, they all look for such a subtree in the library which has semantics that matches the best the desired semantics of the context arising from removing the old subtree.

The first variant of the operator called *Random Desired Operator* (*RDO*) removes a *randomly* chosen subtree from the parent and puts in the missing place the best matching subtree found in the library (see Algorithm 6.2). Because RDO compares semantics of

[1] We use subtree depth criterion because we apply the same type of constrain to entire evolutionary process., i.e., we limit the maximal tree depth. In other setups, other measures, e.g., a maximal subtree size (number of nodes), might be more appropriate.

Algorithm 6.2 *Random Desired Operator* (RDO)

```
1: procedure RDO(p)
2:     r ← random crossover point in p
3:     c ← Context(p, r)            ▷ extract a context by removing subtree r from p
4:     s ← DesiredSemantics(c, t)             ▷ calculate for target semantics t
5:     r′ ← SearchLibrary(s)              ▷ find a subtree best matching given s
6:     return child produced by replacing subtree r with r′ in p
7: end procedure
```

Algorithm 6.3 *Steepest Desired Operator* (SDO)

```
1: procedure SDO(p)
2:     (r_old, r_new, d_best) ← (∅, ∅, ∞)
3:     for all r ∈ nodes in p do
4:         c ← Context(p, r)        ▷ extract a context by removing subtree r from p
5:         s ← DesiredSemantics(c, t)         ▷ calculate for target semantics t
6:         r′ ← SearchLibrary(s)          ▷ find a subtree best matching given s
7:         d ← Distance(s, Semantics(r′), penalty)
                        ▷ penalize inconsistent elements in s (see Equation 6.2.1)
8:         if d < d_best then
9:             (r_old, r_new, d_best) ← (r, r′, d)
10:    return child produced by replacing subtree r_old with r_new in p
11: end procedure
```

subtrees with only one desired semantics, then the *undefined* (i.e., both insignificant and inconsistent) elements in this desired semantics may be just ignored without any influence to the final result. In other words, the semantic distance between the desired semantics and the parts in the library is calculated only on the defined elements of the former.

The second variant, *Steepest Desired Operator* (*SDO*) considers all possible locations for subtree replacing in the parent and chooses the one for which a subtree found in the library fits best. In other words, SDO considers all contexts from the parent individual, and for each of them operates as RDO, returning the pair (context, part) characterized by the smallest semantic distance. Technically, SDO calculates the desired semantics for every node in the parent and for each of them it searches the best matching subtree. Eventually, SDO chooses the place of substitution with a minimal distance between found subtree semantics and the desired semantics of appropriate context. A pseudocode of this procedure is presented in Algorithm 6.3.

Because SDO compares similarities between different pairs of desired semantics and semantics of subtrees from the library, the way it handles the undefined elements in the desired semantics becomes important. As it would be undesired to promote contexts with many inconsistent elements, the similarity measure used by SDO adds, for each such element, a big penalty value to the overall distance, increasing so the dissimilarity between

Algorithm 6.4 *Constrained Steepest Desired Operator* (CSDO)

```
1: procedure CSDO(p)
2:     (r_old, r_new, d_best) ← (∅, ∅, ∞)
3:     for all r ∈ nodes in p do
4:         c ← Context(p, r)              ▷ extract a context by removing subtree r from p
5:         s ← DesiredSemantics(c, t)     ▷ calculate for target semantics t
6:         if s contains only consistent elements then
7:             r′ ← SearchLibrary(s)      ▷ find a subtree best matching given s
8:             d ← Distance(s, Semantics(r′), 0)   ▷ ignore inconsistent elements in s
9:             if d < d_best then
10:                (r_old, r_new, d_best) ← (r, r′, d)
11:    return child produced by replacing subtree r_old with r_new in p
12: end procedure
```

the compared semantics:

$$\textsc{Distance}(s_d, s, penalty) = \sum_{i=1}^{N} \begin{cases} 0 & \text{if } s_{d,i} \text{ is insignificant} \\ penalty & \text{if } s_{d,i} \text{ is inconsistent} \\ |s_{d,i} - s_i| & \text{otherwise} \end{cases} \tag{6.2.1}$$

Let us observe that, without such punishment, i.e., by ignoring inconsistent elements (as RDO does), SDO would, in an extreme case, always choose the context in the parent with fully inconsistent desired semantics (i.e., inconsistent on all positions), because all subtrees would perfectly fit to it. Technically, for symbolic regression problems, we add penalty value 10^{15} for each inconsistent element (as the maximal possible error committed on a single element/fitness case is infinite). For Boolean problems the situation is different, because the maximal possible distance between two elements of semantics is finite (the distance between the '*true*' and '*false*' values). Therefore, we treat inconsistent elements just as mismatched.

Alternatively, instead of adding some penalty value to the similarity measure, SDO may consider only the contexts that have minimal number of inconsistent values in their desired semantics. This version is called *Constrained Steepest Desired Operator* (*CSDO*). Because the number of inconsistent elements can only increase with larger context (i.e., extending any context by adding any node never makes its desired semantics less inconsistent as was before this extension), the algorithm can give up visiting any child node if the extended context would have more inconsistent elements. Moreover, as all our benchmark problems are realizable with the available function set and thus the target semantics has all consistent elements, we can restrict the operation of this algorithm only to contexts with fully consistent desired semantics. In this way, it is possible to substantially reduce the computational cost, because fewer subtrees need to be searched in the library. Algorithm 6.4 presents this procedure.

Algorithm 6.5 *Error Desired Operator* (EDO)

```
 1: procedure EDO(p)
 2:     (r_old, s_best, d_best) ← (∅, ∅, −∞)
 3:     for all r ∈ nodes in p do
 4:         c ← Context(p, r)            ▷ extract a context by removing subtree r from p
 5:         s ← DesiredSemantics(c, t)               ▷ calculate for target semantics t
 6:         d ← Distance(s, Semantics(r), 0)        ▷ ignore any inconsistent elements
 7:         if d > d_best then
 8:             (r_old, s_best, d_best) ← (r, s, d)
 9:     r' ← SearchLibrary(s_best)            ▷ find a subtree best matching given s_best
10:     return child produced by replacing subtree r_old with r' in p
11: end procedure
```

Another advantage of CSDO is that it does not need one parameter that was required by SDO — the penalty value for inconsistent elements. How to set this value appropriately is not obvious and depends on the problem domain (more precisely: on the adopted definition of semantics) and the range of possible values in semantics. In our experiments we set the penalty value arbitrary, and it is possible that another value would be better (or worse).

The fourth proposed variant of breeding operator is *Error Desired Operator* (*EDO*). Similarly to SDO, it considers all contexts that can be generated from the parent program, but unlike it, to choose the most promising context, it compares the desired semantics with the semantics of the subtree present at that location in the parent individual. Eventually, EDO selects the subtree for which that difference is the largest, and browses the library for the best matching subtree (see Algorithm 6.5). EDO, like SDO, does not want to promote the inconsistent elements of desired semantics. However, as it maximizes the difference between semantics, in contrast to SDO, EDO simply ignores the inconsistent elements instead of adding any penalty.

The motivation behind this particular design of EDO is to identify the location in the parent program that is the 'most erroneous', i.e., at which the actual partial result of computation diverges the most from the desired outcome. Note however that this reasoning should be taken with a grain of salt, because, in general, such errors do not propagate proportionally to the end of program (the root node). For instance, given a parent individual and two locations in it with errors assessed in the above way, applying EDO at the location with the larger error does not guarantee greater improvement of fitness than for the other location.

A slightly different approach is to explicitly consider only the contexts with the minimal number of inconsistent elements in the desired semantics. We name this variant *Constrained Error Desired Operator* (*CEDO*). A pseudocode of CEDO is shown in Algorithm 6.6. Because CEDO introduces analogous modifications to the EDO as CSDO does to SDO, therefore CEDO can be expected to have similar advantages as CSDO when

Algorithm 6.6 *Constrained Error Desired Operator* (CEDO)

```
1: procedure CEDO(p)
2:     (r_old, s_best, d_best) ← (∅, ∅, −∞)
3:     for all r ∈ nodes in p do
4:         c ← Context(p, r)              ▷ extract a context by removing subtree r from p
5:         s ← DesiredSemantics(c, t)             ▷ calculate for target semantics t
6:         if s contains only consistent elements then
7:             d ← Distance(s, Semantics(r), 0)      ▷ ignore any inconsistent elements
8:             if d > d_best then
9:                 (r_old, s_best, d_best) ← (r, s, d)
10:    r' ← SearchLibrary(s_best)               ▷ find a subtree best matching given s_best
11:    return child produced by replacing subtree r_old with r' in p
12: end procedure
```

comparing them with their base, unconstrained versions.

The common operation performed by the five methods proposed above is searching the library for a subtree with semantics that has the minimal distance to a given desired semantics. This problem is well known as a nearest neighbor search (NNS) [17, 10] (known also under different names, e.g., post office problem [58]). NNS is an optimization problem of finding in a set of points in a metric space (usually a highly-dimensional one) the one that is the closest to a given point. This problem is not trivial and, especially in highly-dimensional space, computationally demanding.

To effectively realize this task, many approximate nearest neighbor search algorithms were proposed [3, 47, 96, 38]. However, all these algorithms are not prepared to handle unknown values on arbitrary dimensions (i.e., the point in question must be fully defined in the same space as all stored points). Therefore, we use a straightforward algorithm which is able to cope with missing values — the simplest, yet also the slowest, linear search through all points (i.e., in our case: through semantics of all stored subtrees).

6.3. Experiments

To verify performance of all methods described in the previous section, we tested our approach on all 39 benchmark problems described in Chapter 4.2. We conduct a series of experiments involving two genetic operators simultaneously in each of them:

1. one helper operator: either standard crossover (X) or mutation (M), and
2. one from all proposed methods exploiting information about desired semantics, i.e., RDO, SDO, CSDO, EDO, or CEDO.

In the set of control experiments both standard crossover and mutation are used together.

For each pair of operators we tested different proportions of their probabilities varying from 0 to 1 with step 0.1 (resulting in eleven different setups for each pair). Setups are

generally denoted as $O_1 + O_2\,\beta$, where O_1 and O_2 are symbols of used operators, and β is the probability of operator O_2 (i.e., $P(O_1) = 1 - \beta$). When $\beta = 0$ or $\beta = 1$ then the notion simplifies to $O_1\,1.0$ or $O_2\,1.0$, respectively.

Basic parameters of evolution used in these experiments are the same as in previous experiments (see Table 4.3 on page 43). Because we wanted to verify the positive effect of our proposed operators, in our experiments we checked influence of a different proportion of them. We did not test varying values of any other parameters.

In following, by *xDO* we will call any of our proposed operators (RDO, SDO, CSDO, EDO, CEDO) if it does not matter which from these five operators is involved.

After excluding redundant setups with disabled desired operators (i.e. $\beta = 0$ which equals to one of the control setup) or disabled helper operator (setups $X + xDO\,1.0$ is identical to $M + xDO\,1.0$ and are denoted simply as $xDO\,1.0$) we have 106 ($2\times5\times9+5+11$) distinct setups together with all 11 control experiments. Each setup was tested on all 39 problems. To get statistical significant results each setup was run independently 200 times with different seeds of a pseudo random number generator, which give us $106 \times 39 \times 200 = 826800$ total number of evolutionary runs.

6.4. Results

6.4.1. Qualitative Results

In this subsection we present qualitative outcomes obtained in our experiments. The shown results come generally from statistical analysis and therefore they are quite easily interpretable.

Table 6.1 depicts the ranks computed through the Friedman test comparing success ratio of all setups tested on all problems. Setups with the best proportion of used operators are emphasized with a bold font. Tables 6.2 and 6.3 present appropriate ranks of setups tested separately on symbolic regression or Boolean problems appropriately.

It is worth to remind that calculating per problem ranks (the first phase of calculating the final values in the presented rankings — see Section 4.5.3), in case of ties, i.e., if several setups have equal values of analyzed characteristic (success rate, error, etc.) the assigned ranks are averaged and all setups get the same value. The values in presented rankings are averaged over all considered problems. This is the reason why the best setup (not worse on any problem than any other method) might have the final rank much greater than one (the more problems with ties the greater rank).

From the presented tables, it results that *M+RDO 0.3, M+SDO 0.3,* and *X+RDO 0.5* are the best setups (with almost negligible differences in ranking of success ratio) and these three pairs of operators outperforms others in case of solving all problems. Setups *M+RDO 0.2–0.5, M+SDO 0.3–0.5,* and *X+RDO 0.4–0.5* are better than the next best setup with operators *X+SDO*.

Setup	*Rank*	*Setup*	*Rank*	*Setup*	*Rank*
M+RDO 0.3	**23.83**	X+RDO 0.7	29.94	X+CEDO 0.4	61.31
M+SDO 0.3	**23.90**	M+SDO 0.1	30.22	X+CEDO 0.2	61.35
X+RDO 0.5	**24.94**	*(skipped 8 items)*		X+EDO 0.3	61.41
M+RDO 0.4	25.18	**X+CSDO 0.2**	**35.38**	X+EDO 0.4	61.62
X+RDO 0.4	25.54	M+RDO 0.1	35.82	*(skipped 3 items)*	
M+SDO 0.5	25.63	**M+CSDO 0.3**	**36.04**	**X 1.0**	**67.09**
M+RDO 0.5	25.81	X+CSDO 0.1	36.13	**M+CEDO 0.2**	**69.44**
M+SDO 0.4	25.85	X+CSDO 0.3	37.06	**X+M 0.1**	**69.76**
M+RDO 0.2	26.29	X+SDO 0.9	37.09	X+CEDO 0.6	69.86
X+SDO 0.6	**26.58**	M+CSDO 0.2	37.13	M+CEDO 0.4	70.63
M+SDO 0.6	27.01	M+SDO 0.9	38.06	X+M 0.2	70.69
M+RDO 0.7	27.17	**RDO 1.0**	**38.23**	M+CEDO 0.3	70.76
X+SDO 0.5	27.23	X+CSDO 0.4	38.23	X+EDO 0.6	71.29
M+RDO 0.8	27.50	M+CSDO 0.1	38.46	**M+EDO 0.3**	**73.62**
X+SDO 0.4	27.95	M+CSDO 0.4	38.67	*(skipped 3 items)*	
X+RDO 0.3	28.05	M+CSDO 0.5	39.24	M+EDO 0.2	74.55
X+SDO 0.2	28.12	X+CSDO 0.6	39.42	*(skipped 4 items)*	
M+SDO 0.2	28.33	*(skipped 6 items)*		**CSDO 1.0**	**77.68**
X+SDO 0.7	28.33	**SDO 1.0**	**51.78**	*(skipped 17 items)*	
M+RDO 0.6	28.36	*(skipped 2 items)*		**M 1.0**	**88.33**
X+SDO 0.3	28.38	**X+CEDO 0.1**	**58.97**	*(skipped 2 items)*	
X+RDO 0.2	29.31	X+CEDO 0.3	59.00	**CEDO 1.0**	**92.73**
X+SDO 0.1	29.87	**X+EDO 0.1**	**60.22**	**EDO 1.0**	**92.81**

Table 6.1.: Friedman ranks of *success ratio* performance on all 39 problems (both symbolic regression and Boolean domain).

Setup	*Rank*	*Setup*	*Rank*	*Setup*	*Rank*
M+CSDO 0.3	**27.32**	M+CSDO 0.6	33.95	X+EDO 0.2	53.79
M+SDO 0.3	**27.32**	M+SDO 0.6	33.95	*(skipped 3 items)*	
M+RDO 0.3	**30.82**	X+RDO 0.2	34.45	**M+CEDO 0.4**	**56.92**
X+CSDO 0.6	**31.68**	X+CSDO 0.1	34.50	M+CEDO 0.2	57.08
X+SDO 0.6	**31.68**	M+RDO 0.2	34.55	M+CEDO 0.3	58.29
M+CSDO 0.5	32.34	X+SDO 0.1	34.55	*(skipped 2 items)*	
M+SDO 0.5	32.34	M+CSDO 0.1	34.74	**RDO 1.0**	**61.24**
X+SDO 0.2	32.39	M+SDO 0.1	34.74	*(skipped 2 items)*	
M+CSDO 0.2	32.42	M+RDO 0.4	34.92	**M+EDO 0.3**	**64.16**
M+SDO 0.2	32.42	M+RDO 0.5	36.32	M+EDO 0.4	67.00
X+CSDO 0.4	32.53	X+RDO 0.3	36.63	**X 1.0**	**67.21**
X+SDO 0.4	32.53	X+RDO 0.1	37.21	*(skipped 2 items)*	
X+CSDO 0.2	32.55	M+RDO 0.7	39.39	**CSDO 1.0**	**68.42**
M+CSDO 0.4	32.61	*(skipped 9 items)*		**SDO 1.0**	**68.42**
M+SDO 0.4	32.61	**X+CEDO 0.3**	**45.05**	*(skipped 3 items)*	
X+CSDO 0.3	32.66	*(skipped 4 items)*		**X+M 0.1**	**71.61**
X+SDO 0.3	32.66	X+CEDO 0.4	47.97	*(skipped 2 items)*	
X+RDO 0.4	**33.05**	**X+EDO 0.4**	**48.61**	X+M 0.2	75.74
X+CSDO 0.5	33.18	X+CEDO 0.5	49.29	*(skipped 12 items)*	
X+SDO 0.5	33.18	X+EDO 0.3	50.00	**CEDO 1.0**	**90.95**
X+RDO 0.5	33.45	X+EDO 0.5	50.50	**EDO 1.0**	**91.11**
X+CSDO 0.7	33.63	*(skipped 5 items)*		*(skipped 3 items)*	
X+SDO 0.7	33.63	X+CEDO 0.2	53.00	**M 1.0**	**93.82**

Table 6.2.: Friedman ranks of *success ratio* performance on 19 problems from symbolic regression domain.

Setup	*Rank*	*Setup*	*Rank*	*Setup*	*Rank*
M+RDO 0.9	**15.40**	**X+SDO 0.5**	**21.58**	**X+M 0.2**	**65.90**
M+RDO 0.7	15.55	X+SDO 0.6	21.73	**X 1.0**	**66.98**
M+RDO 0.6	15.70	M+RDO 0.1	21.75	X+M 0.4	67.78
X+RDO 0.6	**15.78**	X+SDO 0.8	22.70	X+M 0.1	68.00
M+RDO 0.5	15.83	X+SDO 0.9	23.08	X+M 0.3	68.48
M+RDO 0.4	15.93	X+SDO 0.7	23.30	X+CEDO 0.2	69.28
X+RDO 0.9	15.93	X+SDO 0.4	23.60	X+EDO 0.2	69.28
M+RDO 0.8	16.18	*(skipped 8 items)*		*(skipped 3 items)*	
X+RDO 0.7	16.35	**SDO 1.0**	**35.98**	X+CEDO 0.3	72.25
RDO 1.0	**16.38**	**X+CSDO 0.1**	**37.68**	X+EDO 0.3	72.25
X+RDO 0.8	16.40	X+CSDO 0.2	38.08	*(skipped 6 items)*	
X+RDO 0.5	16.85	X+CSDO 0.3	41.25	**M+CEDO 0.1**	**80.35**
M+RDO 0.3	17.20	**M+CSDO 0.2**	**41.60**	**M+EDO 0.1**	**80.35**
X+RDO 0.4	18.40	M+CSDO 0.1	42.00	M+CEDO 0.2	81.18
M+RDO 0.2	18.45	X+CSDO 0.4	43.65	M+EDO 0.2	81.18
M+SDO 0.5	**19.25**	M+CSDO 0.3	44.33	*(skipped 2 items)*	
M+SDO 0.4	19.43	M+CSDO 0.4	44.43	**M 1.0**	**83.13**
X+RDO 0.3	19.90	X+CSDO 0.5	45.70	*(skipped 6 items)*	
M+SDO 0.6	20.43	M+CSDO 0.5	45.80	**CSDO 1.0**	**86.48**
M+SDO 0.7	20.48	*(skipped 8 items)*		*(skipped 14 items)*	
M+SDO 0.3	20.65	**X+CEDO 0.1**	**63.58**	**CEDO 1.0**	**94.43**
M+SDO 0.8	21.53	**X+EDO 0.1**	**63.58**	**EDO 1.0**	**94.43**

Table 6.3.: Friedman ranks of *success ratio* performance on 20 problems from Boolean domain.

If symbolic regression problems are tested separately then *M+SDO 0.3* together with *M+CSDO 0.3* (which gives almost the same results) slightly outperform *M+RDO 0.3*. Setup *X+SDO 0.6* (and *X+CSDO 0.6*) is third before the *X+RDO 0.4* — the next best pair of operators. If only Boolean problems are solved, most setups with RDO operator, especially *M+RDO 0.9* and *X+RDO 0.6*, clearly outperforms others. The second, well behaving operator is SDO, especially *M+SDO 0.5* setup.

It is important to notice, that the difference between SDO and CSDO in practice strongly depends on the penalty value used by SDO (see Formula 6.2.1). As mentioned earlier, for symbolic regression domain, we use penalty value of 10^{15}. Therefore, as long as the distance between compared semantics is less than 10^{15} (the overwhelming majority of cases), there would be no difference in effect between SDO and CSDO. However, for the Boolean problems the penalty has only a unitary value and it could be easily dominated (majority of cases) by the distance resulting from an ordinary disparity between elements. Therefore, the SDO and CSDO give quite different results for all benchmarks with Boolean problems.

Rankings presented in Tables 6.1–6.3 clearly show that, considering success ratio, the best setups are those with the either RDO or SDO. The successive good operators are CSDO, CEDO, and the worst from the proposed operator is EDO. Despite the fact that this order may change a bit depending on the domain of solved problems, the RDO setups always are in the lead. We recommend SDO as the second operator, especially for symbolic

Setup	*Rank*	*Setup*	*Rank*	*Setup*	*Rank*
M+RDO 0.5	**23.18**	X+SDO 0.4	30.73	M+CSDO 0.8	54.40
M+RDO 0.6	23.19	**RDO 1.0**	**30.88**	X+CEDO 0.3	54.91
M+RDO 0.7	23.22	X+RDO 0.1	31.15	X+EDO 0.2	55.14
M+RDO 0.4	23.64	X+SDO 0.5	31.94	X+CSDO 0.8	57.49
M+RDO 0.8	23.68	M+SDO 0.5	32.06	X+EDO 0.3	59.15
X+RDO 0.7	**24.26**	**X+CSDO 0.1**	**33.81**	X+CEDO 0.4	59.90
X+RDO 0.8	24.38	X+CSDO 0.2	34.24	**SDO 1.0**	**60.28**
M+RDO 0.3	24.65	M+SDO 0.6	34.46	**M+CEDO 0.3**	**61.56**
X+RDO 0.6	24.73	X+SDO 0.6	34.62	M+CEDO 0.2	62.21
M+RDO 0.9	25.35	**M+CSDO 0.3**	**35.51**	M+CEDO 0.4	63.35
X+RDO 0.5	25.40	M+CSDO 0.2	36.01	*(skipped 2 items)*	
X+RDO 0.4	25.67	X+CSDO 0.3	36.67	**X 1.0**	**64.69**
X+RDO 0.3	25.78	X+SDO 0.7	36.69	**X+M 0.1**	**64.99**
X+RDO 0.9	26.14	X+CSDO 0.4	37.44	X+M 0.2	65.09
M+RDO 0.2	26.21	M+SDO 0.7	37.59	*(skipped 4 items)*	
X+RDO 0.2	27.81	M+CSDO 0.1	37.95	**M+EDO 0.2**	**67.62**
M+SDO 0.3	**28.00**	M+CSDO 0.4	38.73	M+EDO 0.3	67.72
M+SDO 0.2	28.79	X+CSDO 0.5	40.68	*(skipped 20 items)*	
M+RDO 0.1	29.60	M+CSDO 0.5	41.35	**M 1.0**	**86.59**
X+SDO 0.2	**29.67**	*(skipped 7 items)*		*(skipped 3 items)*	
X+SDO 0.1	29.92	**X+CEDO 0.1**	**49.13**	**CSDO 1.0**	**90.24**
X+SDO 0.3	29.92	X+CSDO 0.7	49.38	*(skipped 4 items)*	
M+SDO 0.4	30.45	X+CEDO 0.2	50.42	**EDO 1.0**	**101.79**
M+SDO 0.1	30.64	**X+EDO 0.1**	**53.90**	**CEDO 1.0**	**101.90**

Table 6.4.: Friedman ranks of *median error* on training set performed on all 39 problems (both symbolic regression and Boolean domain).

regression problems where the first 17 places in the ranking took 16 setups with different proportion of either SDO or CSDO.

On the other hand, setups with CEDO or EDO do not seem to be a good choice as very often, if the probability is badly chosen, they give worse results than the best control experiments which use only standard crossover and mutation. It is worth to notice that using either EDO or CEDO operator alone gives mostly worse results than just a simple mutation (*M 1.0*). Therefore, we conclude that the strategy to chose a replaced subtree used by EDO is not good for maximizing success ratio (the probability of finding an ideal solution).

When comparing median errors obtained on training set, the order of proposed operators (see Table 6.4, 6.5, and 6.6) are almost the same as comparing the probability of successes. Here, however, higher probability of RDO is more advantageous. Moreover, it is worth to notice that in Table 6.6 presenting errors on Boolean problems the first 31 setups have first place *ex aequo*. This happens because only seven setups with RDO or SDO have success rate below *0.5* on some problem (*M+RDO 0.*1, *X+RDO 0.1–0.2*, and *M+SDO 0.1* only for problem NPAR6; *X+SDO 0.8–0.9* only for ICMP8; *SDO 1.0* for NCMP6, NCMP8, NPAR6, and ICMP8), and therefore the rest RDO/SDO setups have median error equal to zero.

Table 6.7 presents errors made on testing set (only symbolic regression problems). The

Setup	*Rank*	*Setup*	*Rank*	*Setup*	*Rank*
M+RDO 0.5	**18.87**	X+SDO 0.1	32.71	X+CEDO 0.5	50.92
M+RDO 0.6	18.89	X+SDO 0.3	32.71	**X+EDO 0.3**	**54.29**
M+RDO 0.7	18.95	M+CSDO 0.1	33.24	M+CEDO 0.5	54.71
M+RDO 0.4	19.82	M+SDO 0.1	33.24	X+EDO 0.2	56.11
M+RDO 0.8	19.89	M+SDO 0.4	33.79	M+CEDO 0.1	56.34
X+RDO 0.7	**21.08**	M+CSDO 0.4	33.95	X+EDO 0.4	56.34
X+RDO 0.8	21.34	X+RDO 0.1	34.13	*(skipped 5 items)*	
M+RDO 0.3	21.89	X+CSDO 0.4	34.21	**M+EDO 0.3**	**58.92**
X+RDO 0.6	22.05	X+SDO 0.4	34.37	X+EDO 0.5	59.08
M+RDO 0.9	23.32	**RDO 1.0**	**34.68**	M+EDO 0.4	59.13
X+RDO 0.5	23.42	X+CSDO 0.5	36.79	X+EDO 0.1	59.39
X+RDO 0.4	23.97	X+SDO 0.5	36.84	M+EDO 0.2	60.76
X+RDO 0.3	24.21	M+CSDO 0.5	37.05	*(skipped 16 items)*	
X+RDO 0.9	24.95	M+SDO 0.5	37.11	**X 1.0**	**81.21**
M+RDO 0.2	25.08	M+CSDO 0.6	42.03	**X+M 0.1**	**81.89**
X+RDO 0.2	27.42	*(skipped 3 items)*		M+EDO 0.8	82.39
M+CSDO 0.3	**28.76**	**X+CEDO 0.3**	**45.58**	X+M 0.2	82.53
M+SDO 0.3	**28.76**	**M+CEDO 0.3**	**46.29**	*(skipped 5 items)*	
M+CSDO 0.2	30.39	X+CEDO 0.2	46.42	**CSDO 1.0**	**90.61**
M+SDO 0.2	30.39	*(skipped 2 items)*		**SDO 1.0**	**90.61**
M+RDO 0.1	31.00	X+CEDO 0.4	48.18	*(skipped 4 items)*	
X+SDO 0.2	**32.18**	*(skipped 2 items)*		**EDO 1.0**	**98.58**
X+CSDO 0.2	**32.29**	M+CEDO 0.4	49.29	**CEDO 1.0**	**98.79**
X+CSDO 0.1	32.71	X+CEDO 0.1	49.61	*(skipped 2 items)*	
X+CSDO 0.3	32.71	M+CEDO 0.2	49.66	**M 1.0**	**101.58**

Table 6.5.: Friedman ranks of *median error* on training set performed on 19 problems from symbolic regression domain.

Setup	*Rank*	*Setup*	*Rank*	*Setup*	*Rank*
M+RDO 0.2	**27.28**	X+RDO 0.8	27.28	**X+EDO 0.1**	**48.68**
M+RDO 0.3	27.28	X+RDO 0.9	27.28	X+M 0.1	48.93
M+RDO 0.4	27.28	**X+SDO 0.1**	**27.28**	**X 1.0**	**49.00**
M+RDO 0.5	27.28	X+SDO 0.2	27.28	X+M 0.3	49.28
M+RDO 0.6	27.28	X+SDO 0.3	27.28	X+M 0.4	49.50
M+RDO 0.7	27.28	X+SDO 0.4	27.28	*(skipped 5 items)*	
M+RDO 0.8	27.28	X+SDO 0.5	27.28	X+CEDO 0.2	54.23
M+RDO 0.9	27.28	*(skipped 8 items)*		X+EDO 0.2	54.23
M+SDO 0.2	**27.28**	**SDO 1.0**	**31.48**	*(skipped 6 items)*	
M+SDO 0.3	27.28	**X+CSDO 0.1**	**34.85**	X+CEDO 0.3	63.78
M+SDO 0.4	27.28	X+CSDO 0.2	36.10	X+EDO 0.3	63.78
M+SDO 0.5	27.28	X+CSDO 0.3	40.43	*(skipped 2 items)*	
M+SDO 0.6	27.28	X+CSDO 0.4	40.50	**M 1.0**	**72.35**
M+SDO 0.7	27.28	**M+CSDO 0.2**	**41.35**	**M+CEDO 0.1**	**72.40**
M+SDO 0.8	27.28	M+CSDO 0.3	41.93	**M+EDO 0.1**	**72.40**
M+SDO 0.9	27.28	M+CSDO 0.1	42.43	M+CEDO 0.2	74.13
RDO 1.0	**27.28**	M+CSDO 0.4	43.28	M+EDO 0.2	74.13
X+RDO 0.3	**27.28**	X+CSDO 0.5	44.38	*(skipped 14 items)*	
X+RDO 0.4	27.28	M+CSDO 0.5	45.43	**CSDO 1.0**	**89.90**
X+RDO 0.5	27.28	*(skipped 2 items)*		*(skipped 10 items)*	
X+RDO 0.6	27.28	**X+M 0.2**	**48.53**	**CEDO 1.0**	**104.85**
X+RDO 0.7	27.28	**X+CEDO 0.1**	**48.68**	**EDO 1.0**	**104.85**

Table 6.6.: Friedman ranks of *median error* on training set performed on 20 problems from Boolean domain.

Setup	*Rank*	*Setup*	*Rank*	*Setup*	*Rank*
X+RDO 0.5	**19.84**	X+SDO 0.5	31.97	**X+EDO 0.3**	**51.45**
X+RDO 0.8	20.24	**M+CSDO 0.3**	**33.00**	X+EDO 0.4	53.55
X+RDO 0.6	20.74	**M+SDO 0.3**	**33.11**	X+CEDO 0.6	53.92
X+RDO 0.4	20.84	M+RDO 0.1	33.63	M+CEDO 0.5	54.87
X+RDO 0.7	20.95	**RDO 1.0**	**34.05**	X+CSDO 0.8	56.11
X+RDO 0.3	21.24	M+CSDO 0.2	35.13	X+EDO 0.5	56.13
M+RDO 0.4	**22.61**	M+SDO 0.2	35.13	X+SDO 0.8	56.16
M+RDO 0.5	22.66	M+SDO 0.4	38.24	X+EDO 0.2	56.32
M+RDO 0.3	23.34	X+CSDO 0.6	38.37	*(skipped 3 items)*	
M+RDO 0.6	24.05	M+CSDO 0.4	38.45	**M+EDO 0.3**	**59.39**
X+RDO 0.2	24.37	X+SDO 0.6	38.63	M+EDO 0.4	59.50
M+RDO 0.7	24.79	M+CSDO 0.1	38.76	*(skipped 4 items)*	
X+RDO 0.9	25.26	M+SDO 0.1	38.76	M+EDO 0.2	63.24
M+RDO 0.8	26.47	M+CSDO 0.5	39.21	*(skipped 13 items)*	
M+RDO 0.9	26.63	M+SDO 0.5	39.32	**X 1.0**	**80.71**
M+RDO 0.2	26.97	**X+CEDO 0.3**	**42.37**	**X+M 0.1**	**81.71**
X+SDO 0.2	**28.50**	*(skipped 2 items)*		M+EDO 0.8	82.08
X+CSDO 0.2	**28.66**	X+CEDO 0.2	44.37	X+M 0.2	82.55
X+CSDO 0.3	28.92	*(skipped 2 items)*		*(skipped 4 items)*	
X+SDO 0.3	29.03	X+CEDO 0.4	46.34	**CSDO 1.0**	**90.08**
X+CSDO 0.4	30.34	**M+CEDO 0.3**	**46.66**	**SDO 1.0**	**90.08**
X+SDO 0.1	30.45	X+CEDO 0.1	48.42	*(skipped 5 items)*	
X+CSDO 0.1	30.50	X+CEDO 0.5	48.76	**EDO 1.0**	**98.53**
X+SDO 0.4	30.55	M+CEDO 0.4	49.66	**CEDO 1.0**	**98.95**
X+RDO 0.1	31.63	*(skipped 2 items)*		*(skipped 2 items)*	
X+CSDO 0.5	31.97	M+CEDO 0.2	51.39	**M 1.0**	**102.37**

Table 6.7.: Friedman ranks of *median error* on *testing* set performed on 19 problems from symbolic regression domain.

order of setups looks also very similarly to the previous rankings. This suggest that, RDO is generally beneficial and setups with this operator reliably outperforms others.

It is interesting, that SDO performs generally worse than RDO (the only exception is success ratio ranking for symbolic regression) which may be a little surprising. It would appear that the SDO operator which checks all nodes in an individual for the best place of substituting it subtree should perform better than doing this just for a randomly selected node. The experiments, however, clearly shows that this is not true, and setups SDO gives greater error than RDO. Probably, this happens because such steepest approach is too greedy and leads too quickly to a local optima, especially when the library still lacks good, desirable subtrees. On the other hand, this is a good news, because RDO works much faster than SDO, which is additional advantage of RDO.

From ranks presented in Tables 6.1–6.7 it arises that most beneficial probability of RDO is about 0.3–0.5, and 0.3–0.6 (or 0.1–0.4 if minimizing error) of SDO. In general, it seems that problems from the Boolean domain are easier to solve if RDO is used more frequently than in case of solving problems from symbolic regression domain when RDO could be used less often. Nevertheless, the results show that the evolution is not much sensitive to the exact probability of RDO and SDO, as the range of its values is quite broad for the well performing setups.

We compared also each setup with all control experiments to check which setups are advantageous and which are not. Therefore, we perform series of Friedman tests comparing 12 setups together (1 setup in question and 11 control setups) followed by Holm's post-hoc statistical procedure. We compared both achieved success ratio and error on training sets for all 39 problems. Tables 6.8 and 6.9 shows the final results of these tests, where letter s means statistically important (p-value < 0.05) advantage (white background) or disadvantage (gray background) of a setup in the row comparing with control setup in the column. Letter e indicates statistical difference between medians of obtained errors on training set.

These tables confirm all previous conclusions. Especially, they show that all setups with RDO outperforms all control experiments. This is particularly interesting because it means that any chosen proportion between RDO and either mutation or crossover operators is statistically beneficial. In the case of SDO, this operator is profitable if it is combined with either mutation or crossover — then both success ratio and errors are significantly better. When SDO is applied alone, success rates are still better, but there is not always a significant difference in errors. Similarly, CSDO is advantageous if it is not too frequently applied. Using CSDO alone is mainly deteriorating. Setups with either EDO or CEDO are statistically worse in many cases, irrespectively of its probability, and therefore it seems hazardous to use them at all.

To conclude, Figure 6.4.1 visualize graphically which setups are statistically better than other (p-value < 0.05) in the sense of achieving success ratios (*cf.* Table 6.1). The results

	X 1.0	X+M 0.1	X+M 0.2	X+M 0.3	X+M 0.4	X+M 0.5	X+M 0.6	X+M 0.7	X+M 0.8	X+M 0.9	M 1.0
M+RDO 0.1	se	se	se	se	se	se	se	se	se	se	se
M+RDO 0.2	se	se	se	se	se	se	se	se	se	se	se
M+RDO 0.3	se	se	se	se	se	se	se	se	se	se	se
M+RDO 0.4	se	se	se	se	se	se	se	se	se	se	se
M+RDO 0.5	se	se	se	se	se	se	se	se	se	se	se
M+RDO 0.6	se	se	se	se	se	se	se	se	se	se	se
M+RDO 0.7	se	se	se	se	se	se	se	se	se	se	se
M+RDO 0.8	se	se	se	se	se	se	se	se	se	se	se
M+RDO 0.9	se	se	se	se	se	se	se	se	se	se	se
RDO 1.0	se	se	se	se	se	se	se	se	se	se	se
M+SDO 0.1	se	se	se	se	se	se	se	se	se	se	se
M+SDO 0.2	se	se	se	se	se	se	se	se	se	se	se
M+SDO 0.3	se	se	se	se	se	se	se	se	se	se	se
M+SDO 0.4	se	se	se	se	se	se	se	se	se	se	se
M+SDO 0.5	se	se	se	se	se	se	se	se	se	se	se
M+SDO 0.6	se	se	se	se	se	se	se	se	se	se	se
M+SDO 0.7	se	se	se	se	se	se	se	se	se	se	se
M+SDO 0.8	se	se	se	se	se	se	se	se	se	se	se
M+SDO 0.9	se	se	se	se	se	se	se	se	se	se	se
SDO 1.0	s	s	s	s	s	s	s	s	se	se	se
M+CSDO 0.1	s	s	s	se	se	se	se	se	se	se	se
M+CSDO 0.2	s	s	s	se	se	se	se	se	se	se	se
M+CSDO 0.3	s	s	s	se	se	se	se	se	se	se	se
M+CSDO 0.4	s	s	s	s	s	se	se	se	se	se	se
M+CSDO 0.5	s	s	s	s	s	se	se	se	se	se	se
M+CSDO 0.6	s	s	s	s	s	s	se	se	se	se	se
M+CSDO 0.7	s	s	s	s	s	s	se	se	se	se	se
M+CSDO 0.8	s	s	s	s	s	s	se	se	se	se	se
M+CSDO 0.9	s	s	s	s	s	s	s	se	se	se	se
CSDO 1.0	se	se	se	e	e	e	e	e			s
M+EDO 0.1			s							e	se
M+EDO 0.2										se	se
M+EDO 0.3										se	se
M+EDO 0.4										e	se
M+EDO 0.5										e	se
M+EDO 0.6	se	se	se							e	e
M+EDO 0.7	se	se	se	e							se
M+EDO 0.8	se	se	se	se	e						e
M+EDO 0.9	se	se	se	se	se	e	e				
EDO 1.0	se	se	se	se	se	e	e	e	e	e	
M+CEDO 0.1									e	se	se
M+CEDO 0.2										se	se
M+CEDO 0.3										se	se
M+CEDO 0.4										se	se
M+CEDO 0.5										se	se
M+CEDO 0.6	se	se	se							e	e
M+CEDO 0.7	se	se	se							e	se
M+CEDO 0.8	se	se	se	se	e						e
M+CEDO 0.9	se	se	se	se	se	e					e
CEDO 1.0	se	se	se	se	se	e	e	e	e	e	

Table 6.8.: Results of statistical tests comparing setups *M+xDO* (mutation and every desired operator) in rows with all control experiments in columns. Letters show significant (p-value < 0.05) improvement (white background) or decline (gray background): s — probability of success, e — median error on training set.

	X 1.0	X+M 0.1	X+M 0.2	X+M 0.3	X+M 0.4	X+M 0.5	X+M 0.6	X+M 0.7	X+M 0.8	X+M 0.9	M 1.0
X+RDO 0.1	se	se	se	se	se	se	se	se	se	se	se
X+RDO 0.2	se	se	se	se	se	se	se	se	se	se	se
X+RDO 0.3	se	se	se	se	se	se	se	se	se	se	se
X+RDO 0.4	se	se	se	se	se	se	se	se	se	se	se
X+RDO 0.5	se	se	se	se	se	se	se	se	se	se	se
X+RDO 0.6	se	se	se	se	se	se	se	se	se	se	se
X+RDO 0.7	se	se	se	se	se	se	se	se	se	se	se
X+RDO 0.8	se	se	se	se	se	se	se	se	se	se	se
X+RDO 0.9	se	se	se	se	se	se	se	se	se	se	se
RDO 1.0	se	se	se	se	se	se	se	se	se	se	se
X+SDO 0.1	se	se	se	se	se	se	se	se	se	se	se
X+SDO 0.2	se	se	se	se	se	se	se	se	se	se	se
X+SDO 0.3	se	se	se	se	se	se	se	se	se	se	se
X+SDO 0.4	se	se	se	se	se	se	se	se	se	se	se
X+SDO 0.5	se	se	se	se	se	se	se	se	se	se	se
X+SDO 0.6	se	se	se	se	se	se	se	se	se	se	se
X+SDO 0.7	se	se	se	se	se	se	se	se	se	se	se
X+SDO 0.8	se	se	se	se	se	se	se	se	se	se	se
X+SDO 0.9	se	se	se	se	se	se	se	se	se	se	se
SDO 1.0	s	s	s	s	s	s	s	s	se	se	se
X+CSDO 0.1	se	se	se	se	se	se	se	se	se	se	se
X+CSDO 0.2	s	s	s	se	se	se	se	se	se	se	se
X+CSDO 0.3	s	s	s	s	se	se	se	se	se	se	se
X+CSDO 0.4	s	s	s	s	se	se	se	se	se	se	se
X+CSDO 0.5	s	s	s	s	s	se	se	se	se	se	se
X+CSDO 0.6	s	s	s	s	s	se	se	se	se	se	se
X+CSDO 0.7	s	s	s	s	s	s	se	se	se	se	se
X+CSDO 0.8	s	s	s	s	s	s	s	se	se	se	se
X+CSDO 0.9				s	s	s	s	s	se	se	se
CSDO 1.0	se	se	se	e	e	e	e	e			s
X+EDO 0.1					s	se	se	se	se	se	se
X+EDO 0.2							se	se	se	se	se
X+EDO 0.3							s	se	se	se	se
X+EDO 0.4							s	se	se	se	se
X+EDO 0.5									s	se	se
X+EDO 0.6	e	e	e							se	se
X+EDO 0.7	e	e	e							s	se
X+EDO 0.8	se	se	se	e	e						se
X+EDO 0.9	se	se	se	e	e	e	e				
EDO 1.0	se	se	se	se	se	e	e	e	e	e	
X+CEDO 0.1					s	se	se	se	se	se	se
X+CEDO 0.2							se	se	se	se	se
X+CEDO 0.3							s	se	se	se	se
X+CEDO 0.4							s	se	se	se	se
X+CEDO 0.5									s	se	se
X+CEDO 0.6	e	e	e							se	se
X+CEDO 0.7	e	e	e							se	se
X+CEDO 0.8	e	e	e	e							se
X+CEDO 0.9	se	se	se	e	e	e					e
CEDO 1.0	se	se	se	se	se	e	e	e	e	e	

Table 6.9.: Results of statistical tests comparing setups *X+xDO* (crossover and every desired operator) in rows with all control experiments in columns. Letters show significant (p-value < 0.05) improvement (white background) or decline (gray background): s — probability of success, e — median error on training set.

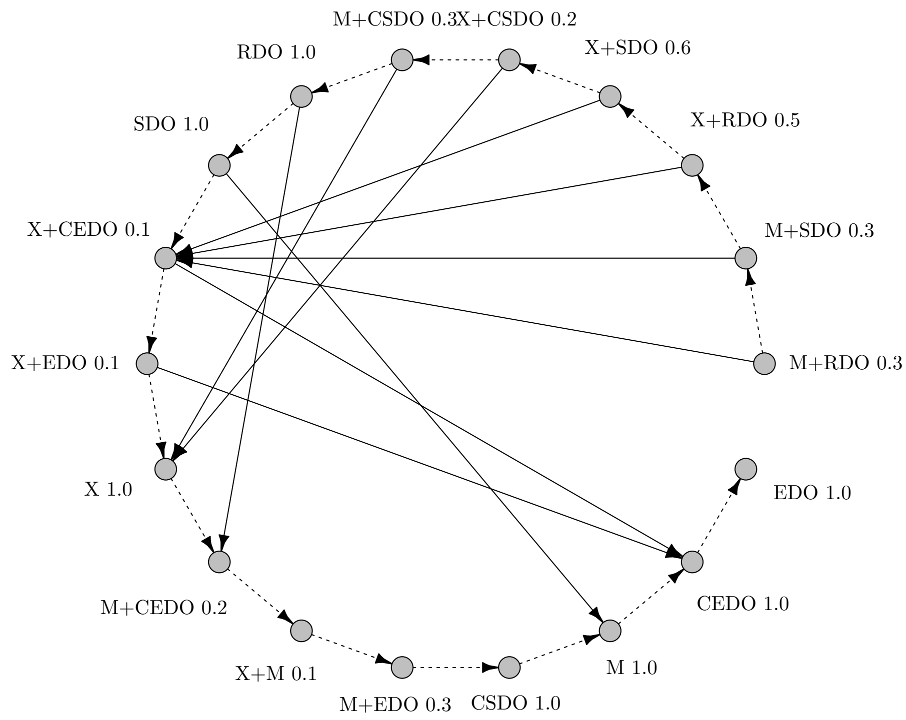


Figure 6.4.1.: Subset of the ranking presented in Table 6.1. Dotted arrows show order of ranks, solid arrows shows the first setup in the ranking which is statistically significantly (Friedman test with Shaffer's post-hoc procedure, p-value < 0.05) worse.

are obtained by Friedman test with Shaffer's post-hoc procedure which compares all 106 possible pairs of tested setups, but on the graph only 18 best setups from each pair of used operators are shown (denoted by bold font in Table 6.1). The dotted arrows show order of setups in the ranking. Solid arrows point the first, statistically significantly worse setup from the ranking, so this means that if some method A statistically outperforms some other method B, then it simultaneously outperforms all methods following B in the ranking (dotted arrows).

It appears, that from setups shown in this chart the first six ones (M+RDO 0.3, M+SDO 0.3, X+RDO 0.5, X+SDO 0.6, X+CSDO 0.2, and M+CSDO 0.3) are statistically better than the best control setup (X 1.0). The advantage of other setups over the control ones are not statistically significant in this comparison.

It is worth to notice that comparing the X 1.0 with the rest setups using Holm's post-hoc procedure, the first 50 setups (up to X+CSDO 0.7 inclusively) from the ranking presented in Table 6.1 are statistically better than the best control experiments X 1.0.

6.4.2. Quantitative Results

Due to the large number of setups tested on 39 problems it is senseless to present here all computed statistics for each pair of setup and problem. Therefore, most of the computed results are placed in Appendix A, where an interested reader can find much more detailed results.

Tables 6.10–6.12 show detailed comparison of two setups selected from the success rate point of view: the best one (M+RDO 0.3) and the best control setup (X 1.0). For each problem these tables present:

- achieved success rate and, if appropriate, mean number of generation when the ideal solution was found,
- median of errors and number of hits obtained on training set by the best of run individual,
- median of errors and number of hits obtained on testing set (only Table 6.10 with symbolic regression problems) by the best of run individual,
- average number of milliseconds required for single evolutionary run,
- number of achieved successes in a time unit (scaled appropriately to get number of successes per hour),
- mean size (i.e., number of nodes) of individuals over all generations,
- average size of the best of run individual in case of a successful run (size of an ideal solution of the problem) and in case of failure (the best but imperfect individual).

Problem	*Setup*	*Success rate*	*Success gen.*	*Train error*	*Train hits*	*Test error*	*Test hits*
	M+RDO 0.3	1.000	1.8	0.000	20.0	0.000	200.0
F01	X 1.0	0.955	10.0	0.000	20.0	0.000	200.0
	best value:	*1.000*	*1.0*	*0.000*	*20.0*	*0.000*	*200.0*
	M+RDO 0.3	0.840	5.1	0.000	20.0	0.000	200.0
F02	X 1.0	0.725	16.1	0.000	20.0	0.000	200.0
	best value:	*0.965*	*1.4*	*0.000*	*20.0*	*0.000*	*200.0*
	M+RDO 0.3	0.335	8.5	0.002	20.0	0.083	189.0
F03	X 1.0	0.680	19.0	0.000	20.0	0.000	200.0
	best value:	*0.680*	*2.0*	*0.000*	*20.0*	*0.000*	*200.0*
	M+RDO 0.3	0.215	10.3	0.002	20.0	0.161	182.5
F04	X 1.0	0.270	30.5	0.174	6.0	1.658	64.0
	best value:	*0.580*	*2.0*	*0.000*	*20.0*	*0.000*	*200.0*
	M+RDO 0.3	0.580	8.1	0.000	20.0	0.000	200.0
F05	X 1.0	0.015	18.3	0.096	4.0	0.910	45.0
	best value:	*0.750*	*1.8*	*0.000*	*20.0*	*0.000*	*200.0*
	M+RDO 0.3	0.695	4.6	0.000	20.0	0.000	200.0
F06	X 1.0	0.460	24.2	0.054	10.0	0.466	102.0
	best value:	*0.945*	*1.0*	*0.000*	*20.0*	*0.000*	*200.0*
	M+RDO 0.3	0.315	7.8	0.001	20.0	0.021	199.0
F07	X 1.0	0.070	20.4	0.049	7.0	0.490	68.0
	best value:	*0.430*	*1.0*	*0.000*	*20.0*	*0.012*	*200.0*
	M+RDO 0.3	0.535	6.9	0.000	20.0	0.000	200.0
F08	X 1.0	0.000	—	0.216	2.0	2.712	18.0
	best value:	*0.535*	*1.0*	*0.000*	*20.0*	*0.000*	*200.0*
	M+RDO 0.3	0.990	3.0	0.000	100.0	0.000	10000.0
F09	X 1.0	0.250	32.2	1.967	10.0	160.697	852.0
	best value:	*1.000*	*1.0*	*0.000*	*100.0*	*0.000*	*10000.0*
	M+RDO 0.3	0.955	3.5	0.000	100.0	0.000	10000.0
F10	X 1.0	0.190	24.6	0.894	20.0	79.067	1831.5
	best value:	*1.000*	*1.2*	*0.000*	*100.0*	*0.000*	*10000.0*
	M+RDO 0.3	1.000	1.5	0.000	100.0	0.000	10000.0
F11	X 1.0	0.170	13.4	2.578	6.0	191.581	559.0
	best value:	*1.000*	*1.1*	*0.000*	*100.0*	*0.000*	*10000.0*
	M+RDO 0.3	0.000	—	0.194	42.0	19.208	4118.5
F12	X 1.0	0.005	92.0	1.890	6.0	164.741	528.5
	best value:	*0.005*	*8.0*	*0.141*	*56.0*	*15.206*	*5383.0*
	M+RDO 0.3	0.310	12.7	0.000	20.0	0.026	198.0
P1	X 1.0	0.000	—	0.127	4.0	1.762	24.0
	best value:	*0.370*	*2.0*	*0.000*	*20.0*	*0.007*	*200.0*
	M+RDO 0.3	0.010	8.5	0.008	18.0	0.675	149.5
P2	X 1.0	0.000	—	0.878	1.0	8.325	10.0
	best value:	*0.075*	*4.0*	*0.002*	*20.0*	*0.252*	*169.5*
	M+RDO 0.3	0.020	12.8	0.007	18.0	1.470	146.5
P3	X 1.0	0.065	56.0	0.390	4.0	3.589	44.0
	best value:	*0.095*	*3.0*	*0.003*	*20.0*	*0.491*	*157.5*
	M+RDO 0.3	0.420	13.3	0.000	20.0	0.006	200.0
R0	X 1.0	0.005	11.0	0.144	4.0	1.398	34.5
	best value:	*0.675*	*1.1*	*0.000*	*20.0*	*0.000*	*200.0*
	M+RDO 0.3	0.010	8.0	0.011	17.0	0.357	149.0
R1	X 1.0	0.000	—	0.703	1.0	6.877	7.0
	best value:	*0.035*	*2.0*	*0.004*	*19.0*	*0.166*	*173.0*
	M+RDO 0.3	0.000	—	0.008	18.0	0.452	149.0
R2	X 1.0	0.000	—	0.754	1.0	7.552	7.0
	best value:	*0.005*	*21.0*	*0.003*	*20.0*	*0.187*	*172.0*
	M+RDO 0.3	0.005	13.0	0.002	20.0	0.135	183.0
R3	X 1.0	0.000	—	0.148	5.0	1.529	52.0
	best value:	*0.015*	*5.0*	*0.001*	*20.0*	*0.086*	*189.0*

Table 6.10.: Success rates, and median errors and hits comparison of the best setup (M+RDO 0.3) with the best control setup (X 1.0) — results for 19 symbolic regression problems. The shown best values (in italic) for each problem and each column may come from different setups.

Problem	*Setup*	*Time [ms]*	*Success per hour*	*Mean size*	*Size (success)*	*Size (failed)*
F01	M+RDO 0.3	21.9	164492.3	12.9	11.7	—
	X 1.0	22.3	154433.2	18.7	12.6	59.1
	best value:	*15.1*	*237723.3*	*9.9*	*11.4*	*19.6*
F02	M+RDO 0.3	782.7	3863.8	118.4	16.3	266.6
	X 1.0	102.2	25546.5	46.8	18.6	82.1
	best value:	*102.2*	*25546.5*	*17.5*	*15.3*	*18.4*
F03	M+RDO 0.3	2880.2	418.7	165.2	25.3	315.9
	X 1.0	101.9	24014.6	42.4	24.1	75.2
	best value:	*101.9*	*24014.6*	*18.2*	*19.9*	*16.8*
F04	M+RDO 0.3	2915.7	265.5	144.9	25.5	276.4
	X 1.0	216.4	4492.3	54.2	34.5	82.4
	best value:	*216.4*	*4492.3*	*20.2*	*21.9*	*20.8*
F05	M+RDO 0.3	1825.2	1144.0	130.7	15.1	271.5
	X 1.0	278.6	193.8	56.1	14.7	77.8
	best value:	*278.6*	*1384.7*	*16.3*	*9.0*	*16.8*
F06	M+RDO 0.3	1286.0	1945.6	151.9	13.0	321.3
	X 1.0	122.2	13547.7	29.9	14.2	51.9
	best value:	*122.2*	*19369.3*	*17.4*	*11.6*	*18.2*
F07	M+RDO 0.3	2696.1	420.6	129.6	16.4	244.7
	X 1.0	217.8	1157.0	40.9	17.7	63.3
	best value:	*217.8*	*1387.1*	*16.8*	*13.8*	*17.4*
F08	M+RDO 0.3	1951.9	986.7	131.0	11.8	267.9
	X 1.0	287.4	0.0	56.1	—	81.0
	best value:	*287.4*	*1390.0*	*19.0*	*11.8*	*19.3*
F09	M+RDO 0.3	223.0	15979.6	36.3	8.3	164.0
	X 1.0	409.9	2195.9	32.3	7.5	53.6
	best value:	*67.7*	*53207.3*	*8.9*	*7.5*	*8.9*
F10	M+RDO 0.3	650.5	5285.1	98.2	10.7	355.1
	X 1.0	422.2	1620.0	33.6	11.9	47.7
	best value:	*121.8*	*29558.1*	*10.2*	*9.3*	*10.9*
F11	M+RDO 0.3	58.3	61801.9	11.6	6.0	—
	X 1.0	509.6	1200.9	34.1	8.3	51.8
	best value:	*47.4*	*75960.7*	*9.7*	*5.9*	*6.0*
F12	M+RDO 0.3	10820.9	0.0	128.0	—	234.7
	X 1.0	582.4	30.9	56.0	97.0	83.5
	best value:	*582.4*	*30.9*	*15.2*	*42.0*	*15.4*
P1	M+RDO 0.3	2756.3	404.9	114.7	26.8	200.2
	X 1.0	324.4	0.0	69.3	—	100.8
	best value:	*324.4*	*499.6*	*22.2*	*15.0*	*22.6*
P2	M+RDO 0.3	4669.8	7.7	168.1	29.0	312.7
	X 1.0	330.7	0.0	70.6	—	101.0
	best value:	*330.7*	*145.1*	*26.8*	*27.0*	*27.5*
P3	M+RDO 0.3	4672.5	15.4	166.4	37.5	305.6
	X 1.0	248.3	942.4	58.3	46.8	86.1
	best value:	*248.3*	*942.4*	*19.9*	*29.0*	*20.4*
R0	M+RDO 0.3	2492.7	606.6	132.3	19.8	261.1
	X 1.0	263.6	68.3	49.3	55.0	67.8
	best value:	*263.6*	*1483.5*	*17.4*	*12.9*	*18.0*
R1	M+RDO 0.3	4041.8	8.9	132.1	20.0	227.1
	X 1.0	305.3	0.0	64.8	—	93.8
	best value:	*305.3*	*16.7*	*25.4*	*19.0*	*25.9*
R2	M+RDO 0.3	4723.9	0.0	160.0	—	300.0
	X 1.0	262.8	0.0	46.2	—	66.9
	best value:	*262.8*	*4.7*	*23.1*	*91.0*	*20.9*
R3	M+RDO 0.3	4307.6	4.2	144.5	31.0	240.7
	X 1.0	306.3	0.0	62.8	—	94.4
	best value:	*306.3*	*24.0*	*15.2*	*25.0*	*15.9*

Table 6.11.: Time execution and mean individual's size comparison of the best setup (M+RDO 0.3) with the best control setup (X 1.0) — results for 19 symbolic regression problems. The shown best values (in italic) for each problem and each column may come from different setups.

Problem	*Setup*	*Success rate*	*Success gen.*	*Train error*	*Train hits*
ICMP6	M+RDO 0.3	1.000	6.7	0.000	64.0
	X 1.0	0.185	75.8	2.000	62.0
	best value:	*1.000*	*4.6*	*0.000*	*64.0*
ICMP8	M+RDO 0.3	1.000	16.8	0.000	256.0
	X 1.0	0.000	—	13.500	242.5
	best value:	*1.000*	*11.4*	*0.000*	*256.0*
IMAJ5	M+RDO 0.3	1.000	3.6	0.000	32.0
	X 1.0	0.985	24.2	0.000	32.0
	best value:	*1.000*	*2.1*	*0.000*	*32.0*
IMAJ6	M+RDO 0.3	1.000	6.2	0.000	64.0
	X 1.0	0.735	56.9	0.000	64.0
	best value:	*1.000*	*3.8*	*0.000*	*64.0*
IMAJ7	M+RDO 0.3	1.000	11.9	0.000	128.0
	X 1.0	0.050	85.2	3.000	125.0
	best value:	*1.000*	*5.8*	*0.000*	*128.0*
IMUX11	M+RDO 0.3	1.000	10.2	0.000	2048.0
	X 1.0	0.010	89.5	231.000	1817.0
	best value:	*1.000*	*5.0*	*0.000*	*2048.0*
IMUX6	M+RDO 0.3	1.000	3.1	0.000	64.0
	X 1.0	0.955	27.8	0.000	64.0
	best value:	*1.000*	*2.0*	*0.000*	*64.0*
IPAR4	M+RDO 0.3	1.000	4.0	0.000	16.0
	X 1.0	0.890	34.6	0.000	16.0
	best value:	*1.000*	*2.1*	*0.000*	*16.0*
IPAR5	M+RDO 0.3	1.000	9.9	0.000	32.0
	X 1.0	0.170	75.6	2.000	30.0
	best value:	*1.000*	*4.1*	*0.000*	*32.0*
IPAR6	M+RDO 0.3	0.975	25.9	0.000	64.0
	X 1.0	0.005	93.0	11.000	53.0
	best value:	*1.000*	*7.3*	*0.000*	*64.0*
NCMP6	M+RDO 0.3	1.000	6.8	0.000	64.0
	X 1.0	0.100	71.1	2.000	62.0
	best value:	*1.000*	*4.7*	*0.000*	*64.0*
NCMP8	M+RDO 0.3	1.000	15.5	0.000	256.0
	X 1.0	0.000	—	14.000	242.0
	best value:	*1.000*	*11.0*	*0.000*	*256.0*
NMAJ5	M+RDO 0.3	1.000	3.9	0.000	32.0
	X 1.0	0.870	33.9	0.000	32.0
	best value:	*1.000*	*2.4*	*0.000*	*32.0*
NMAJ6	M+RDO 0.3	1.000	6.3	0.000	64.0
	X 1.0	0.360	69.1	1.000	63.0
	best value:	*1.000*	*3.7*	*0.000*	*64.0*
NMAJ7	M+RDO 0.3	1.000	11.6	0.000	128.0
	X 1.0	0.000	—	5.000	123.0
	best value:	*1.000*	*5.5*	*0.000*	*128.0*
NMUX11	M+RDO 0.3	1.000	14.9	0.000	2048.0
	X 1.0	0.000	—	334.500	1713.5
	best value:	*1.000*	*8.6*	*0.000*	*2048.0*
NMUX6	M+RDO 0.3	1.000	4.5	0.000	64.0
	X 1.0	0.730	56.5	0.000	64.0
	best value:	*1.000*	*2.9*	*0.000*	*64.0*
NPAR4	M+RDO 0.3	1.000	5.5	0.000	16.0
	X 1.0	0.365	55.0	1.000	15.0
	best value:	*1.000*	*2.8*	*0.000*	*16.0*
NPAR5	M+RDO 0.3	1.000	15.4	0.000	32.0
	X 1.0	0.005	82.0	6.000	26.0
	best value:	*1.000*	*5.3*	*0.000*	*32.0*
NPAR6	M+RDO 0.3	0.870	43.7	0.000	64.0
	X 1.0	0.000	—	18.000	46.0
	best value:	*1.000*	*10.1*	*0.000*	*64.0*

Table 6.12.: Success rates, and median errors and hits comparison of the best setup (M+RDO 0.3) with the best control setup (X 1.0) — results for 20 Boolean problems. The shown best values (in italic) for each problem and each column may come from different setups.

Problem	Setup	Time [ms]	Success per hour	Mean size	Size (success)	Size (failed)
ICMP6	M+RDO 0.3	94.9	37919.9	32.2	115.9	—
	X 1.0	669.2	995.2	172.4	290.7	206.6
	best value:	*78.2*	*46032.6*	*8.1*	*31.0*	*8.6*
ICMP8	M+RDO 0.3	688.2	5231.2	99.9	322.4	—
	X 1.0	588.4	0.0	158.7	—	230.9
	best value:	*96.5*	*6338.7*	*8.6*	*75.0*	*8.7*
IMAJ5	M+RDO 0.3	35.3	101927.5	26.3	66.8	—
	X 1.0	115.9	30605.4	107.2	205.1	152.0
	best value:	*21.2*	*169925.8*	*21.4*	*50.8*	*28.7*
IMAJ6	M+RDO 0.3	102.5	35112.5	42.7	142.9	—
	X 1.0	583.3	4536.3	223.1	350.6	313.9
	best value:	*93.6*	*38451.0*	*22.6*	*71.5*	*27.1*
IMAJ7	M+RDO 0.3	472.3	7622.3	126.9	415.3	—
	X 1.0	1072.3	167.9	302.5	528.9	461.4
	best value:	*204.9*	*9689.4*	*26.0*	*218.5*	*27.6*
IMUX11	M+RDO 0.3	691.7	5204.5	48.8	93.0	—
	X 1.0	625.2	57.6	177.5	179.5	244.9
	best value:	*143.8*	*7726.8*	*10.7*	*56.0*	*12.7*
IMUX6	M+RDO 0.3	35.6	101018.4	19.3	18.4	—
	X 1.0	155.0	22176.4	111.6	129.7	193.9
	best value:	*25.5*	*141343.8*	*8.5*	*12.4*	*7.8*
IPAR4	M+RDO 0.3	41.0	87852.7	38.2	81.9	—
	X 1.0	342.4	9357.4	204.5	269.3	210.1
	best value:	*28.8*	*124963.5*	*34.1*	*42.5*	*27.0*
IPAR5	M+RDO 0.3	260.9	13798.2	101.6	255.5	—
	X 1.0	1110.0	551.3	319.1	440.5	436.8
	best value:	*184.4*	*19525.7*	*47.2*	*36.0*	*58.9*
IPAR6	M+RDO 0.3	2242.7	1565.1	426.1	806.1	751.8
	X 1.0	1096.6	16.4	349.7	220.0	500.7
	best value:	*352.3*	*3424.9*	*53.2*	*32.0*	*59.8*
NCMP6	M+RDO 0.3	79.6	45239.6	27.9	101.2	—
	X 1.0	553.4	650.5	143.8	235.1	186.4
	best value:	*68.3*	*52717.4*	*8.0*	*51.7*	*10.5*
NCMP8	M+RDO 0.3	531.7	6770.9	79.5	257.9	—
	X 1.0	431.5	0.0	134.9	—	203.0
	best value:	*85.5*	*8428.4*	*7.7*	*151.0*	*8.7*
NMAJ5	M+RDO 0.3	29.4	122553.5	23.1	61.6	—
	X 1.0	206.1	15195.5	122.3	179.1	133.9
	best value:	*24.1*	*149229.9*	*15.5*	*40.6*	*21.0*
NMAJ6	M+RDO 0.3	103.8	34677.7	34.9	115.1	—
	X 1.0	608.9	2128.4	179.2	279.1	230.0
	best value:	*87.2*	*41294.6*	*18.7*	*63.0*	*21.6*
NMAJ7	M+RDO 0.3	420.7	8556.7	95.3	323.9	—
	X 1.0	768.2	0.0	204.7	—	308.2
	best value:	*174.0*	*12486.8*	*20.0*	*151.0*	*23.6*
NMUX11	M+RDO 0.3	1437.9	2503.6	104.2	216.4	—
	X 1.0	615.1	0.0	168.3	—	253.4
	best value:	*117.4*	*3147.2*	*14.2*	*141.0*	*16.2*
NMUX6	M+RDO 0.3	47.9	75164.4	23.6	46.8	—
	X 1.0	382.8	6865.2	139.1	191.1	173.6
	best value:	*44.6*	*80759.1*	*11.4*	*28.0*	*14.9*
NPAR4	M+RDO 0.3	50.8	70907.4	34.1	83.8	—
	X 1.0	557.5	2356.8	188.4	247.4	212.2
	best value:	*41.7*	*86394.4*	*22.5*	*54.0*	*23.0*
NPAR5	M+RDO 0.3	462.5	7784.4	132.6	336.3	—
	X 1.0	772.0	23.3	219.9	307.0	286.5
	best value:	*205.3*	*17037.0*	*28.0*	*123.0*	*38.0*
NPAR6	M+RDO 0.3	3827.3	818.3	448.7	669.2	626.4
	X 1.0	682.5	0.0	226.8	—	324.4
	best value:	*224.8*	*3153.9*	*31.9*	*173.0*	*39.5*

Table 6.13.: Time execution and mean individual's size comparison of the best setup (M+RDO 0.3) with the best control setup (X 1.0) — results for 20 Boolean problems. The shown best values (in italic) for each problem and each column may come from different setups.

In addition, for each problem and each property, the best value from all 106 setups is shown to get better image of achieved quality and be able to compare presented methods against others.

The first observation is that M+RDO 0.3 has worse success rate than X 1.0 only on four (F03, F04, F12, P3) out of all 39 problems. For the symbolic regression problems, the setup with a desired operator has a big advantage over the control experiment on the rest of problems except P2, R1, R2, and R3 where both setups have no or hardly any successes. In fact, no setups achieved better success ratio than 0.5% on F12, 7.5% on P2, 3.5% on R1, 0.5% on R2, and 1.5% on R3. On the easy F01 problem both M+RDO 0.3 and X 1.0 setups perform successfully. For the Boolean problems the M+RDO 0.3 found an ideal solution almost always (except problems IPAR6 and NPAR6), whereas X 1.0 often was not able to found any single solution in 200 repetition (for ICMP8, NCMP8, NMAJ7, NMUX11, and NPAR6). For 9 problems X 1.0 found an ideal in at most one tenth tries.

There could be many reasons why X 1.0 has advantage over M+RDO 0.3 setup on the F03, F04, and P3 problems (on F12 it found ideal only once out of 200 tries). One of the explanation, which seems most likely, is that these three problems require quite big programs with many repeated subfunctions. F03, F04, and P3 are polynomials in the form $x^n + x^{n-1} + \cdots + x$ (see Chapter 4.2) which could be quite easily created by a standard crossover operator by simply copying frequent subexpressions. This property is intensified by the fact that only addition and multiplication operators are needed which dramatically increase the probability of constructing proper expressions, for instance, equal to x^n.

Comparing mean generation number in which an ideal solution was found, it appears that M+RDO 0.3 setup needs much less generations to succeed — often even one order of magnitude less than X 1.0. The one exception is the problem R0 where X 1.0 found an ideal solution in 11th generation, however this setup did it only once (for 200 independent runs) comparing to M+RDO 0.3 which found 84 ideal solutions (in 13.3th generation in average).

It appears that X 1.0 got better median error on the F03 problem. However, for all others the M+RDO 0.3 setup is better (or equally good for F01 and F02) comparing median errors and hits on both train and test sets. This observation shows that the desired operator does not only improve the probability of success but also reduces errors of the best individuals in most evolutionary runs.

The mean time of evolutionary runs shown in Tables 6.11 and 6.13 are inaccurate due to the fact that the computations run simultaneously on several very similar computers on which other processes may use the processor time and other system resources as well. However, the measured and averaged times are still informative and give some sense of performance.

In general, the time required by an evolution depends on two main factors:

1. Problem hardness — if an ideal solution is not found earlier, the evolution will lasts all 100 generations.

2. Problem properties — sometimes the evolution tends to build larger individuals which require more time to process, but sometimes smaller programs are promoted.

Due to the need of scanning the whole library of semantically unique subfunctions (subtrees) in the RDO operator, M+RDO 0.3 setup requires generally much more time to execute than X 1.0. It is especially visible for problems which are solved by M+RDO 0.3 but not by X 1.0 (in consequence requiring all 100 generations) for which M+RDO 0.3 setup runs much longer. For example, NUMX11 problem is solved by M+RDO 0.3 over two times longer than X 1.0 setup, despite the fact that M+RDO 0.3 evolution stops finding an ideal after 15th generation on average, and X 1.0 always lasts all 100 generations. On the other hand, a possible fewer number of generations caused that 14 problems from the Boolean domain and only 3 from the symbolic regression domain are solved quicker than in the X 1.0 setup.

To calculate the performance of each setup, the average number of successes per hour is calculated. This measure says how many successful runs is expected if a given setup would run for one hour (starting a new evolution after finishing last one). The computed values shows that in 11 symbolic regression problems and in all Boolean problems the M+RDO 0.3 setup outperforms X 1.0. It is worth to notice that if the used linear search procedure will be replaced with some faster method (e.g., the rough search which will be considered in Section 6.5), there is a chance that RDO operator would have even more advantage. However, in the current implementation, M+RDO 0.3 setup has even 1–2 orders of magnitude worse performance in solving some symbolic regression problems.

A code bloat is a process of an inexorably growing of a mean program size during an evolution. To investigate this effect in both compared setups the mean size of individuals in all generations averaged over 200 runs are calculated and shown in Tables 6.11 and 6.13. It reveals that the approach with RDO operator promotes bigger programs trying to find an ideal solution. This is especially visible when comparing mean size of the best programs in runs which failed in finding the optimum. In M+RDO 0.6 setup, such programs have much more nodes (even 7 times!) than the best but not perfect solutions found by X 1.0 setup. This clearly shows that RDO operator tends to explore a bigger and bigger space when it has trouble with finding a solution.

On the other hand, the tables show that the found, real solutions are often much smaller and more compact in the M+RDO 0.3 setup. For example, the ideal solutions for problem IMUX6 have 18.4 nodes in average if they come from M+RDO 0.3 setup but more than 129 nodes from the X 1.0 setup. This difference is important because both setups are quite successful in solving IMUX6 and the sizes are averaged over large number of solutions (M+RDO 0.3 always found an ideal and X 1.0 found 191 ideals in 200 runs).

It seems that the desired operators are trying to construct as simple solution as possible if they could find such ideal. In other case RDO builds bigger and more complicated programs in the hope that such complexity is required by the solved problem. This is not always true, like in case of problem P3 or F03 when the found ideal programs are not big. However, it seems that this strategy gives in general good results and it enables the search procedure to find a solution eventually. It is worth to notice, that a bigger population potentially generates a bigger library of subtrees used by RDO. Probably, this may be one reason why RDO succeed. However, problems in Boolean domain do not corroborate this hypothesis as the mean program size is mostly (except for problem IMUX6) much smaller than the average size of found ideal solutions. Nevertheless, the growing size of individuals (implying more nodes to process and more subtrees in a library) is an additional explanation why evolution with RDO operator needs more time than the control setup.

Figure 6.4.2 shows number of unique semantics (solid line) and unique fitness values (dotted line) in each generation for R0, F04, NPAR4, and NMUX11 problems. The numbers are averaged from 200 independent runs. However, because some setups are quite successful on some problems, each vertical line shows the generations where some runs terminates and therefore further values must be averaged from smaller sample of runs.

In fact, the number of unique fitness values should never be greater than the number of semantics, but such results come from the used method of counting. Semantics are compared with an epsilon threshold (see Definition 3 on page 34), and therefore if the difference between elements of semantics are less than $1.11E-15$ then both are treated as identical. In contrast, fitness values are compared exactly with full available floating point precision and therefore two fitness values which differ even much less than $1.11E-15$ would be treated as two distinct values.

The presented charts shows that the diversity of a population quickly decrease in the first few generation and then it increases as fast as it drops previously. This observation is common for each setup and each problem — every setup for every problem has a similar 'hole' in the first generations. Whats happen latter depends, however, both on a method and a solved problem. For example, in case of R0 problem, the graph clearly shows that M+RDO 0.3 maintains much higher diversity through all generation than X 1.0. It appears, that this observation is true also for many other problems (not shown in this figure too) — even for F03, F04, and P3 problems where M+RDO 0.3 is worse than X 1.0 in the sense of the acquired success ratio.

Other observation is that in symbolic regression problems the number of unique semantics is roughly equal to the number of unique fitness values. For the Boolean problem this is not true because possible number of fitness values is much smaller (2^{bits}) than the number of possible distinct semantics ($2^{2^{bits}}$). For most Boolean problems, M+RDO 0.3 setup achieves the highest fitness diversity in the first ten generations. In contrast, X

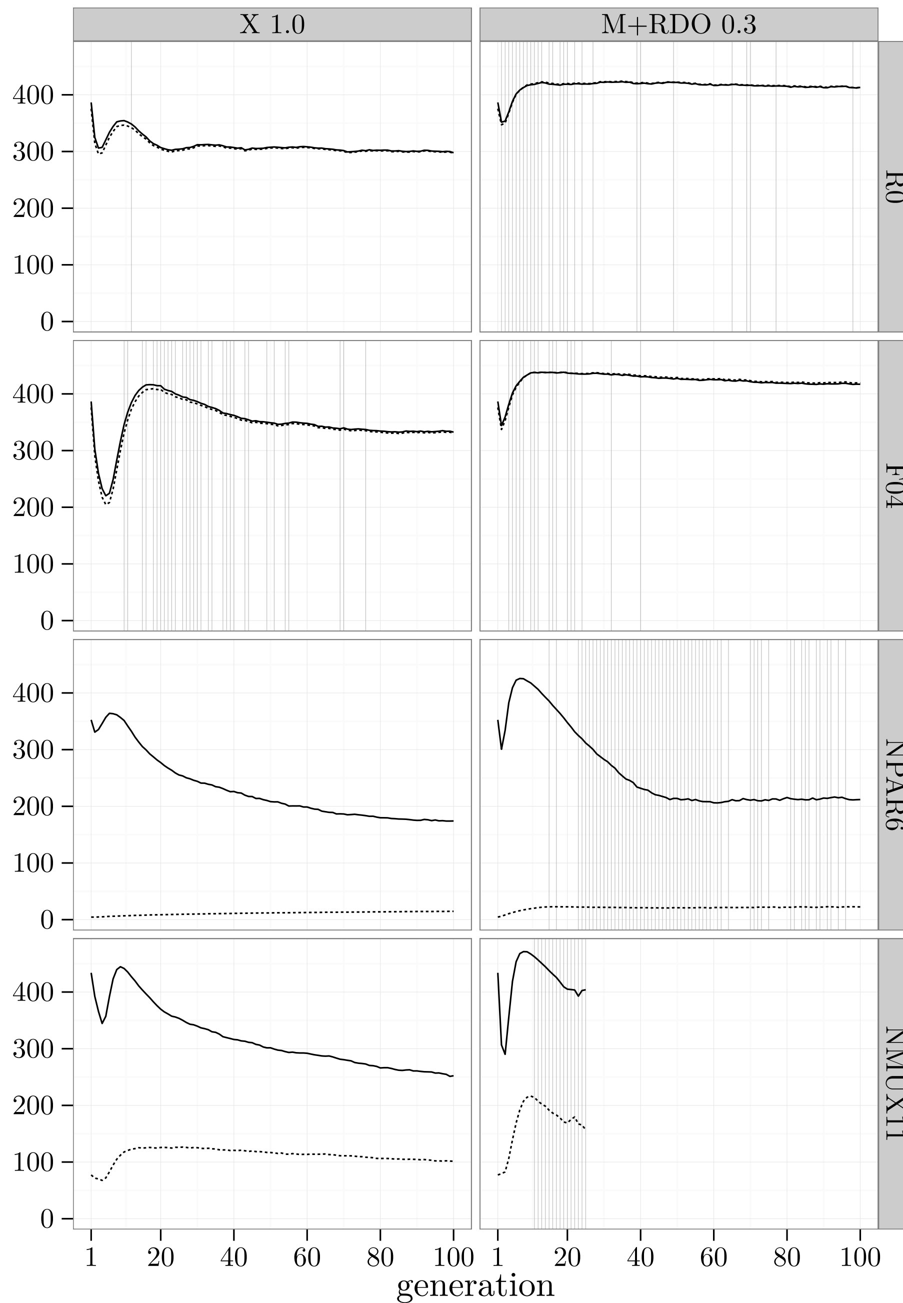


Figure 6.4.2.: Number of individuals with unique semantics (solid line) and unique fitness values (dotted line). Possible greater number of fitness values than semantics come out of floating number comparison (details in text). Vertical lines show generations in which some runs finished with an ideal solution found, therefore values in letter generation are averaged from fewer number of samples.

1.0 sometimes increases this fitness diversity continuously through all generations like in problem NPAR6.

6.5. Discussion and Conclusions

This chapter described a novel approach which uses a property of instruction inversion to calculate desired semantics for any part of a program and then searching a desired building block fitting in that place. The proposed family of methods, exploiting this property of instructions, are trying to build a final solution in a more smart way than simple, 'blind' evolution. This is the main difference between desired building blocks approach comparing to others which construct programs in a purely syntactically manner disregarding the semantically analysis of the product.

We can look at the desired operators also from another angle — they try to narrow the search space explored by the evolution. The approach based on desired semantics may be seen as an attempt to comply with some hard constraints imposed (implicitly) by the problem definition and to avoid wasting resources for testing solutions (programs) that evidently violate them. For instance, in the proposed approach we have considered as such constraints the requirement that every part enables the entire program to be an ideal solution. However, in general the constraints can be defined freely — the goal is to narrow the search space by giving some constraints which should never be violated by any acceptable solution. Because the evolution has no idea how to cope with these constraints (if it knew, then the ideal solution would appear instantly), then in each step it tries to find a subprogram which violates the constraints as little as possible, hopping that this comes to an end. The only difference between all proposed methods (RDO, SDO, CSDO, EDO, and CEDO) lies in the choice of the subprogram which will be optimized in respect to the constraint.

As we have seen, all operators proposed in this chapter optimize locally a parent individual. Therefore it seems as a kind of memetic algorithms [94], however the local optimization does not optimize fitness directly, but rather the adjustment of a subprogram (subtree).

It is worth to notice, that even if the instructions are black boxes (so we have no idea how to invert them), it is theoretically possible to estimate the desired semantics through local optimization. If the elements of used semantics are independent (this is the case in sampling semantics), it is possible (e.g., using gradient methods) to find local optima for each element by analyzing the differences between final context behavior and the target semantics. However such optimization seems computationally much too expensive and thus the net effect of employing desired semantics would be probably negligible, if any.

Presented results clearly shows that the new proposed methods, especially setups like M+RDO 0.3, have a big, statistical significant advantage over vanilla GP on a big set of

different problems with varying difficulty. This is especially visible for problems from the Boolean domains where M+RDO 0.3 was able to find almost always an ideal solution even for problems for which X 1.0 never succeed.

This observation suggests that the proposed approach is extremely efficient for discrete domains. For continuous domains, like in symbolic regression problems, further research is necessary to improve the performance of seeking appropriate building blocks and, in consequence, shorten the overall computational time. However, it is important to notice that the current implementation is still very attractive for most tested symbolic regression problems for which the performance measured in the number of successes obtained in a unit of time is higher than for X 1.0 setup. Additionally, the median error committed by M+RDO 0.3 is smaller than those by X 1.0 (the best control setup).

To verify the hypothesize of beneficial influence of semantic operators, we have performed additional experiments combining the two best desired operators (RDO and SDO) described in this chapter with semantic crossover (SASES — see Section 5.3) and semantic mutation (SSM — see Section 5.4), instead of previously used standard crossover (X) or standard mutation (M). As before, we tested pairs of these operators with different proportions. Also, for each combination of them we use either standard population initialization or the proposed semantic initialization (SEM — see Section 5.2). We compare these setups with the previously presented: both in this chapter and in the preceding chapter (see Section 5.5) — therefore, there are 267 different setups in total.

Tables 6.14 and 6.15 shows Friedman ranks of success ratio and median error, respectively. Again, to make these tables more compact, some part of worse setups from each pair of breeding operators are skipped. For the complete rankings see Appendix A.

As it is easily noticed, the best setups involves either RDO or SDO operator (to be specific — the best 135 setups in the ranking of success ratio, and the best 125 setups in the ranking of errors). The best success ratio gives setups with RDO 0.3–0.6 and SDO 0.3–0.5 (the best setup with SDO has tenth position in the ranking). The smallest error is committed by setups with RDO, especially with RDO 0.5–0.7. The best setup with SDO (M+SDO 0.3) in this ranking (Table 6.15) is on the 63th position. These general observations corroborates remarks presented earlier in Section 6.4.1, and shows the stable recommendations for best setups (e.g., the range of the most profitable probabilities of our desired semantics).

Moreover, the first five setups from both presented rankings (Tables 6.14 and 6.15) use semantic population initialization (setups prefixed with 'SEM|'). This may have significant importance to the performance of our desired operators, as in the experiments we construct the library of subprograms (subtrees) from a current population (see Section 6.2). Therefore, the more diverse the population the more chance that the library contains useful elements.

The best setups also use SSM instead of standard mutation. SASES is used as well,

<table>
<tr><th>Setup</th><th>Rank</th><th>Setup</th><th>Rank</th><th>Setup</th><th>Rank</th><th>Setup</th><th>Rank</th></tr>
<tr><td>SEM|SSM+RDO 0.5</td><td>62.94</td><td>M+SDO 0.4</td><td>77.58</td><td>(skipped 3 items)</td><td></td><td>X+EDO 0.4</td><td>174.31</td></tr>
<tr><td>SEM|X+RDO 0.4</td><td>66.51</td><td>SEM|SSM+RDO 0.3</td><td>77.64</td><td>SEM|X+SDO 0.5</td><td>85.47</td><td>(skipped 2 items)</td><td></td></tr>
<tr><td>SEM|M+RDO 0.5</td><td>69.26</td><td>SSM+SDO 0.4</td><td>77.92</td><td>(skipped 2 items)</td><td></td><td>SASES+SSM 0.3</td><td>176.23</td></tr>
<tr><td>SEM|SASES+RDO 0.6</td><td>69.74</td><td>SASES+RDO 0.7</td><td>77.95</td><td>X+SDO 0.2</td><td>86.05</td><td>SEM|SASES+SSM 0.3</td><td>179.15</td></tr>
<tr><td>SEM|X+RDO 0.6</td><td>69.90</td><td>SEM|SSM+RDO 0.9</td><td>79.12</td><td>(skipped 5 items)</td><td></td><td>(skipped 7 items)</td><td></td></tr>
<tr><td>SASES+RDO 0.5</td><td>70.23</td><td>SEM|M+SDO 0.6</td><td>79.28</td><td>SASES+SDO 0.3</td><td>86.69</td><td>X 1.0</td><td>189.29</td></tr>
<tr><td>SSM+RDO 0.5</td><td>70.28</td><td>SEM|SSM+RDO 0.8</td><td>79.44</td><td>(skipped 20 items)</td><td></td><td>SEM|X 1.0</td><td>189.42</td></tr>
<tr><td>SEM|SASES+RDO 0.4</td><td>71.44</td><td>SSM+SDO 0.3</td><td>79.58</td><td>SASES+SDO 0.6</td><td>91.03</td><td>SASES+SSM 0.4</td><td>189.56</td></tr>
<tr><td>M+RDO 0.3</td><td>71.63</td><td>SASES+RDO 0.4</td><td>79.64</td><td>(skipped 24 items)</td><td></td><td>M+CEDO 0.2</td><td>189.72</td></tr>
<tr><td>SEM|SSM+SDO 0.3</td><td>71.67</td><td>SEM|M+SDO 0.2</td><td>79.86</td><td>SEM|RDO 1.0</td><td>102.51</td><td>SEM|SASES+SSM 0.4</td><td>190.71</td></tr>
<tr><td>SSM+RDO 0.4</td><td>71.85</td><td>M+RDO 0.7</td><td>79.97</td><td>(skipped 6 items)</td><td></td><td>M+CEDO 0.4</td><td>191.41</td></tr>
<tr><td>SEM|X+RDO 0.5</td><td>71.87</td><td>SEM|SASES+RDO 0.8</td><td>80.18</td><td>RDO 1.0</td><td>108.33</td><td>(skipped 3 items)</td><td></td></tr>
<tr><td>SEM|M+SDO 0.4</td><td>72.13</td><td>SEM|X+RDO 0.3</td><td>80.45</td><td>X+CSDO 0.2</td><td>108.77</td><td>X+SSM 0.1</td><td>193.18</td></tr>
<tr><td>SASES+RDO 0.6</td><td>72.26</td><td>M+RDO 0.2</td><td>81.10</td><td>(skipped 3 items)</td><td></td><td>SASES+M 0.5</td><td>194.10</td></tr>
<tr><td>SEM|M+RDO 0.4</td><td>73.33</td><td>SSM+SDO 0.2</td><td>81.50</td><td>X+CSDO 0.1</td><td>111.38</td><td>SEM|X+M 0.1</td><td>194.46</td></tr>
<tr><td>M+SDO 0.3</td><td>73.78</td><td>M+RDO 0.8</td><td>81.62</td><td>M+CSDO 0.3</td><td>111.45</td><td>X+M 0.1</td><td>194.50</td></tr>
<tr><td>SEM|SASES+RDO 0.5</td><td>74.08</td><td>SSM+RDO 0.7</td><td>81.73</td><td>(skipped 3 items)</td><td></td><td>X+M 0.2</td><td>195.15</td></tr>
<tr><td>SSM+RDO 0.6</td><td>74.19</td><td>M+SDO 0.6</td><td>81.79</td><td>M+CSDO 0.2</td><td>113.33</td><td>(skipped 2 items)</td><td></td></tr>
<tr><td>SEM|SSM+SDO 0.4</td><td>74.28</td><td>SEM|SASES+RDO 0.3</td><td>81.97</td><td>(skipped 15 items)</td><td></td><td>M+EDO 0.3</td><td>197.60</td></tr>
<tr><td>SEM|SASES+RDO 0.7</td><td>74.38</td><td>SEM|SASES+SDO 0.5</td><td>82.12</td><td>SEM|SDO 1.0</td><td>140.96</td><td>X+EDO 0.7</td><td>197.72</td></tr>
<tr><td>X+RDO 0.5</td><td>74.63</td><td>X+RDO 0.3</td><td>82.15</td><td>(skipped 2 items)</td><td></td><td>SEM|X+M 0.2</td><td>198.38</td></tr>
<tr><td>SEM|SSM+RDO 0.6</td><td>74.65</td><td>M+RDO 0.6</td><td>82.36</td><td>SDO 1.0</td><td>151.58</td><td>(skipped 2 items)</td><td></td></tr>
<tr><td>M+RDO 0.4</td><td>74.73</td><td>SEM|SSM+SDO 0.2</td><td>82.37</td><td>(skipped 2 items)</td><td></td><td>X+SSM 0.2</td><td>199.67</td></tr>
<tr><td>SEM|M+SDO 0.3</td><td>75.54</td><td>SEM|SASES+RDO 0.9</td><td>82.44</td><td>SASES+M 0.2</td><td>167.24</td><td>M+EDO 0.2</td><td>199.91</td></tr>
<tr><td>M+RDO 0.5</td><td>76.42</td><td>X+SDO 0.6</td><td>82.51</td><td>SEM|SASES+SSM 0.1</td><td>167.49</td><td>(skipped 11 items)</td><td></td></tr>
<tr><td>SEM|SSM+RDO 0.7</td><td>76.47</td><td>SEM|X+SDO 0.4</td><td>82.94</td><td>SEM|SASES 1.0</td><td>168.51</td><td>CSDO 1.0</td><td>207.58</td></tr>
<tr><td>SEM|M+SDO 0.5</td><td>76.65</td><td>(skipped 2 items)</td><td></td><td>X+CEDO 0.3</td><td>169.26</td><td>(skipped 37 items)</td><td></td></tr>
<tr><td>SEM|M+RDO 0.6</td><td>76.83</td><td>SEM|X+SDO 0.2</td><td>83.38</td><td>X+CEDO 0.1</td><td>169.68</td><td>CEDO 1.0</td><td>233.72</td></tr>
<tr><td>SEM|SSM+RDO 0.4</td><td>77.06</td><td>(skipped 3 items)</td><td></td><td>SASES 1.0</td><td>169.83</td><td>EDO 1.0</td><td>234.15</td></tr>
<tr><td>X+RDO 0.4</td><td>77.13</td><td>X+SDO 0.5</td><td>84.36</td><td>SASES+SSM 0.1</td><td>170.36</td><td>M 1.0</td><td>234.37</td></tr>
<tr><td>SEM|M+RDO 0.3</td><td>77.14</td><td>SSM+RDO 0.2</td><td>84.59</td><td>SASES+M 0.1</td><td>170.92</td><td>SEM|SSM 1.0</td><td>234.68</td></tr>
<tr><td>SEM|SSM+SDO 0.5</td><td>77.28</td><td>SEM|SASES+SDO 0.3</td><td>84.85</td><td>X+EDO 0.1</td><td>172.42</td><td>SSM 1.0</td><td>235.23</td></tr>
<tr><td>M+SDO 0.5</td><td>77.47</td><td>SEM|SASES+SDO 0.4</td><td>84.97</td><td>(skipped 2 items)</td><td></td><td>SEM|M 1.0</td><td>235.55</td></tr>
</table>

Table 6.14.: Friedman ranks of *success ratio* performance on all 39 problems (both symbolic regression and Boolean domain). Setups with the best proportion of used operators are emphasized with a bold font. The worse setups for each combination of operators may be skipped to safe space.

Setup	*Rank*	*Setup*	*Rank*	*Setup*	*Rank*	*Setup*	*Rank*
SEM|SSM+RDO 0.6	**68.28**	SSM+RDO 0.3	76.51	SEM|SASES+SDO 0.1	93.77	*(skipped 4 items)*	
SEM|SSM+RDO 0.5	69.41	M+RDO 0.3	76.55	SEM|X+SDO 0.3	93.87	X+EDO 0.2	164.79
SEM|M+RDO 0.6	**69.78**	SEM|X+RDO 0.5	76.56	M+SDO 0.4	93.95	X+CEDO 0.3	165.27
SEM|M+RDO 0.5	69.82	X+RDO 0.6	76.68	**SEM|RDO 1.0**	**94.01**	**SDO 1.0**	**165.50**
SEM|SSM+RDO 0.7	70.09	SEM|SSM+RDO 0.3	77.13	SEM|SASES+RDO 0.1	94.03	*(skipped 7 items)*	
SSM+RDO 0.7	**70.09**	SEM|SSM+RDO 0.9	77.60	SEM|M+SDO 0.4	94.13	**M+CEDO 0.3**	**181.12**
SSM+RDO 0.6	70.24	SASES+RDO 0.8	77.73	SSM+SDO 0.4	94.13	*(skipped 2 items)*	
M+RDO 0.6	**70.27**	SEM|M+RDO 0.3	77.87	*(skipped 11 items)*		M+CEDO 0.2	183.00
M+RDO 0.7	70.40	*(skipped 2 items)*		**RDO 1.0**	**95.99**	**SEM|X 1.0**	**183.32**
M+RDO 0.5	70.51	X+RDO 0.5	78.96	*(skipped 21 items)*		*(skipped 6 items)*	
SEM|M+RDO 0.7	70.60	*(skipped 19 items)*		**X+CSDO 0.1**	**108.04**	**X+SSM 0.1**	**186.72**
SSM+RDO 0.5	70.83	**M+SDO 0.3**	**88.42**	*(skipped 4 items)*		**X 1.0**	**187.18**
SEM|M+RDO 0.8	71.78	*(skipped 2 items)*		X+CSDO 0.2	110.35	SASES+M 0.7	187.26
SEM|SSM+RDO 0.4	71.90	**SASES+SDO 0.3**	**88.83**	SASES+SDO 0.7	112.12	**X+M 0.1**	**187.71**
SASES+RDO 0.7	**72.18**	**SSM+SDO 0.2**	**89.31**	**M+CSDO 0.3**	**115.38**	X+M 0.2	188.04
M+RDO 0.4	72.26	**SEM|SSM+SDO 0.2**	**89.79**	M+CSDO 0.2	116.60	**SEM|X+M 0.1**	**189.55**
M+RDO 0.8	72.32	SEM|SSM+SDO 0.3	89.90	*(skipped 25 items)*		M+CEDO 0.5	189.71
SEM|SASES+RDO 0.7	**72.36**	M+SDO 0.2	90.06	**SEM|SASES 1.0**	**145.33**	X+SSM 0.2	189.94
SEM|SASES+RDO 0.6	72.87	**SEM|M+SDO 0.3**	**90.08**	**SASES 1.0**	**146.79**	SEM|X+M 0.2	190.42
SEM|M+RDO 0.4	73.03	SEM|M+SDO 0.2	90.10	**SASES+M 0.1**	**147.21**	X+CEDO 0.5	190.91
SEM|SSM+RDO 0.8	73.17	**SEM|SASES+SDO 0.3**	**90.90**	X+CSDO 0.7	147.22	**M+EDO 0.3**	**191.19**
SSM+RDO 0.8	73.19	SASES+SDO 0.1	91.21	**SASES+SSM 0.1**	**147.82**	M+EDO 0.2	191.59
SEM|X+RDO 0.7	**73.49**	SASES+SDO 0.2	91.54	SASES+SSM 0.2	148.35	*(skipped 43 items)*	
SSM+RDO 0.4	73.54	**X+SDO 0.1**	**91.77**	SASES+M 0.2	148.88	**CSDO 1.0**	**233.59**
SEM|SASES+RDO 0.8	73.64	SEM|SASES+SDO 0.2	91.79	**SEM|SASES+SSM 0.1**	**148.95**	*(skipped 2 items)*	
X+RDO 0.7	**74.87**	SSM+SDO 0.3	91.92	**X+CEDO 0.1**	**149.78**	**SEM|SSM 1.0**	**238.01**
SASES+RDO 0.6	75.04	**SEM|X+SDO 0.2**	**92.23**	SEM|SASES+SSM 0.2	152.45	**SEM|M 1.0**	**238.58**
X+RDO 0.8	75.21	X+SDO 0.2	92.44	*(skipped 3 items)*		**M 1.0**	**239.14**
SASES+RDO 0.5	75.64	*(skipped 2 items)*		X+CEDO 0.2	157.18	**SSM 1.0**	**239.53**
SEM|X+RDO 0.6	76.14	SEM|SASES+SDO 0.4	93.13	SASES+M 0.3	157.54	*(skipped 2 items)*	
SEM|SASES+RDO 0.5	76.15	SEM|SSM+RDO 0.1	93.68	**X+EDO 0.1**	**157.88**	**EDO 1.0**	**256.78**
SEM|X+RDO 0.8	76.19	X+SDO 0.3	93.72	**SEM|SDO 1.0**	**158.76**	**CEDO 1.0**	**256.94**

Table 6.15.: Friedman ranks of *median error* on training set for all 39 problems (both symbolic regression and the Boolean domain). Setups with the best proportion of used operators are emphasized with a bold font. The worse setups for each combination of operators may be skipped to safe space.

however the results shows that SSM or standard mutation is more advantageous (especially in minimizing median error). Nevertheless, the good results obtained by setups combining many semantically based operators clearly demonstrate that such fusion is beneficial and worth to recommend.

Finally, in Table 6.16 we show a ranking of methods sorted by the achieved success per hour. It demonstrates that the setup SEM|X+RDO 0.4 has the highest mean computational performance. It seems that, averaging over all our benchmark problems, this setup requires the least time to found an ideal solution. More insight analysis shows that RDO operator appears in the most efficient configurations, irrespective of the problem domain. However, the best results comes from setups combining two semantic operators (SEM|X+RDO for symbolic regression domain or SASES+RDO for the Boolean domain).

As we have mentioned several times in this chapter (e.g., in Section 6.2), the brute-force search throughout the whole library for matching a given desired semantics is very time consuming. However, we can imagine various variants of a 'rough' search of nearest neighbor. Such procedure would consider only a small candidate list (a subset of all points) generated with a (possibly simple) heuristics. The rationale of such methods would be to speed up the computations with, if possible, limited deterioration of the overall results (i.e., the distance to the closest point). Considering such algorithms can be particularly justified when the main search is carried out by evolutionary algorithms, as they are stochastic by nature. Such a rough method applied to desired semantics could work as follow:

1. Select randomly one dimension (element) of the desired semantics with a known value v (i.e., omit all the unknown elements: insignificant and inconsistent).

2. Generate a candidate list of k subtrees from the library which have semantics most similar to v on the selected dimension.

3. Search linearly the candidate list, calculating the complete distance to the quested desired semantics, and return the closest semantics from the list.

For example, if parameter k equals to 10% of the whole size of the library, this procedure would lead to a ten times smaller number of subtrees (its semantics) which would have to be analyzed.

In this thesis we are mainly interested in analyzing the effectiveness of the proposed operators in general, and therefore we do not concentrate greatly on optimization issues. However, as the presented results shows, the brute-force search still allows the desired operators to be highly efficient. We hope, that applying some kind of 'rough' search could potentially improve this performance even more.

<table>
<tr><th>Setup</th><th>Rank</th><th>Setup</th><th>Rank</th><th>Setup</th><th>Rank</th><th>Setup</th><th>Rank</th></tr>
<tr><td>SEM|X+RDO 0.4</td><td>40.27</td><td>SSM+RDO 0.5</td><td>70.53</td><td>M+CSDO 0.9</td><td>120.74</td><td>X+EDO 0.3</td><td>157.95</td></tr>
<tr><td>SASES+RDO 0.5</td><td>45.05</td><td>SEM|SSM+RDO 0.5</td><td>71.40</td><td>SDO 1.0</td><td>120.97</td><td>X+M 0.1</td><td>158.10</td></tr>
<tr><td>X+RDO 0.4</td><td>45.18</td><td>SEM|M+RDO 0.6</td><td>72.01</td><td>(skipped 2 items)</td><td></td><td>SEM|X+M 0.1</td><td>158.33</td></tr>
<tr><td>SASES+RDO 0.6</td><td>48.45</td><td>(skipped 3 items)</td><td></td><td>SEM|X+SDO 0.8</td><td>123.21</td><td>X 1.0</td><td>158.50</td></tr>
<tr><td>X+RDO 0.5</td><td>48.49</td><td>SSM+RDO 0.9</td><td>73.69</td><td>(skipped 5 items)</td><td></td><td>(skipped 2 items)</td><td></td></tr>
<tr><td>SEM|SASES+RDO 0.4</td><td>48.71</td><td>(skipped 2 items)</td><td></td><td>SEM|M+SDO 0.8</td><td>125.37</td><td>SEM|X+M 0.2</td><td>159.15</td></tr>
<tr><td>M+RDO 0.3</td><td>49.01</td><td>SSM+RDO 0.4</td><td>77.26</td><td>SEM|M+SDO 0.7</td><td>125.59</td><td>X+CEDO 0.2</td><td>159.60</td></tr>
<tr><td>SEM|X+RDO 0.6</td><td>50.15</td><td>SEM|SASES+RDO 0.1</td><td>77.53</td><td>(skipped 2 items)</td><td></td><td>SEM|SASES+SSM 0.2</td><td>159.92</td></tr>
<tr><td>X+RDO 0.3</td><td>50.65</td><td>SSM+RDO 0.8</td><td>77.59</td><td>SEM|SDO 1.0</td><td>127.44</td><td>(skipped 3 items)</td><td></td></tr>
<tr><td>SASES+RDO 0.4</td><td>51.00</td><td>(skipped 3 items)</td><td></td><td>SEM|SASES+SDO 0.8</td><td>128.79</td><td>X+SSM 0.1</td><td>164.35</td></tr>
<tr><td>SEM|X+RDO 0.5</td><td>52.97</td><td>SEM|SSM+RDO 0.9</td><td>78.19</td><td>(skipped 9 items)</td><td></td><td>(skipped 4 items)</td><td></td></tr>
<tr><td>SEM|X+RDO 0.3</td><td>53.09</td><td>(skipped 2 items)</td><td></td><td>SEM|SSM+SDO 0.8</td><td>133.05</td><td>SASES+SSM 0.2</td><td>166.92</td></tr>
<tr><td>M+RDO 0.4</td><td>54.67</td><td>SEM|SSM+RDO 0.4</td><td>81.17</td><td>SEM|SSM+SDO 0.7</td><td>133.17</td><td>(skipped 3 items)</td><td></td></tr>
<tr><td>SEM|SASES+RDO 0.5</td><td>54.85</td><td>(skipped 12 items)</td><td></td><td>(skipped 20 items)</td><td></td><td>CSDO 1.0</td><td>168.96</td></tr>
<tr><td>M+RDO 0.2</td><td>56.40</td><td>M+CSDO 0.5</td><td>94.14</td><td>SEM|SASES 1.0</td><td>141.64</td><td>SEM|X+M 0.3</td><td>171.92</td></tr>
<tr><td>SASES+RDO 0.7</td><td>57.33</td><td>M+CSDO 0.6</td><td>94.71</td><td>SEM|SASES+SDO 0.4</td><td>142.23</td><td>M+CEDO 0.2</td><td>172.18</td></tr>
<tr><td>SEM|SASES+RDO 0.6</td><td>58.26</td><td>M+CSDO 0.4</td><td>97.53</td><td>SASES 1.0</td><td>144.12</td><td>X+M 0.4</td><td>172.72</td></tr>
<tr><td>SASES+RDO 0.3</td><td>58.46</td><td>(skipped 2 items)</td><td></td><td>(skipped 2 items)</td><td></td><td>M+CEDO 0.4</td><td>173.85</td></tr>
<tr><td>X+RDO 0.2</td><td>58.65</td><td>X+CSDO 0.6</td><td>102.01</td><td>SASES+M 0.2</td><td>146.47</td><td>(skipped 4 items)</td><td></td></tr>
<tr><td>M+RDO 0.5</td><td>59.36</td><td>X+CSDO 0.1</td><td>102.91</td><td>SEM|X+SDO 0.3</td><td>148.18</td><td>X+SSM 0.2</td><td>177.58</td></tr>
<tr><td>SEM|M+RDO 0.3</td><td>59.67</td><td>SASES+SDO 0.9</td><td>106.92</td><td>SEM|SASES+SSM 0.1</td><td>148.56</td><td>(skipped 6 items)</td><td></td></tr>
<tr><td>SEM|M+RDO 0.5</td><td>61.17</td><td>(skipped 6 items)</td><td></td><td>(skipped 3 items)</td><td></td><td>M+EDO 0.3</td><td>183.90</td></tr>
<tr><td>SEM|M+RDO 0.4</td><td>61.46</td><td>SASES+SDO 0.8</td><td>111.18</td><td>SASES+M 0.1</td><td>150.17</td><td>(skipped 2 items)</td><td></td></tr>
<tr><td>SASES+RDO 0.8</td><td>61.88</td><td>M+SDO 0.8</td><td>111.27</td><td>SASES+M 0.3</td><td>151.38</td><td>M+EDO 0.2</td><td>186.04</td></tr>
<tr><td>X+RDO 0.7</td><td>62.37</td><td>X+SDO 0.9</td><td>111.44</td><td>SASES+SSM 0.1</td><td>152.45</td><td>(skipped 35 items)</td><td></td></tr>
<tr><td>X+RDO 0.6</td><td>62.40</td><td>(skipped 2 items)</td><td></td><td>M+SDO 0.1</td><td>153.45</td><td>M 1.0</td><td>221.40</td></tr>
<tr><td>X+RDO 0.9</td><td>62.49</td><td>X+SDO 0.8</td><td>112.49</td><td>X+CEDO 0.3</td><td>154.21</td><td>(skipped 2 items)</td><td></td></tr>
<tr><td>M+RDO 0.8</td><td>62.69</td><td>M+SDO 0.9</td><td>112.53</td><td>X+EDO 0.4</td><td>154.88</td><td>CEDO 1.0</td><td>225.40</td></tr>
<tr><td>SEM|SASES+RDO 0.7</td><td>62.69</td><td>(skipped 8 items)</td><td></td><td>SSM+SDO 0.2</td><td>155.18</td><td>SEM|M 1.0</td><td>226.92</td></tr>
<tr><td>(skipped 2 items)</td><td></td><td>SEM|SASES+SDO 0.9</td><td>119.40</td><td>SEM|X 1.0</td><td>156.03</td><td>(skipped 4 items)</td><td></td></tr>
<tr><td>RDO 1.0</td><td>64.72</td><td>SSM+SDO 0.8</td><td>119.45</td><td>X+CEDO 0.1</td><td>156.12</td><td>EDO 1.0</td><td>231.32</td></tr>
<tr><td>(skipped 6 items)</td><td></td><td>X+SDO 0.1</td><td>119.63</td><td>X+M 0.2</td><td>156.17</td><td>(skipped 2 items)</td><td></td></tr>
<tr><td>SEM|RDO 1.0</td><td>68.67</td><td>SEM|X+SDO 0.9</td><td>120.00</td><td>(skipped 2 items)</td><td></td><td>SEM|SSM 1.0</td><td>238.65</td></tr>
<tr><td>(skipped 2 items)</td><td></td><td>SSM+SDO 0.9</td><td>120.55</td><td>X+EDO 0.1</td><td>157.83</td><td>SSM 1.0</td><td>239.26</td></tr>
</table>

Table 6.16.: Friedman ranks of *success per hour* for all 39 problems (both symbolic regression and the Boolean domain). Setups with the best proportion of used operators are emphasized with a bold font. The worse setups for each combination of operators may be skipped to safe space.

7. Functional Modularity

7.1. Introduction

In this chapter we analyze the properties of *functional modularity*, a concept introduced in [71] for detecting and measuring modularity in problems of automatic program synthesis by means of genetic programming. This content is based on our prior papers [70, 71], but here we show more results and analysis concerning problems from our benchmark suite presented in Section 4.2.

State of the art research demonstrates that GP, similarly to other methods of automatic program synthesis, suffers from lack of scalability. By this we mean that, given a problem for which certain instances are easy to solve using GP, solving larger instances of the same problem requires much more computational resources, or becomes insolvable (GP fails to find an optimal solutions within a reasonable computing time). Scalability is very desirable, because Canonical GP (presented in Section 2.2 on page 17) works well as long as the task is easy, i.e., the sought expression is relatively straightforward and the number of independent variables forming the input data is low. With larger problems, GP has much more troubles and often fails.

Therefore, it seems desirable to design a method for *automatic problem decomposition*, that decomposes the original problem into some *subproblems* which can be potentially easier to solve than the single, large problem. After solving all the subproblems delineated in this way, the ultimate solution could be then assembled from solutions found to these subproblems. Delivering a method to such automatic problem decomposition is our far-reaching research goal.

For this aim, we propose a specific methodology, which we refer to as *functional modularity*. As it will become clear in this chapter, this methodology also engages semantic aspects of GP programs. However, as the proposed concepts and algorithms are generic and do not depend on particular search algorithms nor search operators, we conduct our investigations abstracting from evolutionary computation. The formalization as well as experimental results presented in this chapter concern static, non-evolving samples of randomly generated programs. Thus, in a broader perspective, this part of the thesis may be considered as a study on statistical analysis of semantics of random programs.

7.2. Defining and Exploiting Modularity

The *No Free Lunch* theorem states that no search algorithm is better than another one when compared on a uniformly distributed population of all problems [136]. In this light, superiority of some algorithms tested on real-world problems indirectly demonstrates that some problems (problem instances) are more likely to occur in practice than others. This propels the quest for properties that are common for the real-world problems (or some classes of them) and that may be exploited by search algorithms, resulting in faster convergence. Such properties, studied in the past, include fitness-distance correlation [51], unimodality of the fitness landscape [93], and modularity [130]. This chapter concerns the last one.

The term 'module' is difficult to define without referring to a more specific background. There is quite firm agreement that a module is a *part* of solution (i.e., something that may be clearly delineated from the solution), such that it exhibits some form of independence (full of partial) from the remaining parts of solution (referred to as *context*). That independence is usually understood in terms of module's *contribution* to solution's performance (fitness) [133]. In an extreme case of a fully independent module, its contribution does not depend on the context, in which case the problem becomes *separable* [130] and the module may be optimized independently using *any* context. Such scenario is however unlikely in the real world, where modules and contexts are usually *interdependent,* which means that a module contributes to the overall fitness, but its contribution depends on the context. This dependency can take on different forms and result in module's observed contribution that is more complex than, e.g., the simple additive model as used in NK landscapes [53] or HIFF problems [132]. This complexity renders detection of modules difficult. The other factor that makes it hard is the presence of *multiple* modules, which is common in nontrivial problems, which gives rise to exponential explosion (see [130] for an in-depth analysis of modularity and related topics, like compositionality, accretive evolution, and the building block hypothesis).

The ability of a search algorithm to benefit from modularity is important, because a large proportion of real-world problems turn out to have interdependent modules. Detection and proper exploitation of modularity prior to or during search may speed up convergence, prevent code bloat, and cause the evolved solutions to be more robust. But most importantly, modularity is essential for scalability, which is a particularly difficult issue for genetic programming (GP), as demonstrated in past research [80].

If we agree that a module is a *part* of solution, then the strict definition of a module obviously depends on the representation of solutions that the search algorithm operates on. In genetic algorithms (GA) or evolutionary programming, where solutions are represented as vectors of variables, module has a natural interpretation of a subset of variables. In the tree-based GP, it is most common to equate a module with a subtree. Methods referring

to this module definition include evolutionary module acquisition [2], automatically defined functions [61], adaptive representations [112], and hierarchical genetic programming [5]. At this point it is worth to notice that the approaches proposed in past concerned not only canonical tree-based GP, but also other representations like Cartesian Genetic Programming [129]. However, what these previous approaches have in common is their purely *syntax-based* formulation of modularity; in this chapter, on the contrary, we aim at supporting the analysis of modularity with program *semantic*.

7.3. Defining Modules for Variable-based Representations

In general, the concept of a module for vectors of variables (typical for GA) and the concept of a module for variable-sized structures (typical for GP) are fundamentally different. However, modularity for vector representations seems to be a good starting point for introducing our idea of functional modularity. Thus, in following we briefly summarize a recent study on modularity in GA, related to Harik's work on learning gene linkage [40] and Goldberg's competent genetic algorithms [36].

In Watson's and De Jong's formulation [130, 20], given a set of variables V, a module is identified with a subset of variables $M \subset V$ such that the *linkage* between variables in M is tighter than the linkage between variables from M and the variables from the context, i.e., $V \backslash M$. The definition of linkage refers here to the notion of *context-optimal setting*. The context-optimal setting is a combination of values of variables from M which are optimal for at least one combination of values of context variables. Depending on the algorithm, the considered set of contexts may contain all possible contexts [133] or a sample of them [20]. The more such context-optimal settings exists, the tighter is the linkage between the module and the context.

Table 7.1 shows an exemplary instance of Boolean problem with four variables. The global optimum is 0000 with a maximal fitness value equal to 4. A module M consisting of two first variables (i.e., $M = \{v_0, v_1\}$) has only two possible context-optimal settings: 00 and 11. For context $\cdot\cdot 00$ its context-optimal settings equals 00 (fitness 4), for $\cdot\cdot 01$ it is also 00 (fitness 2), whereas for contexts $\cdot\cdot 10$ and $\cdot\cdot 11$ it is 11 (with fitness values equal to 2 and 3, respectively). A counterexample is a pair (v_2, v_3) with 4 context-optimal settings (for context $00\cdot\cdot$ it is 00, for $01\cdot\cdot$ — 01, for $10\cdot\cdot$ — 10, for $11\cdot\cdot$ — 11). This means that the pair (v_2, v_3) has the tightest possible linkage with other variables and, because of this full dependency, it does not constitute a module.

To summarize, we say that M is a module if the number of its context-optimal settings is smaller than the number of all possible settings of variables in M. If there is only one such optimal combination of variables then the module M is completely independent from the context, which means that the problem is separable. It turns out that, under assumption that the modules are hierarchically organized, it is possible to effectively and

[v_0	v_1]	v_2	v_3	*Fitness*
0	0	0	0	4
0	0	0	1	2
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	2
1	1	1	1	3

Table 7.1.: An instance of Boolean problem with four binary variables $v_0 \dots v_3$. A pair of variables v_0 and v_1 constitute a module $M = \{v_0, v_1\}$ with 2 distinct context-optimal settings (underlined values), whereas the combination of v_2 and v_3 is not a module, because this pair has 4 context-optimal settings (underlined values) — this equals the total number of possible settings of these variables.

robustly detect the modules without considering all possible contexts. This idea has been exploited by Watson, Thierens, and de Jong, who designed the hierarchical genetic algorithm (HGA) and have shown in [20] that it effectively solves a subclass of artificial hierarchical modular problems [19]. This result was obtained in the realm of Boolean problems, where enumerating the settings of a set of variables is possible; an extension of this approach to real-valued problems is still to come.

7.4. Functional Modularity

The aforementioned variable-based concepts of module and its context-optimal setting cannot be directly transplanted into the GP domain. First of all, variables in optimization problems lack natural counterparts in GP. Secondly, even if we identified a variable with, e.g., a specific *locus* in a GP tree, and treated a set of such *loci* as a module, then the interplay between such a module and the remaining part of the tree would be very complex and, in general, could not be easily modeled using fitness contributions (e.g., additive fitness contributions as defined in famous benchmarks like NK landscapes and HIFF [131]). Thirdly, adopting such a module definition still would not enable us to borrow some concepts from HGA [20], as it would be computationally too expensive to enumerate all possible settings of a module (a subtree in such case). And, last but not

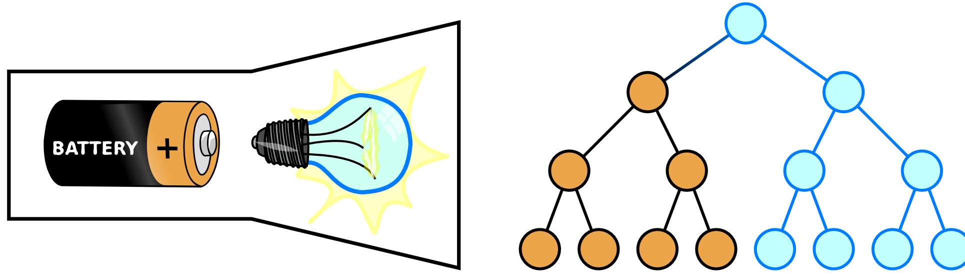


Figure 7.4.1.: A torch example.

least, the genotype-phenotype mapping in GP is many-to-one, i.e., different GP subtrees may have the same semantics, so considering all of them seems superfluous.

However, we hypothesize that discovery of a module is possible without finding its optimal settings. Using more general terms, we claim that a module in solution structure may be detected based on the structure (some characteristics) of the solution space alone, even if we do not know what is the best *value* (setting) for that module. To justify this claim, let us consider an illustrative optimization task of designing a battery-powered torch, with the optimized objective (fitness) being the time for which the torch'es brightness sustains a predefined threshold (expected operating time). Let us delineate the two main torch parts: the battery and the bulb (see Figure 7.4.1). For the sake of this thought experiment, the battery will play the role of a module, while the bulb will act as the context.

The common-sense knowledge suggests that the design of the battery is (at least to some extent) independent of the design of the bulb. Some battery designs are better than others, and some of them may be considered optimal, meaning that they maximize the overall quality (fitness) of the entire solution (torch). It is the case because these two components interact in a quite straightforward way, i.e., there are only a few parameters of the interface between them that matter (the voltage, the resistance of the bulb, the physical specification of the contacts that connect the battery with the bulb). The internal implementation details of battery and bulb are irrelevant as long as their external specification fit within certain norms.

Let us now point out an important feature of modularity in this example. We do not need to test the fitness of the torch to assess the quality of the battery it contains. There may exist other *quality measures,* which are capable to evaluate the module in a way that is consistent with the fitness function. For the above example, it might be the case that measuring the loss of battery voltage after it has been short-circuited for a certain time is sufficient to accurately estimate the expected operating time of the torch. Even if such a perfectly consistent measure does not exist, then it is likely that we can find some approximate surrogate for it, which is sufficiently correlated with the fitness function. The existence of the former, fully consistent measure would imply separability, the existence of the latter, approximate estimator — modular interdependence (see Section 7.1 and [130]).

This example illustrates the possibility of detecting a module in problem structure by discovering the proper (structural) decomposition of solution *and* finding an appropriate quality measure of this module. To distinguish such approach to defining and analyzing modularity in GP from the more common purely structural, syntax-based methods, we coin the term *functional modularity*.

Finding a fully consistent measure of module quality may be difficult or impossible, as it subsumes separability, which is infrequent in real-world design problems and unrealistic in GP context for the reasons listed at the beginning of this section. In general, the more interdependent a module and its context are, the more complex the relation between such measures and the fitness function. In such circumstances, rather than making a qualitative *decision* about module existence, it may be better to quantify the *degree* of modularity of a particular module candidate. Analogously, it seems more reasonable to consider multiple quality measures and evaluate their utilities for module search, than pursuing the search for the ultimate best-of-all quality measure that may never be found. These observations motivated our formal definition of the functional module introduced in following.

7.5. Formalization

Let X be the set of all programs (solution space of the problem of consideration), and let $f : X \to \mathbb{R}$ be a *maximized* fitness function.

Definition 6. A *binary decomposition function* (decomposition for short) is any invertible function $q : X \to P \times C$ that decomposes a solution into two components, i.e., such that:

$$\forall x \in X : \, q(x) = (p, c), \; q^{-1}(p, c) = x, \tag{7.5.1}$$

where p is called a *part* of x and c is the *context* of x in the q meaning. P and C are the sets of all possible (q-compatible) parts and contexts, respectively.

In following, we assume that q is given and fixed, therefore, when needed, p and c are written as $p(x)$ and $c(x)$ to emphasize that they are obtained from solution x. For the decomposition q to be non-trivial, $p(x) \neq x$ and $c(x) \neq x$ must hold.

In general, elements of X, P and C are *programs*. Or, if one would like to reserve the term 'program' to the piece of code that solves the *entire* problem, then the elements of P and C should be called *subprograms*. The solution decomposition function q must observe the constraints imposed by the syntax of the language used for representing solutions, so that at least the parts constitute independently executable pieces of code. However, in the simple case of type-less Koza-style GP considered in this thesis, the distinction between X, P and C is almost negligible — they all are sets of trees that may be generated given the set of terminals and set of functions (nonterminals) (see Section 2.2). The technical difference is that P may be restricted to smaller trees than X (so $P \subseteq X$), and C is a set

of degenerated trees, i.e. trees with a removed subtree (it is common in GP to mark the missing subtree with the '#' symbol, as in the following example: $x \cdot (x + \#)$).

Definition 7. A *part quality function* is any real-valued function $f_P : P \to \mathbb{R}$ assigning a real value to a given part.

A part quality function will be identified with a *subgoal* that parts are supposed to optimize, analogously to the way solutions optimize the fitness function f (though it is worth noticing that such a subgoal does not have to be explicitly known). We assume positive preference ordering on f_P, i.e., we aim at *maximizing* its value.

To enable practical realization we need to constrain f_P to some implementable form. As solution parts $p \in P$ are *programs*, two following classes of part quality functions seem natural: syntactic and semantic ones.

By syntactic part quality function we mean a function f_P that relies exclusively on the code of program p, i.e., how it *looks*. Such quality functions are appropriate for, among others, problems that are decomposable due to independency between particular components of program input, fed into a GP tree via terminals. A simple example could be here a bivariate symbolic regression aimed at finding the $3v_1^2 + 2v_2^2$ function: a decomposition into two univariate problems constrained to particular variables (v_1, v_2) is here obvious. f_P should in such a case prefer parts (program fragments) that use only some of the terminals (say, v_1), letting the remaining terminals (here: v_2) to be used in the context. Such decomposition related to structure of program's input data is sometimes possible thanks to domain knowledge; however, for many real-world problems this particular type of decomposability cannot be assumed.

As opposed to the syntax-based modularity framing, a nice property of the functional approach to modularity is its applicability to other, non-syntactic properties of parts. We focus here on specific class of such functions, which we call *semantic part quality functions*. Such functions investigate how a program *works* in order to assess its quality.

Definition 8. A *semantic part quality function* is a function $f_P : P \to \mathbb{R}$ measuring the similarity between the semantics of the part and certain fixed semantics s_P:

$$f_P(p) = similarity(s(p), s_P), \tag{7.5.2}$$

where $s(p)$ is a sampling semantics of part p, and $similarity(s_1, s_2)$ is a similarity measure defined, e.g., like in Formula 3.4.2 on page 35.

Each semantic part quality function is completely defined by the assumed similarity measure (see Section 3.4) and a *subgoal* represented by the associated semantics s_P. Under the assumption that the similarity measure is fixed, there is one-to-one correspondence between the semantic part quality function f_P and the semantics s_P. This is an analogous situation to the fitness function f defined as a similarity between actual semantics of an

individual and the target semantics defined by a problem (cf. Formula 3.4.4 that defines the most common way of evaluating individuals in GP).

Definition 9. A *monotonicity degree* (*monotonicity* for short) $m(f_P, f)$ of f_P with respect to f is a real-valued function that measures some form of monotonicity (strict, weak, or partial) between the values returned by f_P and the values returned by f, on an assumed set of programs and under certain assumed decomposition function.

In other words, monotonicity measures how much the quality of the part (in f_P sense) is correlated with the fitness function f. In some cases, such correlation could be calculated analytically for the entire population of solutions X. However, in the following experiments, we will estimate these indicators from samples of random programs.

Here, we equate monotonicity with the Spearman's rank correlation coefficient. Thanks to this, we can abstract from the metric scale of fitness and focus on the actual ordering of f and f_P values. Technically, this measure is equivalent to the Pearson's correlation coefficient with ranks substituted for f and f_P:

$$m(f_P, f) := \rho_X(\mathbf{R}f_P, \mathbf{R}f) = \frac{1}{\sigma_{\mathbf{R}f}\sigma_{\mathbf{R}f_P}} \sum_{x \in X} (\mathbf{R}f(x) - \overline{\mathbf{R}f})(\mathbf{R}f_P(p(x)) - \overline{\mathbf{R}f_P}), \quad (7.5.3)$$

where $\mathbf{R}f_P$ and $\mathbf{R}f$ are, respectively, raw scores of f_P and f converted to ranks (calculated within a sample of certain size). $\sigma_{\mathbf{R}f}$ and $\sigma_{\mathbf{R}f_p}$ denote the standard deviations of $\mathbf{R}f_P$ and $\mathbf{R}f$, respectively. Other reasonable definitions of monotonicity include Kendall's tau, gamma statistic, and ordinal contingency [43].

Definition 10. An *optimal part quality function* f_P^* is any part quality function with the highest monotonicity:

$$f_P^* = \arg \max_{f_P : P \to \mathbb{R}} m(f_P, f). \quad (7.5.4)$$

Definition 11. A problem given by solution space X and fitness function f is α*-modular* under the assumed solution decomposition function $q : X \to P \times C$ and monotonicity degree function $m(f_p, f)$ iff

$$m(f_P^*, f) \geq \alpha. \quad (7.5.5)$$

Let us now illustrate these notions in terms of the torch example presented earlier. In that case, X is the population of all possible torch designs and f is the torch fitness measure as defined in our thought experiment. The solution decomposition function q decomposes a torch $x \in X$ into a battery $p = p(x)$ and a bulb $c = c(x)$, and the sets P and C have the interpretation of, respectively, the populations of all batteries and all bulbs that are q-compatible with the torch design, i.e., batteries and bulbs that may be assembled into a torch using q^{-1}. The f_P functions are different battery quality measures: some of them are monotone with respect to f (e.g., the initial battery voltage), some of

them not (e.g., battery color). The torch design problem is α-modular if there exist a battery (part) quality function f_P that has monotonicity $\geq \alpha$.

To summarize, a problem is α-modular if two conditions are fulfilled:

1. There exists a way of decomposing the candidate solutions of the problem into parts (solution decomposition function q).

2. For the given q, there exists a part quality function with monotonicity $\geq \alpha$.

The rationale behind such definition of functional modularity is obviously motivated by the potential benefits from problem decomposition. Its exploitation could proceed as in the following scenario (though other approaches are conceivable). If we knew the solution decomposition function q and the corresponding optimal part quality function f_P^*, and if its monotonicity with respect to f was sufficiently high, we could decompose the problem using q. Then, we could use f_P^* to search for an optimal part p^*; this search would take place in the solution space P, which we expect to be smaller than X (which is true for, e.g., decomposition functions that split entire programs into program fragments, as in the following section). Finally, having found p^* (or its good approximation), we could constrain the search in X by considering only solutions x such that $x = q^{-1}(p^*, c)$, i.e., searching only the space of contexts. If the space X is combinatorial, this implies exponential reduction of search space cardinality, and thus potentially immense gains in the expected runtime of a search algorithm.

Let us note that in the extreme case of part quality function being perfectly monotonous with respect to f (i.e., $m(f_P^*, f) = 1$), such proceeding would guarantee finding the optimal solution, provided we could find p^*. On the other hand, this is a degenerate case: perfect correlation between part quality function f_P and fitness function f would imply that f does not depend on context; f_P would account for (explain) *all* the variability of f across solutions in X. In such a case, context could be ignored, so no problem decomposition in the semantic sense would take place.

7.6. Experiments

In a long run, we are interested in exploiting the modularity for the sake of improving search convergence and other properties of the search algorithm and/or the evolved solutions. However, prior to exploiting modularity, it is essential to verify whether an assumed way of problem decomposition reveals modularity in the considered problem. In other words, as mentioned earlier, the necessary precondition for modularity exploitation is modularity *detection*. Thus, in the experimental part of this chapter we investigate mainly the distribution of monotonicity within different GP problems and its relationships with fitness, without actually running the evolution. More technically, all the results quoted in

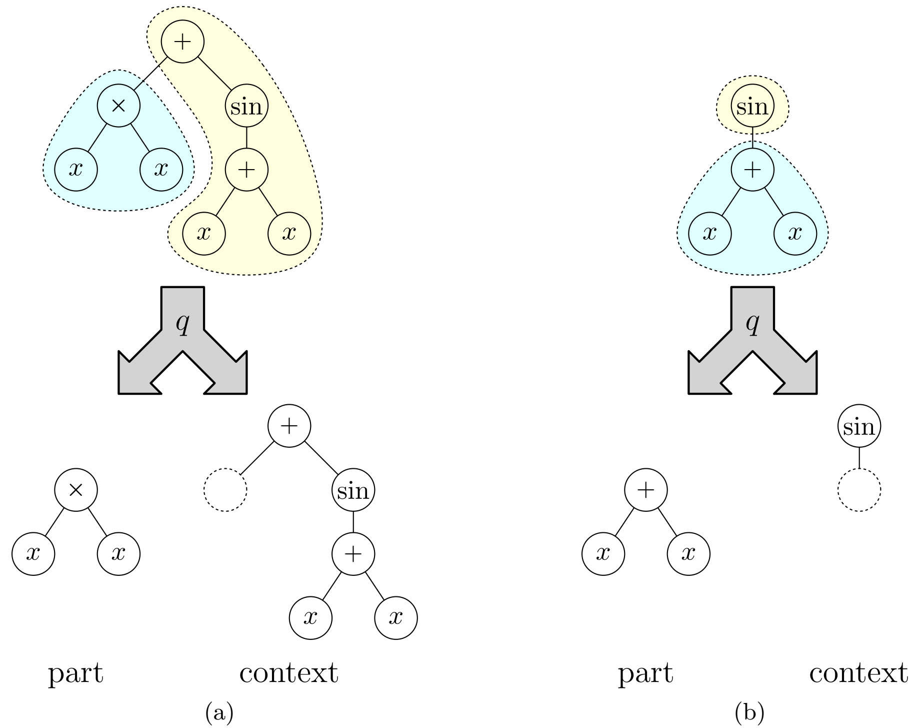


Figure 7.6.1.: Examples of applying decomposition function defined as $q(x)$ = ('the leftmost child of the root node of x', 'the remaining part of x').

following have been obtained by computing appropriate statistics from a random sample of GP individuals.

We carry out all computation for all 39 problems from our benchmark suite (Section 4.2). We expect different structures of modularity to emerge for such range of problems. For example, we anticipate *even parity* (PAR) to exhibit weaker modularity than other Boolean problems due to hypothesized 'ruggedness' of its fitness landscape, resulting from extreme sensitivity to the states of single variables (flipping any input variable flips the desired output value).

In this chapter, to partition a program into a part p and a context c, we employ decomposition function q defined as $q(x)$ = ('the leftmost child of the root node of x', 'the remaining part of x') (see Formula 7.5.1). For instance, the result of decomposition function applied to a tree defining expression $(x \cdot x) + \sin(x + x)$ is $q\left((x \cdot x) + \sin(x + x)\right) = (p, c) = \left((x \cdot x)\,,\, (\# + \sin(x + x))\right)$, where # denotes an empty tree branch — see Figure 7.6.1a. If a root node has only one child, then the context contains only this root, e.g. $q\left(\sin(x + x)\right) = (p, c) = \left((x + x)\,,\, \sin(\#)\right)$ — see Figure 7.6.1b. If a program contains only one node and thus the leftmost child of the root is absent, we ignore the individual and generate another one in its place.

Even if we assume a fixed form of semantic similarity used by semantic part quality function (see Formula 7.5.2), the number of all semantic quality functions is typically very

Algorithm 7.1 Generating random subgoals

```
 1: procedure RandomSubgoals(N)                        ▷ N — number of subgoals
 2:     S ← SpecialSubgoals()                ▷ add special, domain specific semantics
 3:     for i ← 1...N do                                ▷ add N random semantics
 4:         repeat
 5:             repeat
 6:                 x ← Ramped-Half-and-Half(2, 15)
 7:             until root node of x has at least one child
 8:             (p, c) ← Decompose(x)                   ▷ using q as defined in text
 9:             s ← Semantics(p)
10:             v ← ∑_i s_i                     ▷ sum of all elements of semantics s
11:         until s ∉ S and v < 10^15
12:         S ← S ∪ {s}
13:     return S                           ▷ subgoals are identified by semantics
14: end procedure
```

Algorithm 7.2 Generating sample of random individuals

```
 1: procedure Sample-of-Individuals(N)                 ▷ N — number of samples
 2:     P ← ∅                                       ▷ set of generated individuals
 3:     T ← ∅
 4:     d ← 0                                     ▷ number of generated duplicates
 5:     for i ← 1...N do
 6:         repeat
 7:             repeat
 8:                 x ← Ramped-Half-and-Half(2, 15)
 9:             until root node of x has at least one child
10:             (p, c) ← Decompose(x)                   ▷ using q as defined in text
11:             s_x ← Semantics(x)
12:             s_p ← Semantics(p)
13:             if (s_x, s_p) ∈ T then
14:                 d ← d + 1
15:                 if d = 3 · N then        ▷ if too many duplicates were generated
16:                     return P                 ▷ return current set P (|P| < N)
17:                 else
18:                     continue              ▷ skip this duplicated individual x
19:             v_x ← ∑_i s_{x,i}              ▷ sum of all elements of semantics s_x
20:             v_p ← ∑_i s_{p,i}              ▷ sum of all elements of semantics s_p
21:         until v_x < 10^15 and v_p < 10^15
22:         P ← P ∪ {x}
23:         T ← T ∪ {(s_x, s_p)}           ▷ remember the unique pair of semantics
24:     return P
25: end procedure
```

large: there are as many of them as there are possible semantics (s_P in Formula 7.5.2). Because, for most problems it is impossible or impractical to enumerate all of them, we investigate a set S of thousand unique semantics generated as follows. Firstly, we create random individuals using the Ramped Half-and-Half method (see Algorithm 2.3 on page 18), with depths varying from 2 to 15. Then we decompose each program using the decomposition function q and we compute the semantics of the part obtained in this way. If this semantics is unique, then we add it to the set of the part quality functions. We repeat this process to obtain 1000 unique semantics. When applying this procedure to the symbolic regression domain, we filter out semantics with large magnitudes — more specifically, we have accepted a semantics as a candidate for s_P only if the sum of absolute values of all its elements is less than 10^{15}.

Additionally, for each considered benchmark problem, we added several special semantics to S (the set of part quality functions):

- target semantics — i.e., the expected semantics of an ideal solution to a given problem,
- for symbolic regression domain: semantic with all elements equal to v, where $v \in \{\pm1, \pm2, \pm5, \pm10, \pm100, \pm1000\}$, i.e., 12 selected semantics in total,
- for Boolean domain: two semantics with all elements equal to either *true* (1) or *false* (0).

Each additional semantics, except the target one, has all elements equal to the same predefined value. For example, the semantics with values equal to +100 for each fitness case will be denoted as '+100 semantics' in following.

We added these additional semantics to the set S before generating random ones, therefore we have in total 1012 part quality functions for each symbolic regression problem and 1003 for each Boolean problem. Algorithm 7.1 presents the entire procedure for generating subgoals.

Ideally, one would estimate monotonicity of each subgoal by enumerating all possible solutions in X (see Formula 7.5.3). This is unfortunately computationally infeasible. Thus, we estimate monotonicity from a sample of up to 1,000,000 individuals. Algorithm 7.2 shows the procedure for generating this sample of individuals.Technically, we generate random GP programs using the Ramped Half-and-Half method (Algorithm 2.3), with depths varying from 2 to 15. Individuals with program semantics or part semantics (obtained by decomposition function q defined above) having large magnitude are rejected (technically, if a sum of absolute values of semantics elements is $\geq 10^{15}$). The procedure of generating random programs is almost identical to the one used in generating random subgoals described in Algorithm 7.1, but with an additional condition imposed on semantics of entire programs.

Our objective is to gather one million programs with unique combination of program semantics and part semantics. However, some semantics of randomly generated programs are much more frequent than others. Also, for some problems generating so many semantically unique individuals is impossible (when the total number of all possible semantics is much less than one million, e.g., for problem IPAR4) or very unlikely. Therefore, we give up after generating three million duplicates, i.e., random individuals which have been already generated. Because of this, the monotonicity degree for some problems will be estimated from smaller number of programs.

Boolean Domain Issue

The symmetry of the space of Boolean functions is an important feature that must not be ignored in this study. The choice of instructions is that the *a priori* probabilities of generating an individual with semantic $s(x)$ and that of generating an individual with semantic $\overline{s(x)}$ are in general very close. In particular, for the specific set of instructions used later in problems with prefix 'N' (NPAR, NMUX, NMAJ, and NCMP), these probabilities are *exactly* the same. Obviously, the same can be said about the semantics of parts, $s(p)$ and $\overline{s(p)}$. As a consequence, an individual that contributes positively to monotonicity of a subgoal s_P, necessarily contributes negatively to monotonicity of subgoal $\overline{s_P}$. Over the entire sample, these contributions can almost compensate each other and cause the resulting monotonicities to be very close to zero, making it difficult to observe any interesting regularities.

To compensate for this, we employ a de-symmetrized fitness function (cf. Formula 3.4.4):

$$f(x) = \max\{similarity(s(x), t),\ similarity(s(x), \bar{t})\}, \tag{7.6.1}$$

where t is a target semantics of a problem and $\bar{t}$ is target semantics of the negated version of the original problem t.

So, we consider the original and the negated task equivalent, and an individual is rewarded either for optimizing t or optimizing $\bar{t}$, whatever it is better at. We adopt an analogous modification to part quality functions f_P (cf. Formula 7.5.2):

$$f_P(p) = \max\{similarity(s(p), s_P),\ similarity(s(p), \overline{s_P})\}.$$

7.7. Results

7.7.1. Monotonicity Distribution

Figures 7.7.1 and 7.7.2 present the monotonicity of subgoals for several problems selected from the benchmark suite presented in Section 4.2 on page 38. The subgoals have been ordered according to the increasing value of monotonicity. Some special subgoals are marked with vertical lines. The plots for the remaining benchmarks problems, presented

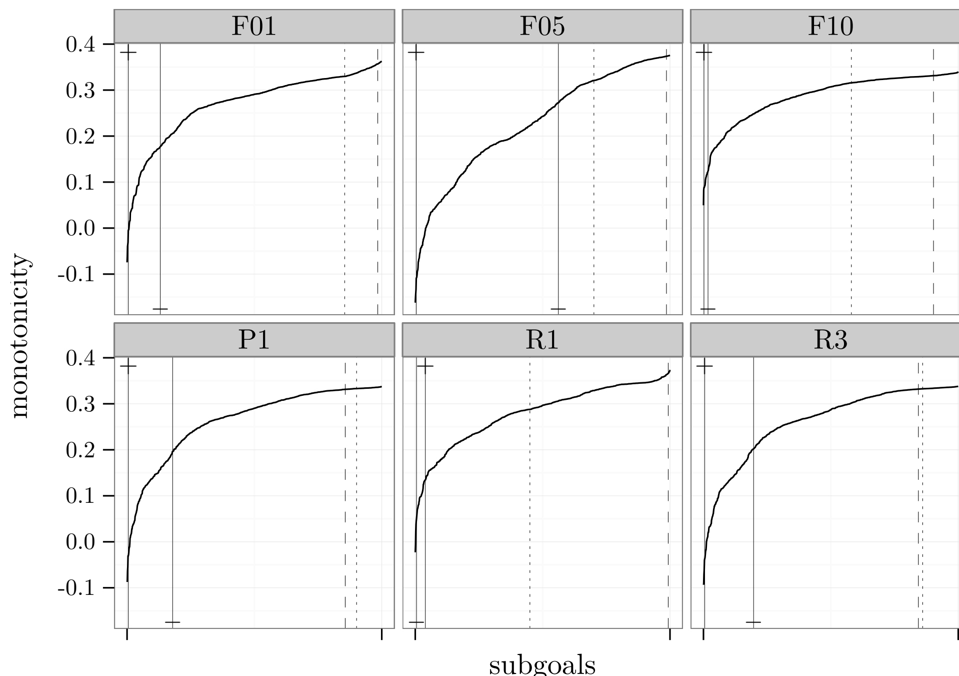


Figure 7.7.1.: Monotonicity of all subgoals for selected symbolic regression problems estimated from 1,000,000 random individuals. Vertical lines mark special subgoals: target semantics (dashed line), zeros (dotted), +100 or −100 (solid line annotated with + or − respectively).

in Appendix A, look very similar to these shown here.

It is easy to notice that, for all problem instances, monotonicity significantly varies across subgoals. However, there is evident difference between symbolic regression and Boolean tasks. For the former, most subgoals have monotonicity above the average, and relatively few of them have monotonicity near to zero. The share of subgoals with negative monotonicity is small, and usually does not exceed 4% of the total number of subgoals (max is 4.15% for F05). For Boolean problems, the situation is fundamentally different. In particular, highly-monotonous subgoals are infrequent. Also, quite many subgoals are 'deceptive' in the sense that they have remarkably negative monotonicity. However, we hypothesize that such a large number of subgoals with monotonicity below zero results from the de-symmetrization procedure applied to all Boolean problems.

This result indicates that, in the space of semantics of parts (subgoals), there are points ('good' subgoals) with the property that if the semantic of the part becomes more similar to one of them, then the fitness of the entire program is likely to increase. And conversely, there are also 'bad' subgoals with the inverse property. In other words, the 'good' subgoals are the subgoals for which the target's fitness landscape (reflecting the similarity of program to target semantics) and the subgoal's fitness landscape (reflecting the similarity of part to subgoal) can be aligned one to another in a way that causes them look similar. The 'bad' subgoals miss such property.

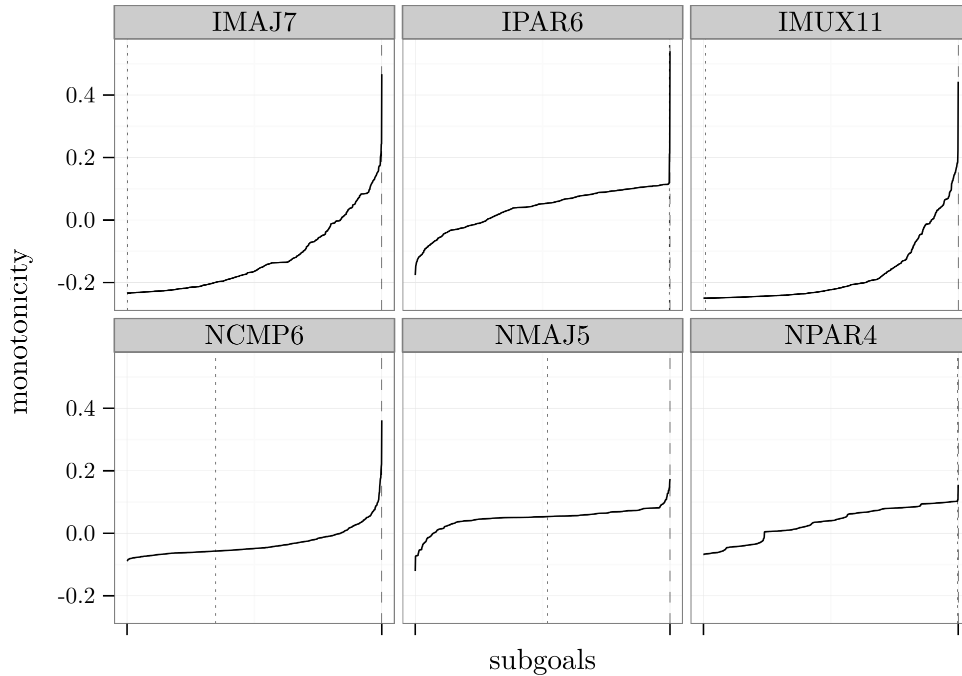


Figure 7.7.2.: Monotonicity of all subgoals for selected Boolean problems estimated from 1,000,000 random individuals. Vertical lines mark special subgoals: equal to target semantics (dashed line) or zeros (dotted).

For all tested problems, the subgoal equivalent to the target semantics has high monotonicity. This suggests that it is quite easy to construct a context which does not change the semantics produced by a part, or which modifies it only slightly. In an extreme case, if a part solves the problem alone, it is likely that also a whole program solves it. This is especially visible for the Boolean domain — such subgoals have the highest monotonicity for all tasks, because it is very easy there to construct a context that either propagates an unmodified result produced by the part or (due to the de-symmetrization) just negates it. Because of this phenomenon, we observe a prominent peak of monotonicity at the right-most end of all charts in Figure 7.7.2. The monotonicities of lower-ranked subgoals (more to the left in the charts) are remarkably lower. However, this decline is more prominent for problems like CMP or PAR, and are less sudden for others (MAJ and MUX).

In terms of α-monotonicity (Formula 7.5.5), R0 is approximately 0.38-modular as the maximum monotonicity over all subgoals amounts here to 0.3795. This is the most modular problem from the symbolic regression domain. IPAR6 problem has the maximal subgoal (here equivalent to the target semantics) monotonicity around 0.54 (precisely: 0.5403), which is the highest value of all Boolean tasks. However, the second subgoal for that problem has monotonicity equal to 0.2142 which is more than twice less than the subgoal equivalent to the target semantics.

Table 7.2 presents ranges and standard deviations of subgoal monotonicity values for all

Problem	*min*	*max*	*stdev*
F01	-0.0745	0.3625	0.0757
F02	-0.0624	0.3645	0.0721
F03	-0.0690	0.3652	0.0732
F04	-0.0607	0.3649	0.0709
F05	-0.1620	0.3757	0.1174
F06	-0.0776	0.3606	0.0768
F07	0.0106	0.3695	0.0691
F08	-0.0068	0.3639	0.0719
F09	0.0524	0.3344	0.0521
F10	0.0496	0.3401	0.0523
F11	0.0415	0.3292	0.0511
F12	-0.0775	0.3236	0.0842
P1	-0.0872	0.3374	0.0812
P2	-0.1507	0.3783	0.1120
P3	-0.0655	0.3653	0.0714
R0	-0.1612	0.3795	0.1177
R1	-0.0228	0.3737	0.0681
R2	-0.0151	0.3578	0.0636
R3	-0.0935	0.3379	0.0837

Problem	*min*	*max*	*stdev*
ICMP6	-0.0812	0.3979	0.0446
ICMP8	-0.1451	0.4434	0.0707
IMAJ5	-0.1078	0.3611	0.0675
IMAJ6	-0.0574	0.3863	0.0648
IMAJ7	-0.2345	0.4668	0.1118
IMUX11	-0.2500	0.4425	0.1051
IMUX6	-0.1413	0.3504	0.0675
IPAR4	-0.0600	0.2125	0.0324
IPAR5	-0.1295	0.4462	0.0532
IPAR6	-0.1764	0.5403	0.0647
NCMP6	-0.0886	0.3613	0.0452
NCMP8	-0.1424	0.3889	0.0585
NMAJ5	-0.1210	0.1737	0.0313
NMAJ6	-0.1201	0.3199	0.0634
NMAJ7	-0.1150	0.2426	0.0585
NMUX11	-0.2274	0.2981	0.0840
NMUX6	-0.1094	0.1857	0.0223
NPAR4	-0.0688	0.1554	0.0531
NPAR5	-0.1125	0.2555	0.0636
NPAR6	-0.1483	0.3342	0.0661

Table 7.2.: Minimal and maximal values, and standard deviations of subgoal monotonicity for all problems.

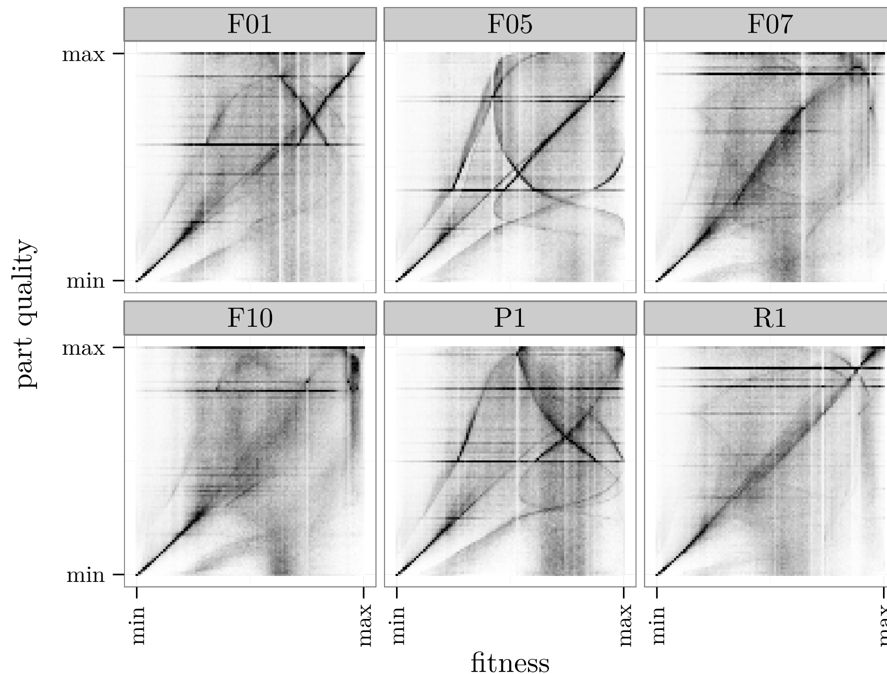


Figure 7.7.3.: 2D rank histograms of relation between fitness and value of the best part quality functions for selected symbolic regression problems. Each column is normalized — for each fitness the most frequent part quality value is shown in black.

39 problems. The figures confirm that the structure of monotonicity substantially varies between different problems. This table shows both the variation of maximal monotonicity, and the dispersion of quality among sampled subgoals. For instance, F05, P2, and R0 are symbolic regression problems with the most varying monotonicity. For Boolean domain, IMAJ7, IMUX11, and NMUX11 have the highest variance among subgoals. This may indicate that those problems are more modular and give more hope for being automatically decomposed.

On the other hand, all symbolic regression problems have quite similar values of the maximal monotonicity which varies from 0.3236 for F12 to 0.3795 for R0, but about half of them have monotonicity around 0.36. This distinguishes this group of problems from the Boolean domain because for them the maximal monotonicity varies from 0.1554 (NPAR4) to 0.5403 (IPAR6). This suggests substantial difference in modularity of our benchmark problems.

7.7.2. Relation Between Part Quality and Fitness

Previous section describes the values of monotonicity for all tested subgoals. Here we present in more detail particular subgoals and the relation between the fitness of an entire program and the quality of its part, as measured with respect to these subgoals.

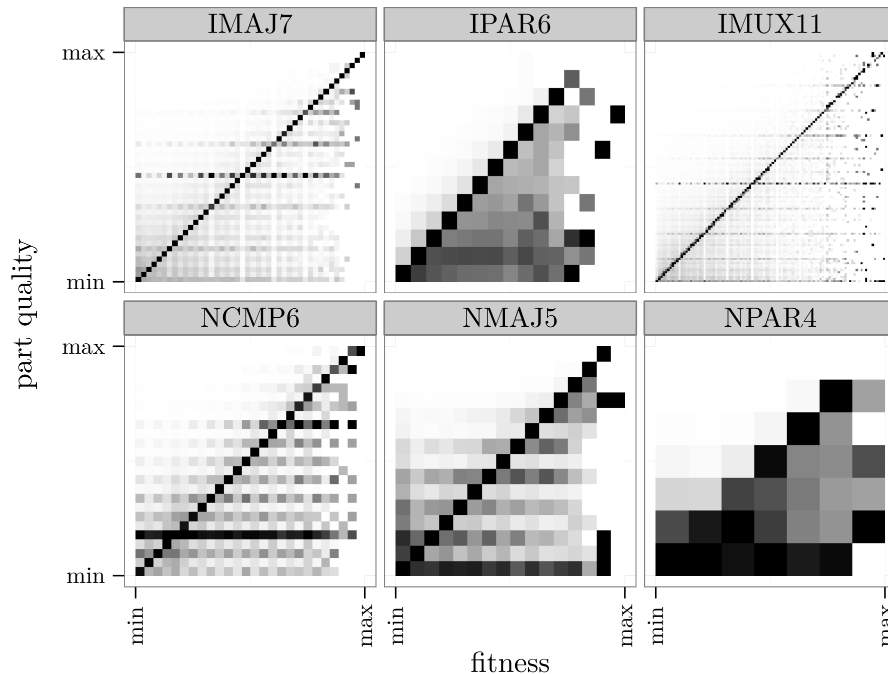


Figure 7.7.4.: 2D rank histograms of relation between fitness and value of the best part quality functions for selected Boolean problems. Each column is normalized — for each fitness the most frequent part quality value is shown in black.

Figures 7.7.3 and 7.7.4 show the best found part quality functions for a set of selected problems. Each chart is a two-dimensional histogram presenting the relation between the ranks of fitness and the ranks of part quality values. Darker shades represent more frequent combinations in our set of randomly generated problems. For clarity, we normalized each column, so that the most frequent value of part quality for each fitness bin is printed in black. In this way, the gray shades range from white to black for each program semantics (column), no matter frequent that program's semantic is (note that, e.g., individuals with high fitness appear very rarely in the sample of random individuals). Histograms for the remaining problems, not shown here but presented in Appendix A, look very similar to those in Figures 7.7.3 and 7.7.4.

In these histograms, visible dark horizontal lines stretching through the whole range of fitness represent programs with a common part (therefore with the same part quality) which are very frequent in the sample of random individuals. However, the interesting fact is that contexts in these programs are able to 'ignore' the output produced by the part and produce final output resulting in such varying fitness value. Therefore, existence of horizontal lines means, that such 'independent' contexts, obtained by the decomposition functions used in our experiments (see Section 7.6), are quite common. On the other hand, vertical bright lines denote individuals with very infrequent fitness values. Because there are very few programs with such unique fitnesses, after the normalization, columns

representing such fitness bins are almost white with only several dark points.

In general, it would be advantageous if the left-top triangle of presented histograms was white and only right-bottom part of them was darker. In such case, optimizing the part quality function would be equivalent to narrow the possible range of fitness from the bottom. In other words, it would mean that the better part quality the better minimal possible fitness of individual. On the contrary, if there is a lot of dark areas above the diagonal it is likely that individuals with good part quality have bad fitness at the same time. This means that better part quality does not necessary imply better programs. If individuals laid only on the diagonal, it would suggest that a problem is separable and it could be solved just by optimizing the part quality function.

Histograms generated for the symbolic regression problems (Figure 7.7.3) show that improving the very bad part quality values effectively improves the whole program. This is not surprising, as the worst individuals often returns huge values, and individuals with parts returning smaller values have both better part quality value and, very likely, better fitness. This happens, because parts of good programs usually have to produce outputs with smaller values so the context could transform them to the desired final values. After the initial phase, the part quality function not always could put the search easily on the right track as for F07 or R1 problems. Often, the trace generated by the part quality function is a bit more deceptive like for F05 or P1, and sometimes it is almost neutral like for F10 where the trace is blurred.

For all Boolean problems it is characteristic that the diagonal is very distinct. This happens because the best subgoal equals the target semantics, as discussed in the previous section. Moreover, it seems that most sampled individuals not laying on the diagonal are localized below it, in the desirable bottom-right part of histogram. However, the de-symmetrization process has a big influence of this effect. Therefore, utilizing the concept of functional modularity as described in this chapter probably needs to exploit some form of de-symmetrization procedure.

Another interesting observation is that the histograms for most Boolean problems are checked. This effect is caused by the fact that it is much more likely that a random program generated from functions used in our setup (as described in Section 4.2 on page 38) has an even number of ones in its semantics. Generating such program is about 3–4 times more probably than creating a random individual with an odd number of ones in its semantics.

Figure 7.7.5 on the following page shows examples of the worst subgoals, i.e. subgoals with the minimal monotonicity. At first glance it could seem that some of them are quite good. For instance, the relation between fitness and part quality for problems F10 or F05 forms a kind of funnel, and it looks like promoting individuals with a particular value of part quality (not the maximal) is advantageous. However, this will work only for the worst individuals and, in contrast to the better subgoals, does not differentiate better programs for which we observe almost ideal horizontal line.

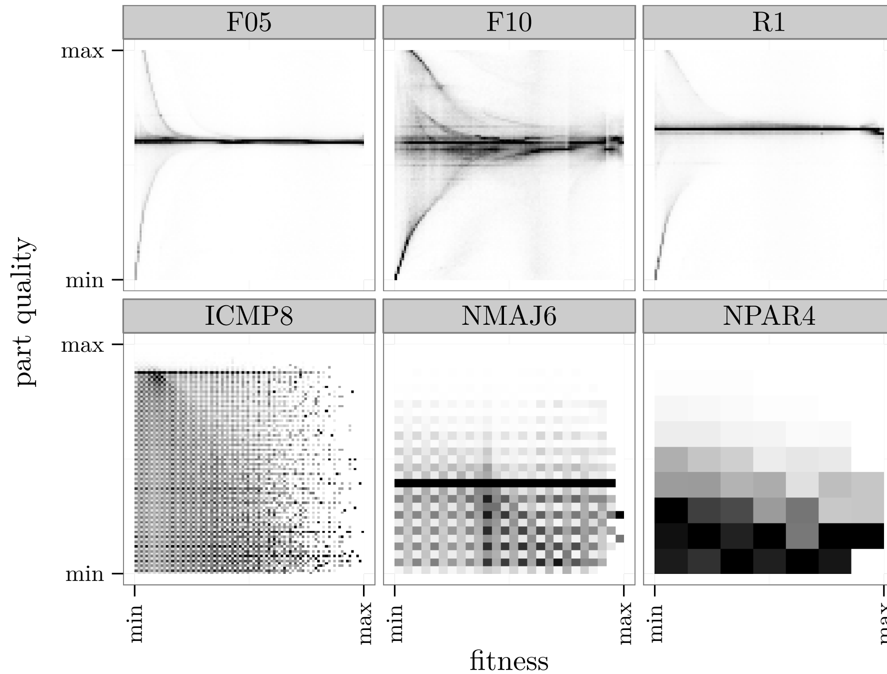


Figure 7.7.5.: 2D rank histograms of relation between fitness and value of the worst part quality functions (i.e. with minimal monotonicity) for selected problems. Each column is normalized — for each fitness the most frequent part quality value is shown in black.

7.7.3. Relation Between Monotonicity and Fitness

Section 7.7.1 focused on the global, unconditional, distribution of monotonicity in the sample of subgoals. The objective of the analysis presented in following is to relate those results to the fitness of complete solutions. More technically, we verify whether the well-performing solutions are more modular than the other ones.

We do that by analyzing in more detail the sample used for the estimation of monotonicity. For each individual x, we extract its part p, calculate its semantics $s(p)$, and find a part quality function which best scores p, i.e. such subgoal s_P that is most similar to $s(p)$ (see Equation 7.5.2). This results in each individual having one subgoal assigned to it. Finally, we can associate a fitness of a program with the monotonicity of the subgoal assigned to it. Figures 7.7.6 and 7.7.7 present two dimensional histograms of this relation. Again, for clarity each column is normalized as in previous section. Additionally, we plot in red the mean values of monotonicity for each fitness group.

Almost all presented graphs show that in average the better individuals the better subgoal is most similar to a part of this individual. This observation confirms that the monotonicity measure actually quite well scores the semantic part quality functions. In

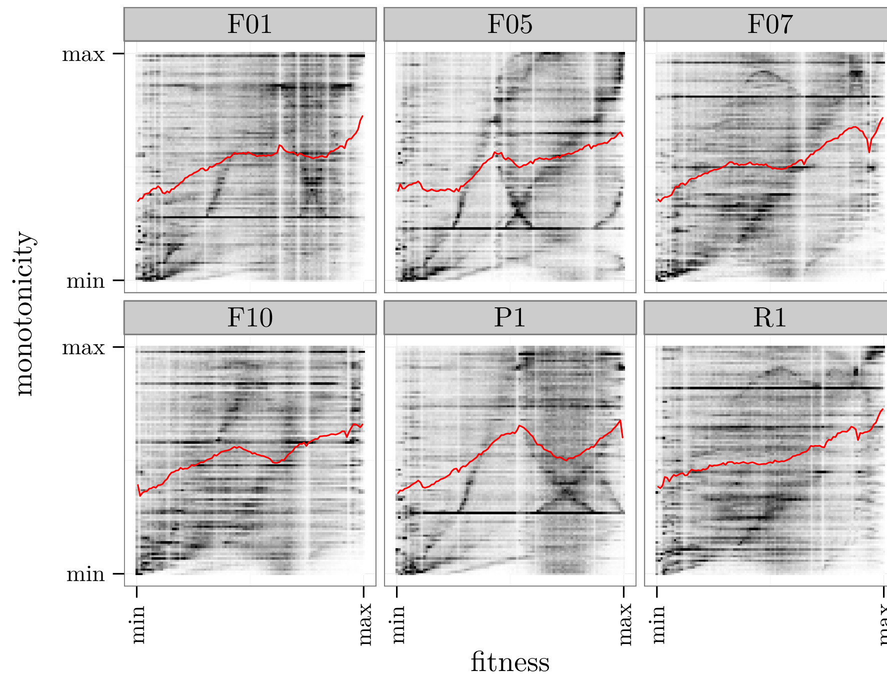


Figure 7.7.6.: 2D rank histograms of relation between fitness and monotonicity assigned to individuals. Each column is normalized — for each fitness group the most frequent monotonicity is shown in black. Mean monotonicity for each fitness group is drawn by red line.

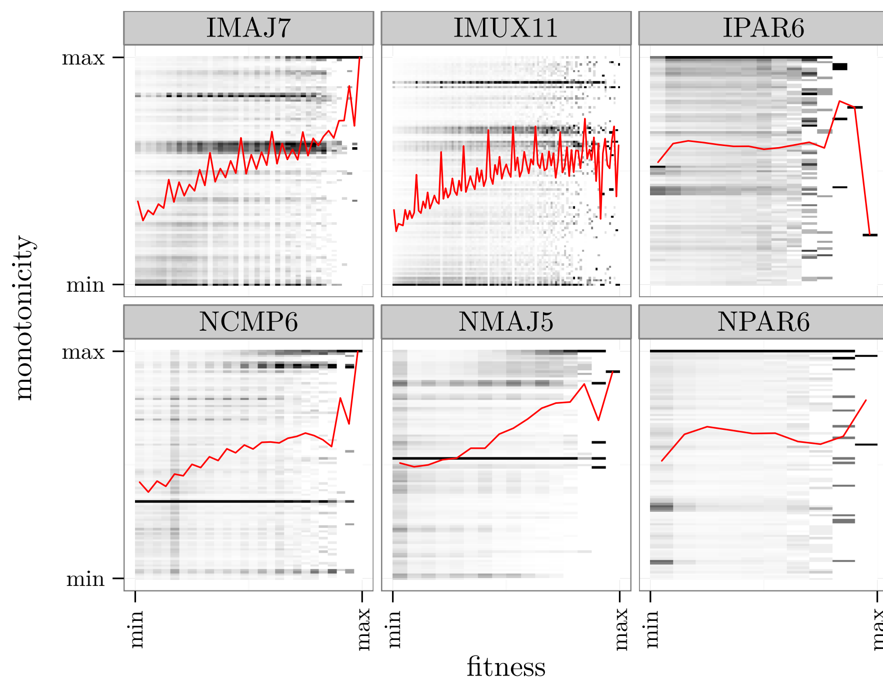


Figure 7.7.7.: 2D rank histograms of relation between fitness and monotonicity assigned to individuals. Each column is normalized — for each fitness group the most frequent monotonicity is shown in black. Mean monotonicity for each fitness group is drawn by red line.

other words, subgoals with higher monotonicity more often rate highly fit individuals than the worse.

However, it is important to remember than only small number from the random programs have high fitness. Therefore, the results presented in most right part of the charts are calculated from very few samples (sometimes just a single observation) and cannot be statistically significant . We observe there a high variation of the mean monotonicity — sometimes ending with a high peak or drastic drop. Nevertheless, for most benchmark problems the tendency of increasing monotonicity with increasing fitness is quite well visible. To problems with a noticeable but temporary decrease in monotonicity for better half of fitness values belong: F08–F12, P1, P2, R0, R3. For these problems we observe quite well mean monotonicity for the middle value of fitness, then some decrease followed again by subsequent increase in value.

For Boolean problem, the positive correlation between fitness and average monotonicity is even better visible. The exception is for two problems NPAR5 and NPAR6 for which it seems that the negative tendency takes place for the most fitness value range. Another obvious observation is a remarkable bifurcation of the monotonicity that follows the pattern of alternating fitness values (for most Boolean problems the bins of fitness contain just a single fitness value). This artefact arises from the different frequency of an even number of ones in semantics of randomly generated programs, as discussed in the previous section. A bit different fluctuation of monotonicity for IMUX11 comes from the aggregation of fitness values in particular histogram bins.

7.8. Discussion and Conclusions

The experimental results authorize us to formulate the following claims:

1. For some problem, different subgoals tend to have significantly different monotonicity.
2. Problem instances display different structure of monotonicity, meant as the characteristics of the distribution of monotonicity across the subgoals.
3. For most problem, the monotonicity of solution's closest subgoal positively correlates with its fitness.

We can hypothesize that some problems tend to be less modular (close-to-uniform distribution of monotonicity), and some more modular (significantly non-uniform distribution of monotonicity). This gives hope for delineating the class of semantically modular problems. For such problems, decomposition based on functional modularity is likely to provide better scalability and enable solving problems that remain intractable using contemporary computational resources.

Many questions and research issues pertaining to this study remain open. As requiring a human to provide the appropriate decomposition function q is far from realistic in most real-world scenarios, a complete decomposition algorithm should be able to discover it autonomously. In particular, for the sake of clarity, in this chapter we used a specific decomposition function q that defines part as the left-hand subtree of the root node. It should be emphasized that this form of decomposition has potential drawbacks: the context is forced to aggregate the output produced by the part with the values returned by the context using a *single* operation. The existence of ideal solutions for the problems considered in this thesis implies that such aggregation is possible, yet not necessarily for *all* possible subgoals. Moreover, even if for some subgoal there exists an optimal context that makes the entire solution ideal, then from the viewpoint of search effort of an evolutionary run (measured, e.g., as the expected number of evaluations needed to find the optimum), some subgoals and some contexts may be easier to optimize than others.

We hypothesize that using other decomposition functions q can alleviate this difficulty. A simple example of such function could be q that defines part as the left-hand child of left-hand child of the root, and context as the remaining part of the tree. In such a case, context would have *two* operations at its disposal to combine the output of the part with the values computed by the context, and it could use different right-hand arguments for these operations. Therefore, there would be more 'degrees of freedom' in the search process and it should be easier to evolve an optimal context.

It is interesting to note that this line of reasoning leads in the end to decomposition functions q that define part as the leftmost *leaf* of the tree. Such decomposition definitely gives a lot to say to the context, so that the abovementioned risk of the context not being able to incorporate the semantics of the part is neglectable. However, a part defined in such a way is very unlikely to have any impact on the overall semantics of the entire tree. Thus, we conclude that using different decomposition functions allows us to control the trade-off between the difficulty of optimizing the context and the contribution of part semantics to the entire solution.

Another question pertains to computational efficiency: in terms of the expected time required to find the optimal solution, does it pay off to decompose the problem into two subproblems and solve them independently, given the extra overhead imposed by the analysis of monotonicity? And if yes, then when? Finally, a more complete theory supporting this approach would be of much help.

The above experimental analysis is a proof-of-concept demonstrating that functional modularity may be helpful for characterizing the compositionality and difficulty of a problem. Knowing the structure of modularity for a particular problem is the first step for effective exploitation of monotonicity, which we will pursue in future research.

Here however, we show a preliminary experiments only to convince that the presented concept of functional modularity may be exploited in practice. To this aim, we additionally

Setup	*Rank*	*Setup*	*Rank*	*Setup*	*Rank*
B0.05|X+M 0.1	**20.47**	B0.20|X+M 0.3	33.35	B0.30|X+M 0.2	45.58
B0.10|X 1.0	**20.91**	B0.15|X+M 0.5	33.46	B0.20|X+M 0.7	45.79
B0.05|X 1.0	**21.21**	X+M 0.5	35.26	B0.30|X+M 0.3	46.03
B0.10|X+M 0.1	**21.55**	B0.10|X+M 0.6	35.28	X+M 0.9	47.78
B0.05|X+M 0.2	22.42	**B0.25|X 1.0**	**35.29**	B0.25|X+M 0.7	48.42
B0.05|X+M 0.3	22.77	B0.20|X+M 0.4	36.09	B0.30|X+M 0.6	48.88
B0.10|X+M 0.2	22.96	**B0.25|X+M 0.1**	**36.79**	B0.30|X+M 0.4	48.90
B0.10|X+M 0.3	23.09	B0.15|X+M 0.6	37.72	B0.30|X+M 0.5	48.96
B0.15|X+M 0.1	**24.73**	B0.25|X+M 0.2	37.72	B0.20|X+M 0.8	49.04
B0.15|X+M 0.2	25.76	B0.05|X+M 0.7	37.76	B0.15|X+M 0.9	49.38
X+M 0.2	**25.82**	B0.20|X+M 0.5	37.87	B0.30|X+M 0.7	50.74
B0.20|X 1.0	**26.08**	X+M 0.6	38.87	B0.05|X+M 0.9	51.18
B0.05|X+M 0.4	26.12	B0.10|X+M 0.7	39.99	B0.10|X+M 0.9	51.23
X 1.0	**26.67**	**B0.30|X 1.0**	**40.87**	B0.30|X+M 0.8	51.42
B0.15|X+M 0.3	26.85	**B0.30|X+M 0.1**	**41.24**	**M 1.0**	**51.69**
B0.20|X+M 0.1	**27.14**	B0.25|X+M 0.3	41.26	B0.20|X+M 0.9	52.08
B0.10|X+M 0.4	27.53	X+M 0.7	41.73	B0.25|X+M 0.8	52.68
X+M 0.1	27.77	B0.20|X+M 0.6	42.19	**B0.05|M 1.0**	**52.76**
B0.15|X 1.0	**28.06**	B0.15|X+M 0.7	42.26	B0.25|X+M 0.9	53.76
B0.05|X+M 0.5	29.68	B0.25|X+M 0.6	43.26	**B0.10|M 1.0**	**54.59**
B0.20|X+M 0.2	30.13	B0.05|X+M 0.8	43.74	B0.30|X+M 0.9	55.18
X+M 0.3	30.36	X+M 0.8	43.74	**B0.20|M 1.0**	**55.82**
B0.10|X+M 0.5	30.45	B0.25|X+M 0.5	43.81	**B0.15|M 1.0**	**56.45**
B0.15|X+M 0.4	31.10	B0.25|X+M 0.4	45.26	**B0.25|M 1.0**	**57.36**
X+M 0.4	31.76	B0.10|X+M 0.8	45.29	**B0.30|M 1.0**	**58.08**
B0.05|X+M 0.6	32.36	B0.15|X+M 0.8	45.38		

Table 7.3.: Friedman ranks of *success ratio* performance on all 39 problems.

run simple genetic programming evolution with parameters as in previous chapters (for details see Section 4.3). The only difference, we introduce here, is that we calculate the fitness used in selection phase as a weighted sum of the 'original' fitness f and the best part quality function f_P. Technically, during the evolution we used a *minimized* fitness computed as:

$$fitness(x) = (1-\alpha) \cdot d(s(x), t) + \alpha \cdot d(s(p), s_P), \tag{7.8.1}$$

where $d(s(x), t)$ is the 'original' fitness value of individual x (i.e., the distance between actual semantics of the individual x and the target semantics t — see Formula 3.4.3 on page 35), $s(p)$ is semantics of the part of individual x (the part is acquired by applying the binary decomposition function q defined in Section 7.6), and s_P is the best subgoal found for each problem individually. We tested six values of α: $\alpha \in \{0.05, 0.1, 0.15, 0.2, 0.25, 0.3\}$.

For each problem from the Boolean domain, the best subgoal (i.e., the most monotone) always equals to the target semantics. For problems from symbolic regression domain, the best subgoal is one from the randomly generated. The only exception is problem P2 for which the most monotone subgoal is also equivalent to the target semantics (at least, in the generated random sample there was no better subgoal than the target — see Section 7.6 for description how these subgoals were generated).

Table 7.3 presents a ranking of setups which exploit functional modularity (denoted as $B\alpha|X + M\,\beta$) and which do not, i.e., the control setups $X + M\,\beta$ (α is the weight used in Equation 7.8.1, and β is the probability of applying mutation operator). As it is easily noticed, setups rewarding individuals which have good parts performs a bit better than the control experiment. The difference between ranks of the best setup B0.05|X+M 0.1 and the best control setup X+M 0.2 is not statistical significant (using Holm's post-hoc procedure).

It is important to remind, that this experiment is just a proof-of-concept which actually demonstrates the usefulness of the functional modularity applied even in such simplistic way. However, to use the functional modularity in practice, there is a need to devise a more suitable methods for exploiting this concept. This will be the goal of our further research.

8. Conclusions

8.1. Summary

Automatic creation of computer programs (understood on different levels of abstraction — from simple algebraic expressions to programs written in languages like C++ or Java) is a means to solve many practical problems (see Section 2.3). Therefore genetic programming, a leading approach that provides such possibility, is an important tool to help deal with them. However, despite the number of successes achieved by GP, this methodology still exhibits a few weaknesses that need to overcome.

We tried to demonstrate in this thesis that semantic approaches could make a breakthrough in GP and lead to qualitatively better results. A number of presented experimental results provide evidence that our proposed semantic extensions of genetic programming can indeed outperform standard GP on most benchmark problems in a significant manner. This conclusion concerns not only success ratio, which reflects how likely is a method to produce a perfect result. It is important to notice that, even if GP is unable to perfectly solve a task, it still can be a useful technique for *approximating* solutions for these problems. Also in this respect, i.e., in terms of the errors committed on the training and tests sets, the semantic-aware methods proposed in this thesis usually yield better results than standard GP.

8.2. Contributions

The main contributions of this thesis may be summarized as follows:

- Presentation of different types of program semantics and their characterization from the viewpoint of genetic programming. From the wide range of possibilities, we decided to concentrate on the sampling semantics, because it offers a reasonable compromise between precision (meant as the extent to which semantics reflects the behavior of a program) and handiness (meant as easy in which it can be analyzed and handled in algorithms, e.g., compared). We proposed an appropriate formalization of this notion.
  [Chapter 3]
- Proposition and formalization of the concept of *desired semantics*. We proposed five methods to exploit this concept. We consider these semantic extensions of GP one

of the main contributions of this dissertation.
[Chapter 6]

- Proposition of the *functional modularity* concept. We introduced a formalization that enables semantic decomposition of a problem into smaller subproblems. We demonstrated selected properties of this formalism, primarily the *monotonicity degree*. The promising results attained in this part of the dissertation form a proof-of-concept that brings convincing rationale for feasibility of this concept.
[Chapter 7]

- Proposition of a method for semantically unique initialization of initial population. We showed that this method usually helps evolution in attaining better results.
[Chapter 5, Section 5.2]

- Formation of quite a big benchmark suite comprising 39 problems (19 symbolic regression tasks, and 20 logic function synthesis task). The problems in the suite vary in difficulty, with some of them being quite hard, as demonstrated in the experimental part of this thesis.
[Chapter 4]

- Experimental verification of the effectiveness of all proposed methods. We performed both comparative experiments as well as experiments aimed at investigating the properties of the new methods proposed here. It turns out that particularly our RDO and SDO operators are highly profitable, especially for the problems in the Boolean domain. The presented results demonstrate also that simultaneous use of multiple semantic extensions is even more beneficial.
[Chapter 6]

- The original proposition of *Success per hour* metric (see Section 4.5.1), used to measure the performance of algorithms. The attractive feature of this metric consists in its practical perspective: it provides realistic estimate of the expected computational effort using contemporary hardware.
[Chapter 4, Section 4.5.1]

- Practical demonstration of the importance of the assortment of fitness cases. We showed that the selection of the input data (examples) for a training set has essential influence on the achieved performance of GP algorithms. In our experiments, the success ratio of some types of GP algorithms for some benchmarks varied immensely, up to from 12% to 100%, depending only on the set of selected fitness cases. Therefore, we conclude that in GP, a task should be identified not only by a target function and a range of possible input variables, but also by a strictly defined, fixed

set of fitness cases.
[Chapter 4, Section 4.4]

- A software implementation of both the proposed and selected existing semantic extensions to GP. The implementation is quite well optimized to minimize the runtime. Thanks to the employed cache mechanism, the evaluation of evolved programs is even six times faster than the default evaluation method implemented in the ECJ package [83].
[Chapter 4, Section 4.5.2]

- Preliminary results of a practical application of functional modularity. We performed experiments that tested a simple method based on that concept. Results showed that it is possible to identify a good part quality function that improves results obtained by an evolutionary process.
[Chapter 7, Section 7.8]

- Experimental verification of the performance achieved by Nguyen *et al.*'s [126, 99] semantic crossover (SASES) and semantic mutation (SSM) operators on our benchmark suite. These results gave us a point of reference for performance assessment of our semantic extensions.
[Chapter 5, Section 5.3 and 5.4]

8.3. Future Work

A number of various well-performing semantic extensions to genetic programming, proposed in this thesis as well as elsewhere, shows that considering semantics of evolved programs in addition to syntax is profitable. Therefore, it seems that methods that identify and exploit such semantic aspects of programs form a very promising direction in the GP research.

We especially believe that our innovative approach of desired semantics and other methods developed in this spirit will lead to essential breakthrough in the field of genetic programming. There are several directions in which this research can develop. In the variant of operators presented in Chapter 6, we used the desired semantics in a very strict way, searching the library for the part that exactly matches the desired values (i.e., the values that make the entire program return the correct output). However, to exploit the presented idea, exact desired values are not necessary. It may be profitable to only *narrow* the space of the considered parts in such way that the genetic operators could choose 'semantically well fitting' part to be inserted into the parent individual. In other words, we hypothesize that a method that relies on inexact, approximate concept of desired semantics can still be more effective than standard GP operators, which essentially perform quite a haphazard juggling of code pieces. This hypothesis, if true, would have another

important consequence: our approach could be also applied to problems where the instructions are not invertible. Our next goal is to investigate such problems and develop methods that can operate effectively in such more demanding environments.

Another interesting research endeavor would be to investigate the relation between the size of evolved programs and the success of evolutionary run. In particular, it may be interesting to confront the averaged best-of-run sizes of individuals from successful runs (i.e. perfect solutions for a given problem) with those from unsuccessful runs (i.e. programs which do not solve the stated problem). The presented results, especially for the experiments involving our desired semantics operators (e.g., RDO or SDO) from Section 6.4.2, suggest that such analysis could help to explain why some runs failed.

Yet another direction of possible research is to design an algorithm exploiting the concept of functional modularity (see Chapter 7) in a practical manner. The initial experiments presented in this thesis assumed that one is given a good part quality function in advance. However, in practice this is rarely the case: without some extra insight into domain knowledge, defining solution parts and the corresponding part quality functions is challenging. Determining part quality functions in an additional preliminary stage, though possible, is computationally quite demanding. We postulate therefore that such functions should (and could) be found on-line during actual evolutionary process. Therefore, we will endeavor to design such a method and demonstrate it applicability and performance.

Last but not least, we plan also to propose alternative definitions of sampling semantics, desired semantics of a context, and other formalisms, tailored to other types of genetic programming. Particularly, our methods seem to be quite easily adaptable to linear genetic programming [6] and cartesian genetic programming [92]. It would be interesting to verify whether such analogs of the methods presented here would be equally effective in such alternative programming environments.

A. Appendix

In this thesis we presented some results of performed experiments. The attached CD contains much more details in form of tables and graphs (over 40,000 files). The content of CD is divided according to the chapters and sections where the experiments were described:

- Chapter 5 — Semantically-oriented Search Operators
  - Section 5.2 — Population Initialization
  - Section 5.3 — Crossover
  - Section 5.4 — Mutation
  - Section 5.5 — Summary
- Chapter 6 — Desired Semantics
  - Section 6.4 — Results
  - Section 6.5 — Discussion and Conclusions
- Chapter 7 — Functional Modularity
  - Section 7.7.1 — Monotonicity Distribution
  - Section 7.7.2 — Relation Between Part Quality and Fitness
  - Section 7.7.3 — Relation Between Monotonicity and Fitness
  - Section 7.8 — Discussion and Conclusions

The CD includes:

- Tables (as *txt* and *csv* files):
  - Success ratio,
  - Success per hour,
  - Mean time [ms],
  - Mean size of individuals (averaged over all runs and all generations),
  - Mean depth of individuals (averaged over all runs and all generations),
  - Mean success generation — in which generation an ideal was found,
  - Mean best-of-run error,
  - Median best-of-run error,
  - Mean best-of-run hits,
  - Mean best-of-run test error,

  - Median best-of-run test error,
  - Mean best-of-run size,
  - Mean best-of-run depth.

- Rankings (with results of Holm's post-hoc statistical procedure comparing both the best setup and the best *control* setup) of:
  - Success ratio,
  - Median best-of-run error,
  - Median best-of-run test error,
  - Success per hour.

- Graphs showing changes in time of:
  - Diversity — unique number of semantics (solid line) and unique fitness values (dotted line),
  - Size — minimal and maximal value, and three quartiles are shown,
  - Error — minimal value, and three quartiles are shown (only in "Section 6.4 — Results").

Bibliography

[1] Alexandros Agapitos, Michael O'Neill, Anthony Brabazon, and Theodoros Theodoridis. Learning environment models in car racing using stateful genetic programming. In *Proceedings of the 2011 IEEE Conference on Computational Intelligence and Games*, pages 219–226, Seoul, South Korea, 31 August - 3 September 2011. IEEE.

[2] Peter J. Angeline and Jordan B. Pollack. The evolutionary induction of subroutines. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, pages 236–241, Bloomington, Indiana, USA, 1992. Lawrence Erlbaum.

[3] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, November 1998.

[4] W. Banzhaf, P. Nordin, R. Keller, and F. Francone. *Genetic Programming: An Introduction. On the automatic Evolution of Computer Programs and its Application.* Morgan Kaufmann, 1998.

[5] Wolfgang Banzhaf, Dirk Banscherus, and Peter Dittrich. Hierarchical genetic programming using local modules. Technical Report 50/98, University of Dortmund, Dortmund, Germany, 1998.

[6] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann, San Francisco, CA, USA, January 1998.

[7] Lawrence Beadle and Colin Johnson. Semantically driven crossover in genetic programming. In Jun Wang, editor, *Proceedings of the IEEE World Congress on Computational Intelligence*, pages 111–116, Hong Kong, 1-6 June 2008. IEEE Computational Intelligence Society, IEEE Press.

[8] Lawrence Beadle and Colin G. Johnson. Semantic analysis of program initialisation in genetic programming. *Genetic Programming and Evolvable Machines*, 10(3):307–337, September 2009.

[9] Lawrence Beadle and Colin G Johnson. Semantically driven mutation in genetic programming. In Andy Tyrrell, editor, *2009 IEEE Congress on Evolutionary Computa-*

tion, pages 1336–1342, Trondheim, Norway, 18-21 May 2009. IEEE Computational Intelligence Society, IEEE Press.

[10] Nitin Bhatia and Vandana. Survey of nearest neighbor techniques. *CoRR*, abs/1007.0085, 2010.

[11] Tobias Blickle and Lothar Thiele. Genetic programming and redundancy. In J. Hopf, editor, *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)*, pages 33–38, Im Stadtwald, Building 44, D-66123 Saarbrücken, Germany, 1994. Max-Planck-Institut für Informatik (MPI-I-94-241).

[12] Walter Bohm and Andreas Geyer-Schulz. Exact uniform initialization for genetic programming. In Richard K. Belew and Michael Vose, editors, *Foundations of Genetic Algorithms IV*, pages 379–407, University of San Diego, CA, USA, 3–5 August 1996. Morgan Kaufmann.

[13] Dimo Brockhoff, Tobias Friedrich, Nils Hebbinghaus, Christian Klein, Frank Neumann, and Eckart Zitzler. Do additional objectives make a problem harder? In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 765–772, New York, NY, USA, 2007. ACM.

[14] Edmund K. Burke, Steven Gustafson, and Graham Kendall. Diversity in genetic programming: An analysis of measures and correlation with fitness. *IEEE Transactions on Evolutionary Computation*, 8(1):47–62, 2004.

[15] Edmund K. Burke, Steven Gustafson, Graham Kendall, and Natalio Krasnogor. Is increased diversity in genetic programming beneficial? an analysis of the effects on performance. In Ruhul Sarker, Robert Reynolds, Hussein Abbass, Kay Chen Tan, Bob McKay, Daryl Essam, and Tom Gedeon, editors, *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, pages 1398–1405, Canberra, 8-12 December 2003. IEEE Press.

[16] Kumar Chellapilla. Evolutionary programming with tree mutations: Evolving computer programs without crossover. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 431–438, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[17] Kenneth L. Clarkson. Nearest-neighbor searching and metric space dimensions. In *In Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*. MIT Press, 2006.

[18] Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In John J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187, Carnegie-Mellon University, Pittsburgh, PA, USA, 24-26 July 1985.

[19] Edwin D. de Jong, Richard A. Watson, and Dirk Thierens. A generator for hierarchical problems. In *GECCO '05: Proceedings of the 2005 workshops on Genetic and evolutionary computation*, pages 321–326, New York, NY, USA, 2005. ACM.

[20] Edwin D. de Jong, Richard A. Watson, and Dirk Thierens. On the complexity of hierarchical problem solving. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1201–1208, New York, NY, USA, 2005. ACM.

[21] J. Derrac, S. García, D. Molina, and F. Herrera. A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. *Swarm and Evolutionary Computation*, 2011.

[22] The Free Dictionary. Definition of semantics by the free online dictionary, thesaurus and encyclopedia, 2012.

[23] Stephen Dignum and Riccardo Poli. Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat. In Dirk Thierens, Hans-Georg Beyer, Josh Bongard, Jurgen Branke, John Andrew Clark, Dave Cliff, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Tim Kovacs, Sanjeev Kumar, Julian F. Miller, Jason Moore, Frank Neumann, Martin Pelikan, Riccardo Poli, Kumara Sastry, Kenneth Owen Stanley, Thomas Stutzle, Richard A Watson, and Ingo Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1588–1595, London, 7-11 July 2007. ACM Press.

[24] V. Èerný. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41–51, 1985. 10.1007/BF00940812.

[25] Pedro G. Espejo, Sebastian Ventura, and Francisco Herrera. A survey on the application of genetic programming to classification. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 40(2):121–144, March 2010.

[26] Candida Ferreira. Gene expression programming: a new adaptive algorithm for solving problems. rejected for publication, 2000.

[27] L.J. Fogel. *On the Organization of Intellect.* University of California, Los Angeles - Engineering, 1964.

[28] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In Guenther Raidl, Franz Rothlauf, Giovanni Squillero, Rolf Drechsler, Thomas Stuetzle, Mauro Birattari, Clare Bates Congdon, Martin Middendorf, Christian Blum, Carlos Cotta, Peter Bosman, Joern Grahl, Joshua Knowles, David Corne, Hans-Georg Beyer, Ken Stanley, Julian F. Miller, Jano van Hemert, Tom Lenaerts, Marc Ebner, Jaume Bacardit, Michael O'Neill, Massimiliano Di Penta, Benjamin Doerr, Thomas Jansen, Riccardo Poli, and Enrique Alba, editors, *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954, Montreal, 8-12 July 2009. ACM. Best paper.

[29] A. Frank and A. Asuncion. UCI machine learning repository, 2010.

[30] M. Friedman. A comparison of alternative tests of significance for the problem of m rankings. *The Annals of Mathematical Statistics*, 11(1):86–92, 1940.

[31] Milton Friedman. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association*, 32(200):675–701, 1937.

[32] Alex S. Fukunaga. Automated discovery of composite sat variable-selection heuristics. In Rina Dechter and Richard S. Sutton, editors, *AAAI/IAAI*, pages 641–648. AAAI Press / The MIT Press, 2002.

[33] S. Garcıa and F. Herrera. An extension on statistical comparisons of classifiers over multiple data sets for all pairwise comparisons. *Journal of Machine Learning Research*, 9(2677-2694):66, 2008.

[34] Fred Glover. Tabu search-part i. *ORSA Journal on Computing*, 1(3):190–206, Summer 1989.

[35] Fred Glover and Claude McMillan. The general employee scheduling problem: an integration of ms and ai. *Comput. Oper. Res.*, 13(5):563–573, May 1986.

[36] David E. Goldberg. *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[37] Steven Gustafson, Edmund K. Burke, and Graham Kendall. Sampling of unique structures and behaviours in genetic programming. In Maarten Keijzer, Una-May O'Reilly, Simon M. Lucas, Ernesto Costa, and Terence Soule, editors, *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, pages 279–288, Coimbra, Portugal, 5-7 April 2004. Springer-Verlag.

[38] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, pages 1312–1317. AAAI Press, 2011.

[39] Julia Handl, Simon C. Lovell, and Joshua Knowles. Multiobjectivization by decomposition of scalar cost functions. In *Proceedings of the 10th international conference on Parallel Problem Solving from Nature: PPSN X*, pages 31–40, Berlin, Heidelberg, 2008. Springer-Verlag.

[40] G.R. Harik. *Learning Gene Linkage to Efficiently Solve Problems of Bounded Difficulty Using Genetic Algorithms.* PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[41] Ami Hauptman, Achiya Elyasaf, Moshe Sipper, and Assaf Karmon. GP-rush: using genetic programming to evolve solvers for the rush hour puzzle. In Guenther Raidl, Franz Rothlauf, Giovanni Squillero, Rolf Drechsler, Thomas Stuetzle, Mauro Birattari, Clare Bates Congdon, Martin Middendorf, Christian Blum, Carlos Cotta, Peter Bosman, Joern Grahl, Joshua Knowles, David Corne, Hans-Georg Beyer, Ken Stanley, Julian F. Miller, Jano van Hemert, Tom Lenaerts, Marc Ebner, Jaume Bacardit, Michael O'Neill, Massimiliano Di Penta, Benjamin Doerr, Thomas Jansen, Riccardo Poli, and Enrique Alba, editors, *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 955–962, Montreal, 8-12 July 2009. ACM.

[42] Ami Hauptman and Moshe Sipper. Evolution of an efficient search algorithm for the mate-in-N problem in chess. In Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 78–89, Valencia, Spain, 11-13 April 2007. Springer.

[43] T. Hill and P. Lewicki. *Statistics: methods and applications: a comprehensive reference for science, industry, and data mining.* StatSoft, Inc., 2006.

[44] N. X. Hoai, R. I. McKay, D. Essam, and R. Chau. Solving the symbolic regression problem with tree-adjunct grammar guided genetic programming: The comparative results. In David B. Fogel, Mohamed A. El-Sharkawi, Xin Yao, Garry Greenwood, Hitoshi Iba, Paul Marrow, and Mark Shackleton, editors, *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002*, pages 1326–1331. IEEE Press, 12-17 May 2002.

[45] J. H Holland. Adaptive algorithms for discovering and using general patterns in growing knowledge-bases. 1980.

[46] S. Holm. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics*, pages 65–70, 1979.

[47] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[48] Takuya Ito, Hitoshi Iba, and Satoshi Sato. Depth-dependent crossover for genetic programming. In *Proceedings of the 1998 IEEE World Congress on Computational Intelligence*, pages 775–780, Anchorage, Alaska, USA, 5-9 May 1998. IEEE Press.

[49] David Jackson. Promoting phenotypic diversity in genetic programming. In Robert Schaefer, Carlos Cotta, Joanna Kolodziej, and Guenter Rudolph, editors, *PPSN 2010 11th International Conference on Parallel Problem Solving From Nature*, volume 6239 of *Lecture Notes in Computer Science*, pages 472–481, Krakow, Poland, 11-15 September 2010. Springer.

[50] Colin Johnson. Genetic programming crossover: Does it cross over? In Leonardo Vanneschi, Steven Gustafson, Alberto Moraglio, Ivanoe De Falco, and Marc Ebner, editors, *Proceedings of the 12th European Conference on Genetic Programming, EuroGP 2009*, volume 5481 of *LNCS*, pages 97–108, Tuebingen, April 15-17 2009. Springer.

[51] Terry Jones and Stephanie Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 184–192. Morgan Kaufmann, 1995.

[52] Ilan Kadar, Ohad Ben-Shahar, and Moshe Sipper. Evolving boundary detectors for natural images via genetic programming. In *19th International Conference on Pattern Recognition, ICPR 2008*, pages 1–4, Tampa, Florida, USA, December 8-11 2008.

[53] S.A. Kauffman. Adaptation on rugged fitness landscapes. In D. Stein, editor, *Lectures in the Sciences of Complexity, Santa Fe Institute Studies in the Sciences of Complexity*, pages 527–618. Addison Wesley, 1989.

[54] Maarten Keijzer. Improving symbolic regression with interval arithmetic and linear scaling. In Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa, editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 70–82, Essex, 14-16 April 2003. Springer-Verlag.

[55] Bijan KHosraviani, Raymond E. Levitt, and John R. Koza. Organization design optimization using genetic programming. In Maarten Keijzer, editor, *Late Breaking Papers at the 2004 Genetic and Evolutionary Computation Conference*, Seattle, Washington, USA, 26 July 2004.

[56] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[57] Joshua D. Knowles, Richard A. Watson, and David Corne. Reducing local optima in single-objective problems by multi-objectivization. In *Proceedings of the First International Conference on Evolutionary Multi-Criterion Optimization*, EMO '01, pages 269–283, London, UK, UK, 2001. Springer-Verlag.

[58] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[59] J. R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N. S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, volume 1, pages 768–774, Detroit, MI, USA, 20-25 August 1989. Morgan Kaufmann.

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

[61] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge Massachusetts, May 1994.

[62] John R. Koza, Sameer H. Al-Sakran, and Lee W. Jones. Automated re-invention of six patented optical lens systems using genetic programming. In Hans-Georg Beyer, Una-May O'Reilly, Dirk V. Arnold, Wolfgang Banzhaf, Christian Blum, Eric W. Bonabeau, Erick Cantu-Paz, Dipankar Dasgupta, Kalyanmoy Deb, James A. Foster, Edwin D. de Jong, Hod Lipson, Xavier Llora, Spiros Mancoridis, Martin Pelikan, Guenther R. Raidl, Terence Soule, Andy M. Tyrrell, Jean-Paul Watson, and Eckart Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1953–1960, Washington DC, USA, 25-29 June 2005. ACM Press.

[63] John R. Koza and Riccardo Poli. Genetic programming. In Edmund K. Burke and Graham Kendall, editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, chapter 5, pages 127–164. Springer, 2005.

[64] K. Krawiec. Genetic programming-based construction of features for machine learning and knowledge discovery tasks. *Genetic Programming and Evolvable Machines*, 4:329–343, 2002.

[65] Krzysztof Krawiec. *Evolutionary Feature Programming: Cooperative learning for knowledge discovery and computer vision.* Number 385 in . Wydawnictwo Politechniki Poznanskiej, Poznan University of Technology, Poznan, Poland, 2004.

[66] Krzysztof Krawiec. On relationships between semantic diversity, complexity and modularity of programming tasks. In Terry Soule, Anne Auger, Jason Moore, David Pelta, Christine Solnon, Mike Preuss, Alan Dorin, Yew-Soon Ong, Christian Blum, Dario Landa Silva, Frank Neumann, Tina Yu, Aniko Ekart, Will Browne, Tim Kovacs, Man-Leung Wong, Clara Pizzuti, Jon Rowe, Tobias Friedrich, Giovanni Squillero, Nicolas Bredeche, Stephen Smith, Alison Motsinger-Reif, Jose Lozano, Martin Pelikan, Silja Meyer-Nienberg, Christian Igel, Greg Hornby, Rene Doursat, Steve Gustafson, Gustavo Olague, Shin Yoo, John Clark, Gabriela Ochoa, Gisele Pappa, Fernando Lobo, Daniel Tauritz, Jurgen Branke, and Kalyanmoy Deb, editors, *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 783–790, Philadelphia, Pennsylvania, USA, 7-11 July 2012. ACM.

[67] Krzysztof Krawiec and Bir Bhanu. Visual learning by coevolutionary feature synthesis. *IEEE Transactions on System, Man, and Cybernetics – Part B*, 35(3):409–425, June 2005.

[68] Krzysztof Krawiec and Bir Bhanu. Visual learning by evolutionary and coevolutionary feature synthesis. *IEEE Transactions on Evolutionary Computation*, 11(5):635–650, October 2007.

[69] Krzysztof Krawiec and Pawel Lichocki. Approximating geometric crossover in semantic space. In Guenther Raidl, Franz Rothlauf, Giovanni Squillero, Rolf Drechsler, Thomas Stuetzle, Mauro Birattari, Clare Bates Congdon, Martin Middendorf, Christian Blum, Carlos Cotta, Peter Bosman, Joern Grahl, Joshua Knowles, David Corne, Hans-Georg Beyer, Ken Stanley, Julian F. Miller, Jano van Hemert, Tom Lenaerts, Marc Ebner, Jaume Bacardit, Michael O'Neill, Massimiliano Di Penta, Benjamin Doerr, Thomas Jansen, Riccardo Poli, and Enrique Alba, editors, *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 987–994, Montreal, 8-12 July 2009. ACM.

[70] Krzysztof Krawiec and Bartosz Wieloch. Analysis of semantic modularity for genetic programming. *Foundations of Computing and Decision Sciences*, 34(4):265–285, 2009.

[71] Krzysztof Krawiec and Bartosz Wieloch. Functional modularity for genetic programming. In Guenther Raidl, Franz Rothlauf, Giovanni Squillero, Rolf Drechsler, Thomas Stuetzle, Mauro Birattari, Clare Bates Congdon, Martin Middendorf, Christian Blum, Carlos Cotta, Peter Bosman, Joern Grahl, Joshua Knowles, David Corne, Hans-Georg Beyer, Ken Stanley, Julian F. Miller, Jano van Hemert, Tom Lenaerts, Marc Ebner, Jaume Bacardit, Michael O'Neill, Massimiliano Di Penta, Benjamin Doerr, Thomas Jansen, Riccardo Poli, and Enrique Alba, editors, *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 995–1002, Montreal, 8-12 July 2009. ACM.

[72] Krzysztof Krawiec and Bartosz Wieloch. Automatic generation and exploitation of related problems in genetic programming. In *IEEE Congress on Evolutionary Computation (CEC 2010)*, Barcelona, Spain, 18-23 July 2010. IEEE Press.

[73] W. B. Langdon and R. Poli. Fitness causes bloat: Mutation. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 37–48, Paris, 14-15 April 1998. Springer-Verlag.

[74] W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[75] William B. Langdon. Size fair and homologous tree genetic programming crossovers. *Genetic Programming and Evolvable Machines*, 1(1/2):95–119, April 2000.

[76] William B. Langdon and Wolfgang Banzhaf. Repeated sequences in linear genetic programming genomes. *Complex Systems*, 15(4):285–306, 2005.

[77] Hod Lipson. How to draw a straight line using a GP: Benchmarking evolutionary design against 19th century kinematic synthesis. In Maarten Keijzer, editor, *Late Breaking Papers at the 2004 Genetic and Evolutionary Computation Conference*, Seattle, Washington, USA, 26 July 2004.

[78] Jason Lohn, Gregory Hornby, and Derek Linden. An evolved antenna for deployment on nasa's space technology 5 mission. In Una-May O'Reilly, Tina Yu, Rick L. Riolo, and Bill Worzel, editors, *Genetic Programming Theory and Practice II*, chapter 18, pages 301–315. Springer, Ann Arbor, 13-15 May 2004.

[79] D. Loiacono, P.L. Lanzi, J. Togelius, E. Onieva, D.A. Pelta, M.V. Butz, T.D. Lošnneker, L. Cardamone, D. Perez, Y. Sáez, et al. The 2009 simulated car racing championship. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(2):131–147, 2010.

[80] Moshe Looks. *Competent Program Evolution.* Doctor of science, Washington University, St. Louis, USA, 11 December 2006.

[81] Moshe Looks. On the behavioral diversity of random programs. In Dirk Thierens, Hans-Georg Beyer, Josh Bongard, Jurgen Branke, John Andrew Clark, Dave Cliff, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Tim Kovacs, Sanjeev Kumar, Julian F. Miller, Jason Moore, Frank Neumann, Martin Pelikan, Riccardo Poli, Kumara Sastry, Kenneth Owen Stanley, Thomas Stutzle, Richard A Watson, and Ingo Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1636–1642, London, 7-11 July 2007. ACM Press.

[82] Sean Luke. Two fast tree-creation algorithms for genetic programming. *IEEE Transactions on Evolutionary Computation*, 4(3):274–283, September 2000.

[83] Sean Luke. *The ECJ Owner's Manual – A User Manual for the ECJ Evolutionary Computation Library*, zeroth edition, online version 0.2 edition, October 2010.

[84] Sean Luke and Liviu Panait. A survey and comparison of tree generation algorithms. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 81–88, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[85] Paul Massey, John A. Clark, and Susan Stepney. Human-competitive evolution of quantum computing artefacts by genetic programming. *Evolutionary Computation*, 14(1):21–40, Spring 2006. Best of GECCO 2004 special issue.

[86] James McDermott, Edgar Galvan-Lopez, and Michael O'Neill. A fine-grained view of phenotypes and locality in genetic programming. In Rick Riolo, Ekaterina Vladislavleva, and Jason H. Moore, editors, *Genetic Programming Theory and Practice IX*, Genetic and Evolutionary Computation, chapter 4, pages 57–76. Springer, Ann Arbor, USA, 12-14 May 2011.

[87] James McDermott, David R. White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Jaskowski, Krzysztof Krawiec, Robin Harper, Kenneth De Jong, and Una-May O'Reilly. Genetic programming needs better benchmarks. In Terry Soule, Anne Auger, Jason Moore, David Pelta, Christine Solnon, Mike Preuss, Alan Dorin, Yew-Soon Ong, Christian Blum, Dario Landa Silva, Frank Neumann, Tina Yu, Aniko Ekart, Will Browne, Tim Kovacs, Man-Leung Wong, Clara Pizzuti, Jon Rowe, Tobias Friedrich, Giovanni Squillero, Nicolas Bredeche, Stephen Smith, Alison Motsinger-Reif, Jose Lozano, Martin Pelikan, Silja

Meyer-Nienberg, Christian Igel, Greg Hornby, Rene Doursat, Steve Gustafson, Gustavo Olague, Shin Yoo, John Clark, Gabriela Ochoa, Gisele Pappa, Fernando Lobo, Daniel Tauritz, Jurgen Branke, and Kalyanmoy Deb, editors, *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 791–798, Philadelphia, Pennsylvania, USA, 7-11 July 2012. ACM.

[88] James McDermott, David R. White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Jaskowski, Krzysztof Krawiec, Robin Harper, Kenneth De Jong, and Una-May O'Reilly. Genetic programming needs better benchmarks. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, GECCO '12, pages 791–798, New York, NY, USA, 2012. ACM.

[89] Nicholas Freitag McPhee and Justin Darwin Miller. Accurate replication in genetic programming. In Larry J. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 303–309, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[90] Nicholas Freitag McPhee, Brian Ohs, and Tyler Hutchison. Semantic building blocks in genetic programming. In Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, Anna Isabel Esparcia Alcázar, Ivanoe De Falco, Antonio Della Cioppa, and Ernesto Tarantino, editors, *Genetic Programming*, volume 4971 of *LNCS*, pages 134–145. Springer, 2008.

[91] Zbigniew Michalewicz. *Genetic algorithms + data structures = evolution programs (2nd, extended ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1994.

[92] Julian F. Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1135–1142, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[93] Alberto Moraglio, Krzysztof Krawiec, and Colin G. Johnson. Geometric semantic genetic programming. In Carlos A. Coello Coello, Vincenzo Cutello, Kalyanmoy Deb, Stephanie Forrest, Giuseppe Nicosia, and Mario Pavone, editors, *Parallel Problem Solving from Nature, PPSN XII (part 1)*, volume 7491 of *Lecture Notes in Computer Science*, pages 21–31, Taormina, Italy, September 1-5 2012. Springer.

[94] Pablo Moscato. On evolution, search, optimization, genetic algorithms and martial

arts: Towards memetic algorithms. Technical Report C3P Report 826, California Institute of Technology, 1989.

[95] Stephen Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

[96] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In Alpesh Ranchordas and Helder Araújo, editors, *VISAPP (1)*, pages 331–340. INSTICC Press, 2009.

[97] Robert G. Newcombe. Interval estimation for the difference between independent proportions: comparison of eleven methods. *Statistics in Medicine*, 17(8):873–890, 1998.

[98] Quang Uy Nguyen, Xuan Hoai Nguyen, and Michael O'Neill. Semantic aware crossover for genetic programming: The case for real-valued function regression. In Leonardo Vanneschi, Steven Gustafson, Alberto Moraglio, Ivanoe De Falco, and Marc Ebner, editors, *Proceedings of the 12th European Conference on Genetic Programming, EuroGP 2009*, volume 5481 of *LNCS*, pages 292–302, Tuebingen, April 15-17 2009. Springer.

[99] Quang Uy Nguyen, Xuan Hoai Nguyen, and Michael O'Neill. Semantics based mutation in genetic programming: The case for real-valued symbolic regression. In R. Matousek and L. Nolle, editors, *15th International Conference on Soft Computing, Mendel'09*, pages 73–91, Brno, Czech Republic, June 24-26 2009.

[100] Peter Nordin and Wolfgang Banzhaf. Complexity compression and evolution. In Larry J. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 310–317, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[101] Una-May O'Reilly, Tina Yu, Rick L. Riolo, and Bill Worzel, editors. *Genetic Programming Theory and Practice II*, volume 8 of *Genetic Programming*, Ann Arbor, MI, USA, 13-15 May 2004. Springer.

[102] Michael Orlov and Moshe Sipper. Flight of the FINCH through the Java wilderness. *IEEE Transactions on Evolutionary Computation*, 15(2):166–182, April 2011.

[103] Cynthia B. Perez and Gustavo Olague. Learning invariant region descriptor operators with genetic programming and the F-measure. In *19th International Conference on Pattern Recognition (ICPR 2008)*, pages 1–4, Tampa, Florida, USA, December 8-11 2008.

[104] Riccardo Poli and William B. Langdon. Schema theory for genetic programming with one-point crossover and point mutation. *Evolutionary Computation*, 6(3):231–252, 1998.

[105] 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).

[106] Stefan Preble, Michal Lipson, and Hod Lipson. Two-dimensional photonic crystals designed by evolutionary algorithms. *Applied Physics Letters*, 86(6):061111+, 2005.

[107] Bill Rand, Alexandru-Adrian Tantar, Emilia Tantar, Peter A. N. Bosman, Nikhil Padhye, Aaron Baughman, Stefan Van Der Stock, Michael Perlitz, Steven M. Gustafson, Jonathan Jesneck, Gisele L. Pappa, John Woodward, Matthew R. Hyde, Jerry Swan, Stefan Wagner, Michael Affenzeller, Anne Auger, Alexandre Chotard, Nikolaus Hansen, Verena Heidrich-Meisner, Olaf Mersmann, Petr Posik, Mike Preuss, Forrest Stonedahl, Rick Riolo, Stephane Doncieux, Yaochu Jin, Jean-Baptiste Mouret, Daniele Loiacono, Albert Orriols-Puig, Ryan Urbanowicz, Kent McClymont, Ed Keedwell, Richard Everson, Jonathan Fieldsend, David Walker, F. Fernandez de Vega, Carlos Cotta, Steven Gustafson, Ekaterina Vladislavleva, Stephen L. Smith, Stefano Cagnoni, Robert M. Patton, Sherri Goings, Alison Motsinger-Reif, Katya Rodriguez, Christian Blum, Gabriela Ochoa, and Jason H. Moore, editors. *GECCO Companion '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, Philadelphia, Pennsylvania, USA, 7-11 July 2012.

[108] I. Rechenberg. *Evolutionsstrategie: optimierung technischer systeme nach prinzipien der biologischen evolution.* Frommann-Holzboog, 1973.

[109] Rick Riolo, Trent McConaghy, and Ekaterina Vladislavleva, editors. *Genetic Programming Theory and Practice VIII*, Genetic and Evolutionary Computation, Ann Arbor, USA, 20-22 May 2010. Springer.

[110] Rick Riolo, Ekaterina Vladislavleva, and Jason H. Moore, editors. *Genetic Programming Theory and Practice IX*, Genetic and Evolutionary Computation, Ann Arbor, USA, 12-14 May 2011. Springer.

[111] Justinian P. Rosca. Entropy-driven adaptive representation. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 23–32, Tahoe City, California, USA, 9 July 1995.

[112] Justinian P. Rosca and Dana H. Ballard. Genetic programming with adaptive representations. Technical Report TR 489, University of Rochester, Computer Science Department, Rochester, NY, USA, February 1994.

[113] Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, 3 April 2009.

[114] J.P. Shaffer. Modified sequentially rejective multiple test procedures. *Journal of the American Statistical Association*, pages 826–831, 1986.

[115] Robert E. Smith, Stephanie Forrest, and Alan S. Perelson. Population diversity in an immune system model: Implications for genetic search. In Darrell L. Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 153–165. Morgan Kaufmann, San Mateo, CA, 1993.

[116] Terence Soule and James A. Foster. Effects of code growth and parsimony pressure on populations in genetic programming. *Evolutionary Computation*, 6(4):293–309, Winter 1998.

[117] Lee Spector. Evolving control structures with automatically defined macros. In E. V. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 99–105, MIT, Cambridge, MA, USA, 10–12 November 1995. AAAI.

[118] Lee Spector. *Automatic Quantum Computer Programming: A Genetic Programming Approach*, volume 7 of *Genetic Programming*. Kluwer Academic Publishers, Boston/Dordrecht/New York/London, June 2004.

[119] Lee Spector. Evolving quantum computer algorithms. In Christian Blum, editor, *GECCO 2011 Late breaking abstracts*, pages 1081–1110, Dublin, Ireland, 12-16 July 2011. ACM.

[120] Lee Spector, David M. Clark, Ian Lindsay, Bradford Barr, and Jon Klein. Genetic programming for finite algebras. In Maarten Keijzer, Giuliano Antoniol, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Nikolaus Hansen, John H. Holmes, Gregory S. Hornby, Daniel Howard, James Kennedy, Sanjeev Kumar, Fernando G. Lobo, Julian Francis Miller, Jason Moore, Frank Neumann, Martin Pelikan, Jordan Pollack, Kumara Sastry, Kenneth Stanley, Adrian Stoica, El-Ghazali Talbi, and Ingo Wegener, editors, *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1291–1298, Atlanta, GA, USA, 12-16 July 2008. ACM.

[121] Rainer Storn and Kenneth Price. Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11:341–359, 1997. 10.1023/A:1008202821328.

[122] Lappoon R. Tang, Mary Elaine Califf, and Raymond J. Mooney. An experimental comparison of genetic programming and inductive logic programming on learning

recursive list functions. Technical Report AI 98-271, Artificial Intelligence Lab, University of Texas at Austin, USA, May 1998.

[123] Julian Togelius, Simon Lucas, Ho Duc Thang, Jonathan M. Garibaldi, Tomoharu Nakashima, Chin Hiong Tan, Itamar Elhanany, Shay Berant, Philip Hingston, Robert M. MacCallum, Thomas Haferlach, Aravind Gowrisankar, and Pete Burrow. The 2007 IEEE CEC simulated car racing competition. *Genetic Programming and Evolvable Machines*, 9(4):295–329, December 2008.

[124] Leonardo Trujillo and Gustavo Olague. Synthesis of interest point detectors through genetic programming. In Maarten Keijzer, Mike Cattolico, Dirk Arnold, Vladan Babovic, Christian Blum, Peter Bosman, Martin V. Butz, Carlos Coello Coello, Dipankar Dasgupta, Sevan G. Ficici, James Foster, Arturo Hernandez-Aguirre, Greg Hornby, Hod Lipson, Phil McMinn, Jason Moore, Guenther Raidl, Franz Rothlauf, Conor Ryan, and Dirk Thierens, editors, *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 1, pages 887–894, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[125] Nguyen Quang Uy, Nguyen Xuan Hoai, Michael O'Neill, R. I. McKay, and Edgar Galvan-Lopez. Semantically-based crossover in genetic programming: application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines*, 12(2):91–119, June 2011.

[126] Nguyen Quang Uy, Bob McKay, Michael O'Neill, and Nguyen Xuan Hoai. Self-adapting semantic sensitivities for semantic similarity based crossover. In *2010 IEEE World Congress on Computational Intelligence*, pages 4034–4040, Barcelona, Spain, 18-23 July 2010. IEEE Computational Intelligence Society, IEEE Press.

[127] Nguyen Quang Uy, Michael O'Neill, Xuan Hoai Nguyen, Bob McKay, and Edgar Galvan Lopez. Semantic similarity based crossover in GP: The case for real-valued function regression. In Pierre Collet, Nicolas Monmarche, Pierrick Legrand, Marc Schoenauer, and Evelyne Lutton, editors, *9th International Conference, Evolution Artificielle, EA 2009*, volume 5975 of *Lecture Notes in Computer Science*, pages 170–181, Strasbourg, France, October 26-28 2009. Springer. Revised Selected Papers.

[128] James Alfred Walker and Julian Francis Miller. Investigating the performance of module acquisition in cartesian genetic programming. In Hans-Georg Beyer, Una-May O'Reilly, Dirk V. Arnold, Wolfgang Banzhaf, Christian Blum, Eric W. Bonabeau, Erick Cantu-Paz, Dipankar Dasgupta, Kalyanmoy Deb, James A. Foster, Edwin D. de Jong, Hod Lipson, Xavier Llora, Spiros Mancoridis, Martin Pelikan, Guenther R. Raidl, Terence Soule, Andy M. Tyrrell, Jean-Paul Watson, and Eckart

Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1649–1656, Washington DC, USA, 25-29 June 2005. ACM Press.

[129] James Alfred Walker and Julian Francis Miller. The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 12(4):397–417, August 2008.

[130] R.A. Watson. *Compositional Evolution: Interdisciplinary Investigations in Evolvability, Modularity, and Symbiosis*. PhD thesis, Brandeis University, 2002.

[131] R.A. Watson, G.S. Hornby, and J.B. Pollack. Modeling building-block interdependency. In *Proc. Fifth International Conference Parallel Problem Solving from Nature (PPSN V)*, pages 97–106. Springer, 1998.

[132] Richard A. Watson and Jordan B. Pollack. Hierarchically consistent test problems for genetic algorithms: Summary and additional results. In Scott Brave and Annie S. Wu, editors, *Late Breaking Papers at the 1999 Genetic and Evolutionary Computation Conference*, pages 292–297, Orlando, Florida, USA, 13 July 1999.

[133] Richard A. Watson and Jordan B. Pollack. Modular interdependency in complex dynamical systems. *Artif. Life*, 11(4):445–458, 2005.

[134] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In Stephen Fickas, editor, *International Conference on Software Engineering (ICSE) 2009*, pages 364–374, Vancouver, May 16-24 2009.

[135] Pawel Widera, Jonathan M. Garibaldi, and Natalio Krasnogor. GP challenge: evolving energy function for protein structure prediction. *Genetic Programming and Evolvable Machines*, 11(1):61–88, March 2010.

[136] D. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[137] Man Leung Wong and Kwong Sak Leung. An adaptive inductive logic programming system using genetic programming. In John Robert McDonnell, Robert G. Reynolds, and David B. Fogel, editors, *Evolutionary Programming IV Proceedings of the Fourth Annual Conference on Evolutionary Programming*, pages 737–752, San Diego, CA, USA, 1-3 March 1995. MIT Press.

[138] Man Leung Wong and Kwong Sak Leung. Combining genetic programming and inductive logic programming using logic grammars. In *1995 IEEE Conference on Evolutionary Computation*, volume 2, pages 733–736, Perth, Australia, 29 November - 1 December 1995. IEEE Press.

[139] John R. Woodward and Ruibin Bai. Canonical representation genetic programming. In Lihong Xu, Erik D. Goodman, Guoliang Chen, Darrell Whitley, and Yongsheng Ding, editors, *GEC '09: Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, pages 585–592, Shanghai, China, June 12-14 2009. ACM.