POZNAN UNIVERSITY OF TECHNOLOGY

Tomasz P. Pawlak

Competent Algorithms for Geometric Semantic Genetic Programming

A dissertation submitted
to the Council of the Faculty of Computing
in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Supervisor: Krzysztof Krawiec, Ph. D., Dr. Habil, Associate Prof.

Poznań, Poland
2015

To my beloved wife, Paulina.

Abstract

Genetic Programming (GP) is a machine learning technique for automatic induction of computer programs from examples. The examples typically consist of two parts: program arguments – the input and a target program output. Both input and output may be expressed in terms of numeric or textual variables or even a conglomerate of the above. This problem formulation enables formalizing semantics of a program as a tuple of outputs it returns in effect of execution on the sample inputs. Use of semantics in GP gains an interest in research community, since the semantic methods developed so far in GP proved capable of achieving lower error, better generalization and smaller programs and quicker convergence to an optimum than the contemporary methods.

We embrace existing notions of semantics of program, semantic distance, semantic neutrality and effectiveness of genetic operators under the umbrella of common conceptual framework for Semantic Genetic Programming (SGP).

Then, we show that if a fitness function is a metric, the fitness landscape spanned over a space of all programs proxied by semantics becomes a cone with the optimal semantics in the apex. This provides justification for use of the recently developed Geometric Semantic Genetic Programming (GSGP), where geometric genetic operators utilize the conic shape of the landscape. We derive properties of progress and progress bounds of geometric operators for different combinations of fitness functions and semantic distances.

We present a comprehensive literature review of existing semantic methods, discuss their advantages and disadvantages and for each show how the defined properties apply to it.

Next, we propose an algorithm for backpropagating semantics trough program structure and *competent algorithms* for operators: population initialization, parent selection, mutation and crossover that are approximately geometric, effective and free of certain drawbacks of the existing geometric methods.

Then, we experimentally assess the proposed algorithms and compare them with the existing methods in terms of training set fitness, generalization on test set, probability of performing geometric and effective application, size of produced programs and computational costs. We use a suite of nine symbolic regression and nine Boolean program synthesis benchmarks. The analysis shows that the proposed algorithms achieve performance that is consistently better than that offered by other semantic GP methods for symbolic regression domain and not worse than the best other methods for Boolean domain.

Finally, we experimentally find the proportions of competent mutation and competent crossover that lead to the optimal results in the above range of benchmarks.

Contents

Table of symbols

The table below presents the symbols used in the thesis accompanied with references to places of their first use. Some of the symbols may appear in text with an optional sub- or superscript, with meaning depending on the context. If there is a generalization-specialization relation between two distinct notions, we use the same symbol for both of them, if not stated otherwise. Symbols not listed in this table are local and their meaning depends on the context.

Symbol	Meaning	Introduced in
[...]	Vector	
(...)	Tuple or open interval depending on the context (see below)	
{...}	Set or multiset	
(a, b)	Open interval from a to b	
$\langle a, b\rangle$	Closed interval from a to b	
$\Pr(\cdot)$	Probability	
$\mathcal{P}$	Genotype set, specifically: program set	Def. 2.1 on page 17
$\mathcal{S}$	Phenotype set, specifically: semantics set	Def. 2.1 on page 17
$s(\cdot)$	Genotype-phenotype mapping, specifically: semantic mapping	Def. 2.1 on page 17
p	Genotype, specifically: program	Def. 2.1 on page 17
s	Phenotype, specifically: semantics	Def. 2.1 on page 17
t	Target semantics	Def. 4.3 on page 44
P	Population of individuals	Def. 2.3 on page 18
$f(\cdot)$	Fitness function	Def. 2.2 on page 18
Π	Optimization problem	Def. 2.2 on page 18
Φ	Instruction set, Φ_t denotes terminals, Φ_n nonterminals	Def. 3.2 on page 29
I	Set of program inputs	Def. 3.3 on page 30
O	Set of program outputs	Def. 3.3 on page 30
in	Program input	Def. 4.1 on page 43
out	Program output; with asterisk refers to an optimal output	Def. 4.1 on page 43
F	List of fitness cases	Def. 4.1 on page 43
$d(\cdot, \cdot)$	Semantic distance	Def. 4.4 on page 44
$S(x, y)$	Segment between z and y	Def. 5.1 on page 47
$B(x, r)$	Ball of radius r centered in x	Def. 5.2 on page 48
$C(A)$	Convex hull of set A	Def. 5.3 on page 48

Acknowledgments

The work presented in this thesis was carried out between October 2011 and June 2015 in the Laboratory of Intelligent Decision Support System at Faculty of Computing at Poznan University of Technology. I would like to thank all the people that supported me during the work on this thesis. I would like to especially thank my supervisor, Krzysztof Krawiec for his encouragement, valuable comments, support and deserved criticism that enabled me to make this work even better and complete.

Furthermore, I would like to thank my family that supported me in and motivated to continuous work to achieve my goals.

Finally, I would like to thank Polish National Science Centre that supported the research presented in this thesis under grant number DEC-2012/07/N/ST6/03066.

Chapter 1

Introduction

1.1 Motivation

The notion of *automatic programming* can be traced back to 1950s [14, 15], when it was meant as a way of automated translation of program code expressed as a high-level specification into a low-level machine language. Nowadays we call a tool that carries out this above mentioned task a *compiler* and it became an inherent part of every software development process. In the meantime, the term ‘automatic programming’ was granted another meaning. The development of artificial intelligence methods led to emergence of a new field of research under the umbrella of automatic programming methods, called *inductive programming* — a study on methods that construct programs using incomplete knowledge. By ‘incomplete knowledge’ is meant that problem statement does not contain complete information on what the resulting program is supposed to do. More specifically, in program induction program behavior is constrained only by a set of examples of expected behavior. Thus, program induction task is twofold: it is expected to automatically synthesize a program that meets the given constrains, and that program is supposed to generalize beyond these constraints.

To the best of our knowledge the first works on inductive programming date back to the 1970s, when the first attempts to induce LISP programs from examples were made [171, 25]. After a decade, in the early 1990s this idea gave rise to *inductive logic programming* (ILP) [120], where the hypotheses (programs in logic) are constructed with use of the background knowledge.

In roughly the same time period, a different approach, called *genetic programming* (GP) [80, 141] was born. Genetic programming is an inspired by natural evolution methodology for automatic design of discrete structures from discrete entities. These structures are often, but not limited to, computer programs written in a chosen programming language. GP is not constrained to any programming paradigm or domain and in this sense is more general than ILP that induces logic programs only. To date GP is considered the most widespread, general and successful contemporary approach to inductive programming, with over 10000 scientific contributions in scientific materials [92]. The highlights of GP include automatically designed antenna with human-competitive characteristic that is used by NASA in its Space Technology mission 5 [99, 100, 67], automatic discovery of physical laws based on measurements acquired from the physical process [156] and automatic repair of bugs in software written by humans [53, 96].

GP is however not free from challenges. First of all, the relation between program structure (its source code) and its behavior is very complicated. A minute change of program code, e.g., a negation of a logical expression, can drastically change program behavior. On the other hand, a substantial syntactic change like, e.g., code refactoring, may have no influence on program behavior. This characteristic leads to difficulty in designing efficient genetic operators for GP. Secondly, many moves conducted by GP genetic operators are neutral, i.e., they do not affect program behavior.

This may be the case when the move occurs in a part of code that does not have any impact on program behavior. The third challenging issue in GP is undesirable growth of program code over iterations of the program induction algorithm. These three issues seem to be the main causes of GP's under-performance in some real-world applications, e.g., where sizeable programs need to be synthesized to solve a given task.

In almost three decades of GP research, multiple attempts have been made to cope with the abovementioned issues. Although many of them were successful, none of these issues were completely solved. Fortunately, the dawn of semantic-awareness in GP in recent years of research brings hope that the issues can be finally overcome. Semantics is a formal object that describes the behavior of a program. A wide range of recent works [20, 22, 21, 74, 178, 179, 130, 85, 115, 87, 86, 186, 133, 55, 131] show that GP equipped with semantic-aware routines performs better and can solve more difficult program induction problems than the ordinary variant of GP.

1.2 Scope of the thesis

The broad range of variants and applications of genetic programming inclines us to narrow our investigation to this paradigm of program induction (program synthesis) [80, 141]. We state the program induction problem in terms of supervised learning [152, Ch 18], where a set of examples is given, each consisting of program arguments and an output value. The examples are split into training and test parts, for induction and verification of generalization of a program, respectively. We assume that programs are represented as abstract syntax trees and do not use memory or storage persistent between executions, feature no side effects and are deterministic, i.e., the consecutive execution with immutable arguments lead to the same program outcome.

In this thesis we focus on a common conceptual framework for GP that embraces notions of *semantics*, *effectiveness* and *geometry* at each stage of evolutionary search. That is, we propose a consistent formalization of these notions for population initialization, parent selection and variation operators and focus on interactions between the parts of evolution that come from these formalisms.

The presented theoretical analysis is to a great extent general and independent on a domain of the evolved programs. The proposed methods are prepared for synthesis of imperative programs working with real and Boolean variables and assessed in problems of symbolic regression and induction of Boolean expressions. The methods can be easily adopted to an other programming paradigms, problem domains and learning schemes.

In the past semantics was used by either a variation operator [20, 22, 178, 179], population initialization [21, 75] or selection [55] to prevent breeding, creating or selecting two or more semantically equal programs, respectively, thus to improve effectiveness of the evolutionary search. Other works [114, 115, 83, 85, 87, 86, 186, 133] used semantics to assign a location in multidimensional space to a program and defined genetic operations on programs that have certain well-defined impact on this location. These semantic methods are often called geometric, since they are aware of geometry of the space. To our knowledge, there is only one work [131] that attempts to combine the geometric considerations with investigation of effectiveness. We also address this topic in this thesis.

1.3 Research goals

We formulate six research goals:

- Formulate a common conceptual and formal framework for semantic methods in genetic programming,

- Show that search in program space can be made more efficient when intermediated by semantic space,
- Define and prove properties of progress for geometric semantic genetic programming,
- Identify the sources of neutrality of genetic operations,
- Develop geometric and neutrality-resistant algorithms for population initialization, selection, crossover and mutation for genetic programming, so that they together enable solving difficult program induction problems quickly, reliably and accurately,
- Design, implement, and test the proposed algorithms in a common software framework,
- Verify the developed algorithms and compare them with existing techniques in a computational experiment.

1.4 Explanation of the title

In linguistic terms, the term 'competent' in title of this thesis may be considered controversial when attributed to an inanimate entity, which is genetic programming. That is, after Oxford Dictionary of English [170], the term 'competent' is reserved to a person having the necessary ability, knowledge or skill to do something, or to an entity being legal authority. However, genetic programming is an algorithm and the community of computer scientists often impersonates algorithms by e.g., saying that an algorithm *searches*, *finds*, *solves* etc. Therefore, it seems natural to also endow the algorithm with 'competence'.

Moreover, the term 'competent' has a long-standing presence in evolutionary algorithms. It has been introduced there by Goldberg [60], to characterize the class of *competent genetic algorithms* that solve hard problems, quickly, reliably, and accurately — hence overcoming the limitations of the extant technology. Later the term 'competent' was also used in the context of genetic programming by Sastry and Goldberg [155] with the same meaning.

In this study we adopt the terminology proposed by Goldberg and similarly to the Goldberg's definition, we identify the *competent algorithms for geometric semantic genetic programming* with algorithms that solve difficult program induction problems quickly, reliably and accurately.

1.5 Related papers

The achievements presented in this thesis are related to the following publications of the author:

- Tomasz P. Pawlak, Review and Comparative Analysis of Geometric Semantic Crossover Operators, *Genetic Programming and Evolvable Machines*, Springer, 2015 [134],
- Tomasz P. Pawlak, Bartosz Wieloch, Krzysztof Krawiec, Semantic Backpropagation for Designing Search Operators in Genetic Programming, *IEEE Transactions on Evolutionary Computation*, IEEE Press, 2014 [133],
- Tomasz P. Pawlak, Combining Semantically-Effective and Geometric Crossover Operators for Genetic Programming, PPSN XIII, Lecture Notes in Computer Science 8672:454–464, Springer, 2014 [131],
- Tomasz P. Pawlak, Krzysztof Krawiec, Guarantees of Progress for Geometric Semantic Genetic Programming, Workshop on Semantic Methods in Genetic Programming, PPSN XIII, 2014 [132],
- Krzysztof Krawiec, Tomasz P. Pawlak, Approximating Geometric Crossover by Semantic Backpropagation, GECCO '13, pp. 941–948, ACM, 2013 [86],

- Krzysztof Krawiec, Tomasz P. Pawlak, Locally Geometric Semantic Crossover: A Study on the Roles of Semantics and Homology in Recombination Operators, *Genetic Programming and Evolvable Machines* 14(1):31–63, Springer, 2013 [87],
- Krzysztof Krawiec, Tomasz P. Pawlak, Quantitative Analysis of Locally Geometric Semantic Crossover, PPSN XII, Lecture Notes in Computer Science 7491:397–406 [85],
- Krzysztof Krawiec, Tomasz P. Pawlak, Locally Geometric Semantic Crossover, GECCO '12, pp. 1487–1488, ACM, 2012 [84].

1.6 Organization of the thesis

Chapter 2 introduces assumptions, general scheme and variants of evolutionary algorithms. Chapter 3 describes details of genetic programming in its canonical form and common contemporary modifications.

In Chapter 4 we introduce the notions of semantics, neutrality and effectiveness for operators involved in GP search (initialization, selection, and variation operators). In Chapter 5 we present the recently introduced notion of geometry in semantic GP and the theory behind it, including the characteristics of geometric operators with respect to a wide range of fitness functions.

Chapter 6 contains a comprehensive survey of existing semantic methods in genetic programming.

Chapter 7 is devoted to complete description of Competent Algorithms for operators of GP that combine the desirable features of effective and geometric methods.

Chapter 8 poses research questions to be answered in experiments and presents the general description of them. The outcomes of extensive computational experiments are presented in two subsequent chapters: Chapter 9 concerns assessment of genetic operators in isolation, and Chapter 10 consists of verification how the operators act together in GP search.

Finally, Chapter 11 concludes the thesis and provides suggestions for future work.

Chapter 2

Evolutionary Algorithms at a glance

The purpose of this chapter is to introduce the key concepts of evolutionary algorithms. We start from the brief introduction to evolutionary algorithms, discuss general assumptions, and present a scheme of evolutionary algorithm. Then we review the existing variants of evolutionary algorithms. The chapter is concluded by a review of famous achievements of evolutionary algorithms.

2.1 Introduction

Evolutionary algorithm (EA) [47] is an optimization technique inspired by natural evolution. For this reason, the general scheme of the algorithm as well as nomenclature of EAs are to a great extent borrowed from their biological counterparts. In the following we bring these notions and use them consistently across this study. The beginnings of EAs can be traced back to the late 1960s and early 1970s to the works of Fogel *et al.* [52], Holland [65, 66], and Rechenberg [143]. Although their works were independent, they share many common features, discussed in Subsections 2.1.1–2.1.6, and as such were later united under the common name of evolutionary algorithm. What is worth noting is that those features are inherent to the evolutionary algorithms – no other optimization technique incorporates *all* of them – which distinguishes EAs as a separate class of optimization algorithms.

2.1.1 Genotype-phenotype separation

EAs are designed to solve optimization problems, where the objective is to find a solution that optimizes given objective function. The technical *representation* of a solution, known as *genotype*, is logically separated from the use or effects of its application to the problem, known as *phenotype*. In this sense the genotype encodes the corresponding phenotype, like in nature DNA encodes an actual look and operation of a human. The distinction is made to conveniently manipulate a simple structure that describes a more complicated model of a solution to the problem, e.g., a genotype being a string of bits may represent integer number on the phenotypic level, similarly a genotype consisting of a matrix of numbers may encode connections in a graph in the phenotype. An atomic component of genotype (e.g., bit or number, respectively) is called *gene*. Formally, we define:

Definition 2.1. Let $\mathcal{P}$ be a genotype set, $\mathcal{S}$ be a phenotype set, and $s : \mathcal{P} \to \mathcal{S}$ be a genotype-phenotype mapping (a function).

Note that we do not impose any restrictions on $\mathcal{P}$ and $\mathcal{S}$, thus they can contain any formal objects. However there exists a mapping function $s(\cdot)$ that interlinks objects in these two sets. There are also no specific restrictions for $s(\cdot)$, thus multiple genotypes in $\mathcal{P}$ can be assigned to a single phenotype in $\mathcal{S}$ and there may be phenotypes in $\mathcal{S}$, that do not have a counterpart in $\mathcal{P}$. If $s(\cdot)$ is an identity function, the separation of genotypes and phenotypes vanishes.

Thanks to the separation, the same genotypes can represent different phenotypes depending on interpretation and problem statement. Since that the standard variants of EAs cope well with distinct real world problems and the algorithm itself abstracts from the particular meaning of the solution. Thus there is no need for major modifications to the algorithm to adapt it to a certain problem.

2.1.2 Fitness function

To navigate the quality of solutions, we introduce *fitness function* that assigns a numeric value, called *fitness*, to each solution, called *individual.* By biological analogy fitness indicates how 'fit' an individual is to a given environment, like in EA fitness indicates what 'good' and 'bad' solutions are, hence what the improvement means. Therefore fitness function actually defines the problem to be solved. In the field of the classical optimization the fitness function would be called objective function and the problem to be solved, an optimization problem. Formally:

Definition 2.2. Let $f : \mathcal{S} \to \mathbb{R}$ be a fitness function. $\Pi = (\mathcal{P}, f)$ is an optimization problem and $p^* = \arg\max_{p\in\mathcal{P}} f(s(p))$ is an optimal solution to the problem Π.

Note that the domain of fitness function is phenotype set, while the optimization problem involves its representation — genotype set. Since the genotype-phenotype mapping acts as a link between these two sets, fitness function can be used to indirectly assign a fitness to a genotype. Hence in the following we allow for a shorthand notation $f(p) \equiv f(s(p))$. The fitness function is to be maximized or minimized depending on particular problem statement, note however that a minimized function can be easily turned into maximized one by multiplying it by -1 and vice versa. The optimal solution p^* to the problem is a solution p for which the corresponding phenotype achieves the highest value of the fitness function. Note that there may be more than one such a solution.

2.1.3 Population of solutions

The algorithm operates on a collection of solutions, in EA terminology called *population of individuals.* The search space is sampled by the population and explored in parallel, possibly in different directions during evolutionary search. This lessens the risk of algorithm getting stuck in local optima , in contrast to e.g., local search heuristics. Formally:

Definition 2.3. Population of individuals P is multiset of genotypes from $\mathcal{P}$, i.e., $P = (\mathcal{P}, m)$, where $m : \mathcal{P} \to \mathbb{N}_{\geq 1}$ is a function that maps genotype into a number of its occurrences in P.

For convenience we allow in the following the abuse of notation by using the set-like notation for both contents of the multiset, i.e., $\{a, a, b\}$ and operators, e.g., $\subset, \subseteq, \supset, \supseteq, \in$, jointly to sets and multisets.[1]

The population P is usually constrained by a given size, that is kept constant during the evolutionary run. However due to evolutionary search may obtain the same solution in a different ways, the number of distinct individuals in population, often called *diversity*, may vary in time. The diversity is usually calculated by counting up a distinct genotypes in the population (genotypic diversity), or unique phenotypes obtained using genotype-phenotype mapping from the population (phenotypic diversity) or distinct fitness values (fitness-based diversity).

[1] A set can be considered as a multiset having $m : \mathcal{P} \to \{1\}$.

2.1.4 Generational characteristic

The algorithm creates a new individuals from the existing ones in a population by mixing together given two or more genotypes or randomly modifying a given single genotype. In this scheme, by biological analogy the individuals chosen to be mixed or modified are called *parents*, and the created individuals are called *offspring*. This process is repeated iteratively and the created offspring belong to a new *generation* that replaces generation of their parents.

2.1.5 Stochasticism

Evolutionary algorithm is a stochastic optimization technique, i.e., EA involves random variables that influence outcomes of genetic operators and affect selection of individuals to be modified or to be promoted to the next generation. The probabilities of certain actions are parameters of the algorithm.

2.1.6 Metaheuristic

Since many real-world problems are NP-hard, there do not exist exact polynomial-time algorithms for them. Thus an attempt to solve such a problems for a big real-world instances becomes futile. This characteristic is the main motivation for development of heuristics, i.e., algorithms that are likely to produce well-performing (yet not necessarily optimal) solutions in polynomial time. Evolutionary algorithm is just a one of such algorithms, or more specifically a meta-heuristics, i.e., a general (i.e., not problem-specific) 'meta-algorithm'. The characteristic feature of EAs is their ability to maintain both exploration and exploitation. By 'exploration' we mean search in distant, distributed parts of the solution space to find regions of good solutions, and by 'exploitation' we mean the search condensed in a small region of good solutions to find a local optimum of this region.

2.2 The Evolutionary Algorithm

The general scheme for all EAs is presented in Algorithm 2.1. It starts from initialization of the population P. Then each solution is evaluated. Next, the main part of the algorithm begins: *the evolutionary loop*. In the loop a new population P' is being filled by the offspring bred using selected parents from the current population P. Breeding is done by applying variation operations to the selected parents. When population P' gets filled, replaces the current one (P), and each individual in it is a subject to the evaluation. The evolutionary loop is repeated until a termination condition is satisfied. Finally the best solution found in the population is returned by the algorithm. Alternatively we can store the best solution found across entire evolutionary run and return it, however for brevity we omit this extension in Algorithm 2.1.

Instead of replacement of an entire population by another population at the end of evolutionary loop, only a part of the population may be replaced by a new one. This substitution model is called *steady-state* [183]. In steady-state model in each generation μ individuals are selected from current population P and become parents to λ offspring. The next generation population P' consists then of individuals from the old one except λ individuals that are replaced by λ offspring to keep the population size constant across generations, i.e., $P' = P \backslash \{p_1, p_2, ..., p_\lambda\} \cup \{p'_1, p'_2, ..., p'_\lambda\}$.

Two alternative substitution models were proposed by Rechenberg [143], in (μ, λ) model μ parents are selected[2] from P to produce λ offspring that constitute the next population P' (possibly each parent breeds multiple times). In $(\mu + \lambda)$ model μ parents are added together with λ offspring

[2] Rechenberg's model originally selects μ the best individuals in P, however other selection mechanisms (cf. Section 3.5) are possible

Algorithm 2.1: Pseudocode of the general evolutionary algorithm scheme.

```
function EvolutionaryAlgorithm
    $P \leftarrow$InitializePopulation()
    Evaluate($P$)
    while $\neg$TerminationCondition do    :: Evolutionary loop
        $P' \leftarrow \emptyset$
        while $|P'| \neq |P|$ do
            $\{p_1, p_2, ...\} \leftarrow$ Select($P$)
            $\{p'_1, p'_2, ...\} \leftarrow$ ApplyVariationOperations($\{p_1, p_2, ...\}$)
            $P' \leftarrow P' \cup \{p'_1, p'_2, ...\}$
        $P \leftarrow P'$
        Evaluate($P$)
    return $\arg\max_{p \in P} f(p)$
```

to the next population P'. Note the former model assumes that population size is λ and the latter one that $\mu + \lambda$.

The operations on individuals EA performs by means of random functions, called initialization, selection and variation operators, respectively. Each realization of a random function may result in a different function. Below we discuss all of the distinguished parts of EA, including the operators and evaluation and termination condition.

2.2.1 Initialization

At the beginning EA prepares a population of solutions using an initialization operator, defined below:

Definition 2.4. Initialization operator is a nullary random function that returns programs, i.e., $OP : \emptyset \to \mathcal{P}$.

Notice by definition the initialization operator produces a single individual at once, hence the operator is to be applied multiple times to fill up the entire population.

2.2.2 Evaluation

Evaluation phase consists of application of fitness function $f(\cdot)$ to each individual in the population.

The fitness function typically calculates a kind of an error, a cost or profit of application of a solution to a problem. For instance common fitness functions are: absolute error, squared error or relative error w.r.t. an expected output, total cost of running a schedule, amount of used material or a number of games won by an individual etc.

The evaluation phase is often the most computationally expensive part of the algorithm, since it often requires running a simulation, computing a complex physical model or playing large amount of games. From this reason it is quite common to avoid reevaluation of an individual by caching fitness values for particular genotypes or use a fast approximate of ‘real’ fitness function, e.g., function running a physical model with less precision.

2.2.3 Selection

The role of selection is to choose individuals from the current population that take part in genetic operations and produce offspring. In EA selection chooses *good* individuals, i.e., individuals that are not necessary the best in the population. Actually the systematic choice of the best individual can quickly lead to a loss of diversity and convergence of entire population to a single point of search space, consequently leading to unsatisfactory results of the optimization. The preference of good individuals to the bad ones, called *selection pressure*, depends on the particular selection method and may be a subject to be tuned. Formally we define:

Definition 2.5. Selection operator is a random function, such that $OP : \mathcal{P}^{|P|} \to \mathcal{P}$.

That is, a selection operator takes as an argument the entire population and returns a single genotype from it. However in some applications there is a need to select two or more somehow related genotypes, for such a case we define after Miller [110] a mate selection operator and extend it to the case of $n \geq 2$ mates:

Definition 2.6. n-mate selection operator is a random function, such that $OP : \mathcal{P}^{n-1} \times P^{|P|} \to P$.

It distinguishes itself from a simple selection operator in that it takes $n - 1$ extra arguments — a set of *mate genotypes* (other candidate solutions) with respect to which the related nth genotype is to be selected from the population.

Below we briefly introduce the most common selection operators. An interested reader is referred to [47, Ch 3] for discussion on pros and cons of these operators.

Definition 2.7. Fitness proportional selection is selection operator $OP : P^{|P|} \to P$ which, when applied to population P, chooses an individual $p \in P$ with probability proportional to its fitness, i.e., $^{f(p)}/_{\sum_{p' \in P} f(p')}$.

Definition 2.8. μ-tournament selection (μ-TS) [109], where $\mu : \mu \in \mathbb{N}_{\geq 1}, \mu \leq |P|$ is tournament size, is selection operator $OP : P^{|P|} \to P$ that randomly draws a set C of μ candidate individuals from population and returns the best candidate in C.

The higher the value of μ is, the higher the probability of choosing good individuals from the population. Hence μ controls selection pressure of the operator. In practical applications μ is rather small, typically in the range $\langle 2, 7 \rangle$ [47, 80, 101]. Tournament selection is insensitive to absolute values of fitness , hence it is the preferred operator in many applications.

2.2.4 Variation operators

By a variation operator (line 8 of Algorithm 2.1) we mean any operator that produces new individuals from the existing ones. Arguments of the operator are typically called *parents* and the returned value is referred to as *offspring*. Typically in EA we distinguish three variation operators: reproduction (cloning), mutation and crossover, although literature abounds with many other operators.

Definition 2.9. Reproduction is identity function, i.e., $OP : \mathcal{P} \to \mathcal{P}$.

In other words reproduction returns a solution given as the argument without any changes in it. Reproduction together with selection of n best individuals in population are commonly used to realize elitism, i.e., a mechanism that moves n best individuals from the current population to the next one. The intent behind this mechanism is to guarantee the survival of the best so far solution in the population. Reproduction is sometimes also used when an other variation operator fails and a technical realization of EA requires an offspring to be provided anyway.

Definition 2.10. Mutation is an unary random function that, given a single parent solution, produces an offspring , i.e., $OP : \mathcal{P} \to \mathcal{P}$.

Since mutation takes a single solution and modifies it to create a new one, its function is similar to the function of search operators employed in local search algorithms. In EAs we commonly assume that phenotype of the produced offspring is similar to phenotype of its parent, which usually is achieved by sharing common fragments of genotype. This idea can be extended to a two-parent operator that creates a single offspring:

Definition 2.11. Crossover is a binary random function that, given two parent solutions, produces an offspring that is a mixture of both parents, i.e., $OP : \mathcal{P} \times \mathcal{P} \to \mathcal{P}$.

Crossover gives the evolutionary algorithm ability to combine a desirable parts of the given parent solutions and drop the undesirable ones to create an offspring being superior to its parents. However the most variants of crossovers are stochastic and unaware of a quality of a solution. Only thanks to the selection pressure the good offspring survive and the bad ones extinct in the population over generations, contributing so to the progress of evolution. Note that the algorithms' ability to combine features of solutions is quite unique, and uncommon for a wide range of optimization techniques other than EA. However even in EAs are a few exceptions (discussed later in Section 2.3).

In general a higher-arity variation operators are mathematically feasible and easy to technically realize, however since such a breeding schematics are not found in the nature, they are barely used in practice. Nevertheless some studies indicate positive effects of multi-parent breeding on the progress of optimization [48, 174, 35].

The particular algorithms of discussed operators are representation-dependent, and as such will be discussed shortly in respective sections.

2.2.5 Termination condition

EA is metaheuristic, and as such it does not feature a strict worst-case time complexity or any other limit on computation costs. There is only a termination condition that determines when to stop the algorithm.

There are multiple reasons for which the optimization is to be terminated. First possibility is when the global optimum is found — in some applications it is quite easy to determine if a solution is optimal, e.g., when fitness function attains its upper or lower bound. In others the optimality of a solution is not known, e.g., when evolving game strategies where the strategy can be evaluated only on a set of opponents. However even in the former case, the algorithm is not guaranteed to reach the optimum, thus some limits on execution time have to be imposed. These limits typically involve number of generations, computation time, or number of evaluations of a fitness function. Sometimes, the termination condition also features indicators of stagnation, e.g., a loss of diversity dropping below a certain threshold, or no improvement of fitness of the best found solution for a few generations.

2.3 Variants of Evolutionary Algorithms

The purpose of this section is to briefly review the existing variants of evolutionary algorithms in their canonical or standard forms, thus excluding the uncommon or exotic derivatives.

2.3.1 Evolutionary Programming

Historically the first attempt to simulate the natural evolution to create an artificial intelligence system was made in 1960s by Fogel *et al.* [52]. Fogel was probably the first who noticed that instead of simulating a particular achievements of Nature, like e.g., neural networks, it can be more beneficial to simulate the force that created those achievements — natural evolution. The proposed technique was named *evolutionary programming* (EP). The key idea is to create a system that is able to adapt its behavior to meet some specific goals in a given changing environment and generalizes its characteristics among different states of the environment.

The original formulation of evolutionary programming involves a pragmatic approach to adapt a representation of the genotype to a given problem. However in many real-world applications it boils down to optimization of function of the form $g : \mathbb{R}^n \to \mathbb{R}$, thus a typical adopted representation is a vector of numbers, i.e., $[x_1, x_2, ..., x_n] \in \mathbb{R}^n$. Nowadays we use an extra vector of adaptation coefficients for mutation operator, thus the actual genotype becomes extended to a vector of form $[x_1, x_2, ..., x_n, \sigma_1, \sigma_2, ..., \sigma_n] \in \mathbb{R}^{2n}$. This form of EP is called meta-EP [50] and became *de facto* standard in contemporary EP.

Since the genotype representation is supposed to be problem-dependent, the variation operators must be adapted to it. For the sake of brevity, we focus in the following on operators working on real-valued vectors only. In the original formulation of EP, there is no crossover and, as it was later verified, it does not improve the algorithm' performance significantly [51]. Hence mutation is the only operator involved in search. EP mutation is an unary operator that, given a parent genotype in form $[x_1, x_2, ..., x_n, \sigma_1, \sigma_2, ..., \sigma_n]$, produces an offspring genotype $[x'_1, x'_2, ..., x'_n, \sigma'_1, \sigma'_2, ..., \sigma'_n]$ according to formula:

$$\sigma'_i = \sigma_i(1 + \alpha N(0, 1)) \tag{2.1}$$

$$x'_i = x_i + \sigma'_i N(0, 1) \tag{2.2}$$

where $N(0, 1)$ denotes a random variable with Gaussian distribution with average 0 and standard deviation 1, and $\alpha \approx 0.2$ [47, Ch 5]. To prevent neutral mutations, i.e., mutations that do not change solution and lead to ineffective search, a rule of thumb is recommended that prevents too small values of σ'_i, i.e., $\sigma'_i = \max\{\sigma_i(1 + \alpha N(0, 1)), \epsilon\}$, where ϵ is small value greater than 0.

2.3.2 Evolutionary Strategies

The other method, designed specifically for optimization of real function of real variables, i.e., $g : \mathbb{R}^n \to \mathbb{R}$, was proposed by Rechenberg in 1970s [143]. The method is named *evolutionary strategies* (ES) and, similarly to Fogel's evolutionary programming, it also involves a real-valued vector as a genotype and a Gaussian-like mutation as the main variation operator. However in contrary to evolutionary programming, it also engages crossover.

Genotype in ES is represented by a vector of real numbers, consisting of three parts: the objective variables $\overrightarrow{x}$ to be optimized, standard deviations $\overrightarrow{\sigma}$, and rotation angles $\overrightarrow{\alpha}$ i.e., $[\overrightarrow{x}, \overrightarrow{\sigma}, \overrightarrow{\alpha}] = [x_1, x_2, ..., x_n, \sigma_1, \sigma_2, ..., \sigma_n, \alpha_1, \alpha_2, ..., \alpha_n] \in \mathbb{R}^{3n}$.

Over the years, correlated mutation operator [150] became a standard in ES, thus we introduce it below. Correlated mutation performs a multimodal Gaussian translation of the given point $\overrightarrow{x}$ of genotype in the search space, taking into account different dynamics of its dimensions and possible rotations of axes. Formally given genotype of form $[\overrightarrow{x}, \overrightarrow{\sigma}, \overrightarrow{\alpha}]$ the algorithm outputs a new genotype $[\overrightarrow{x}', \overrightarrow{\sigma}', \overrightarrow{\alpha}']$ in three steps:

$$\sigma'_i = \sigma_i e^{\tau' N(0,1) + \tau N_i(0,1)} \tag{2.3}$$

$$\alpha'_i = (\alpha_i + \beta N_i(0, 1)) \mod 2\pi - \pi \tag{2.4}$$

$$\overrightarrow{x}' = \overrightarrow{x} + R(\overrightarrow{\alpha}') \cdot \operatorname{diag}(\overrightarrow{\sigma}') \cdot \overrightarrow{N}(\overrightarrow{0}, I) \tag{2.5}$$

where $\tau \propto 1/\sqrt{2/\sqrt{n}}$, $\tau' \propto 1/\sqrt{2n}$, and β is rotation angle change coefficient, typically 5° [47, Ch 4]. $N(\cdot, \cdot)$ is value drawn from normal distribution; when appears with variable index i, it is drawn separately for each i, when appears with $\overrightarrow{\cdot}$, it is a random vector drawn from multimodal Gaussian distribution. $R(\cdot)$ is rotation matrix of given angles, $\operatorname{diag}(\cdot)$ is diagonal matrix, and I is identity matrix. The ranges of α'_is are typically bound to $\langle -\pi, \pi)$ by calculating modulo this interval in Eq. (2.4).

Crossover is much simpler operator. There are two schemes: intermediary and discrete. Given two parent genotypes (denoted by a superscript index), a crossover produces a single offspring by applying the following formula to each component i of genotype separately:

$$x'_i = \begin{cases} \frac{1}{2}(x_i^1 + x_i^2), & \text{intermediary} \\ x_i^1 \text{ or } x_i^2, \text{ chosen uniformly} & \text{discrete} \end{cases}$$

According to [47, Ch 4], these two schemes are used together. The intermediary one is recommended for crossing over of $\overrightarrow{\sigma}$ and $\overrightarrow{\alpha}$ parameters, and the discrete one for the variable part $\overrightarrow{x}$. The discrete scheme supports exploration of different combinations of components of a vector, and intermediate scheme assures a more cautious adaptation of parameters.

Evolutionary strategies were the first evolutionary optimization technique that featured self-adaptation of parameters [47, Ch 4], realized there by tuning of σ_is in Eq. (2.3). The typical behavior of ES is that the standard deviations σ_is decrease over the course of evolution, particularly when the population converges to the optimum.

2.3.3 Genetic Algorithms

Genetic algorithm (GA) introduced by Holland [65, 66] in 1970s is the third and the most recognizable variant of evolutionary algorithm. The initial works of Holland concerned the strategies of adaptation in nature and in artificial systems, however most of them were actually focused on function optimization.

The canonical and most common representation of genotype in GA is a string of bits of constant length, i.e., $[b_1, b_2, ..., b_n] \in \{0, 1\}^n$. Although such a representation is very general and natural to those familiar with computer science, it is low-level and conceptually distant from the phenotypic features it is supposed to encode. For this reason, a genotype-phenotype mapping in GA is supposed to ensure that all solutions can be produced, as well as that all genotypes produce a valid phenotypes. In practice, these assumptions are often softened, and the mapping is allowed to sometimes produce an invalid phenotype, however in such a case typical workaround is to penalize such a solution during evaluation phase. The other challenge is *epistasis* — the interactions between distant and apparently unrelated bits that separately lead to different phenotype features than together. This is, among others, the case for coding of integer numbers, where one bit determines a sign of an integer and other bits determine absolute value. Other issue comes from different significance of particular bits that is dependent on bits' positions, and a minute change of genotype can reflect in a major change of phenotype. Consider four-bit integer numbers 0111_2 (7) and 1000_2 (8), in this case change of phenotype value just by one results in complete inversion of the entire genotype. On the other hand change of 0111_2 (7) to 0110_2 (6) requires only a flip of a single bit. The discussed problem can be to some extent handled by introducing Gray code [62], a coding system that assigns to a consecutive integer numbers a binary strings that differ only by a single bit, however for other genotype-phenotype mappings it may be not so obvious.

Other representations of genotype, like integer vectors or permutations are also common in the literature, however in the following we focus only on the standard form of GA.

There are three common crossover operators in GA: each of them takes as arguments two parent genotypes in form $[b_1^1, b_2^1, ..., b_n^1]$ and $[b_1^2, b_2^2, ..., b_n^2]$ to produce an offspring in form $[b'_1, b'_2, ..., b'_n]$.

Let us start from the *one-point crossover*. It starts with drawing at random an integer k from the range $\langle 1, n-1 \rangle$, then it splits parent genotype vectors at the drawn location and exchanges the tail of the first parent with the tail of the second one, producing so the offspring. Offspring's genotype is $[b_1^1, b_2^1, ..., b_k^1, b_{k+1}^2, b_{k+2}^2, ..., b_n^2]$,[3] where Γth bit is the last one taken from the first parent and $k+1$ bit is the first bit taken from the second parent. The split location is known as crossover point.

[3] Blue color denotes part of genotype taken from the second parent.

A natural generalization of one-point crossover is the *m-point crossover.* In an m-point crossover, a vector $\overrightarrow{k} = [k_1, k_2, ..., k_m]$ of crossover points is drawn from the range $\langle 1, n-1 \rangle$, where $1 \leq m \leq n-1$ and $\forall_{i,j:i<j} : k_i < k_j$. Genotypes are split in m points, and then even-numbered subvector separated by split points is taken from the second parent and replaces the corresponding one in the first parent, producing so the offspring. Thus the offspring genotype is $[b^1_1, b^1_2, ..., b^1_{k_1}, b^2_{k_1+1}, b^2_{k_1+2}, ..., b^2_{k_2}, b^1_{k_2+1}, b^1_{k_2+2}, ..., b^1_n]$.[3]

The other approach of iterating over parents' genotypes in parallel, and at each bit drawing randomly a bit from one of the parents is known as *uniform crossover.* The probabilities of choosing a particular parent's bit are usually equal for both of them, and the offspring's genotype takes a form $[b^1_1, b^2_2, b^2_3, b^1_4, ..., b^1_n]$.[3]

Note that all the presented crossover operators clearly waste the half of the genes provided by the parents, that may be used to produce a second offspring. To be consistent, we assume that second offspring can be produced by running crossover again with parents swapped and the same crossover points. Production of two offspring is actually the case for most real-world applications of GA.

Note that the choice of a particular crossover operator imposes a certain positional bias on the search. For instance one- and m-point crossovers tend to keep subsequences of genes together, while the uniform crossover does not. If for a particular genotype-phenotype mapping, a conglomerate of genes contributes to a single phenotype feature, then a natural choice would be one- or m-point crossover, since the uniform one is unlikely to preserve the parts of genotypes that have meaningful interpretation. In turn uniform crossover allows us to easily control the distributional bias, i.e., percentage of bits taken from each parent.

GA mutation, given a single parent genotype $[b_1, b_2, ..., b_n]$ produces an offspring genotype $[b'_1, b'_2, ..., b'_n]$ by iterating over the parent's bits and flipping each of them with certain probability. Thus offspring's genotype may take the form $[b_1, \neg b_2, b_3, ..., \neg b_n]$. The probability of negating a bit is rather small. The typical approach is to negate on average a single bit per genotype [47, Ch 3].

Schema Theorem and Building Blocks Hypothesis

The theorem on the improvement of population of solutions and progress in genetic algorithms is known as schema theorem [65, 66]. For brevity we narrow our considerations to GA using strings of bits, one-point crossover, bitwise mutation and fitness proportionate selection (cf. Definition 2.7). We discuss only the most important conclusions, for a more thorough discussion and proofs please refer to [66].

Definition 2.12. Schema (plural: schemata) H=$[h_1, h_2, ..., h_n] \in \{0, 1, \#\}^n$ is a hyperplane, where 0 and 1 are fixed values, while # means 'don't care', i.e., 0 or 1. Genotype $[b'_1, b'_2, ..., b'_n] \in \{0, 1\}^n$ matches schema H if the fixed values of H are equal to corresponding values in the genotype, i.e., $\forall_{i=1..n} b_i = h_i$.

Definition 2.13. The order $o(H)$ of a schema H is the number of fixed values in H and the defining length $l(H)$ of a schema H is the distance between the first and the last fixed positions in H.

Theorem 2.14. *The proportion $m(H, g)$ of solutions matching schema H at subsequent generations g is:*

$$m(H, g+1) \geq m(H, g) \cdot \frac{f(H)}{\overline{f}} \cdot \left(1 - \Pr(C) \cdot \frac{l(H)}{n-1}\right) \cdot (1 - \Pr(M) \cdot o(H))$$

where $f(H)$ is fitness of schema H calculated as an average of fitness of all matched genotypes, $\overline{f}$ is average fitness in population, $\Pr(C)$ *is probability of crossover,* $\Pr(M)$ *is probability of mutation and n is length of the genotype.*

The conclusion of Theorem 2.14 is that a number of schemata that are fit above average, short and low-ordered increase in population over generations. GA implicitly identifies in population such schemata and juxtaposes them during the course of evolution, producing so better and better schemata. This explanation is commonly referred in the literature to as *building blocks hypothesis,* and the short, low-order and highly fit schemata are called *building blocks* [59].

2.3.4 Genetic Programming

Genetic programming (GP) is the fourth main variant of evolutionary algorithm, intended for induction of programs. Since genetic programming is the main concern of this study, we thoroughly discuss it in Chapter 3. Then in Chapter 4 we describe the novel, semantic form of genetic programming and in Chapter 5 we discuss its advanced, geometric variant.

2.4 Estimation of Distribution Algorithms

A common property of all EA variants presented in Section 2.3 is explicit use of a population of individuals and variation operators that operate on the individuals by mixing and modifying them to contribute to the search progress. The *Estimation of Distribution Algorithms* (EDA) [94] do not involve particular individuals nor variation operators, and use an estimated probability distribution of good solutions as a model of population. This estimation is obtained by means of e.g., tuning parameters of multimodal Gaussian distribution, induction of Bayesian networks, or other probabilistic formalisms. Hence the formal object being evolved consists only of the parameters of the distribution, rather than of actual solutions (i.e., individuals). A sample of actual genotypes is drawn from this probabilistic model only to evaluate it and to finally provide a concrete solution.

EDA does not involve variation operators; instead, it adapts the parameters of the model in a representation-dependent way. It should be clear by now that EDA at conceptual level is a kind of meta-algorithm that can be adapted to any representation of a genotype, hence can be combined with any variant of evolutionary algorithm presented in Section 2.3.

2.5 Applications

Evolutionary algorithms proved to be useful in solving non-trivial real-world problems in engineering, information technology, physics, medicine, forensics and many other areas. In this section we attempt to summarize some of the recent famous and human-competitive achievements of EAs. All of the results presented below received *Humies* — awards given in annual competition to human-competitive results developed automatically with use of biology-inspired algorithms [8]. The achievements of genetic programming will be reviewed in a separate Section 3.12 in the next chapter.

Manos *et al.* [105, 104] used a multi-objective genetic algorithm to design the geometry of polymer optical fibres designed for a particular areas of interest. They developed a genotype-phenotype mapping that maps a bitstring genotype into a radially symmetric structure representing a cross-section of a optical fibre that is guaranteed to satisfy manufacturing constraints. Various kinds of designs were discovered, and as the authors claim, new design themes were identified. Finally the optical fibre designed by GA was manufactured and its usefulness was experimentally proved.

Yao *et al.* [191] developed genetic algorithm for fast and robust detection of ellipses in imaging data. Their algorithm outperformed industry-standard family of Hough transform algorithms. Hough transform was first designed for detection of lines [69], next generalized to an arbitrary shape [45], then immunized to a noisy data by randomization [189]. Yao's genetic algorithm outperformed the most robust Hough transform variant in terms of accuracy of approximation of complex and overlapping shapes, insensitivity to the absolute size of a hape and low number of false positives.

Grasemann *et al.* [61] used coevolutionary genetic algorithm to develop wavelet for image encoder for task of compression of fingerprint images. The developed encoder outperformed the industry standard JPEG2000 [2] and a hand-designed wavelet that won FBI competition for fingerprint image compression software [76] in terms of peak signal-to-noise ratio of the compressed image w.r.t. the original one. Consequently the developed wavelet enables storing fingerprint images with a higher quality given the same storage space, or with the comparable quality requiring $15\% - 20\%$ less storage.

Bartels *et al.* [18] used evolutionary strategies to optimize pulse shape of a laser beam to indirectly control X-ray harmonics emitted by atoms of gas in response to the laser light. Use of evolutionary strategies allowed authors to tune spectral characteristics of the emitted radiation and improve intensity of X-rays by an order of magnitude.

Li *et al.* [98] used mixed-integer evolution strategies to optimize parameters for detectors of lumen in intravascular ultrasound images. The proposed technique automatically finds good parameter values for a given set of images and the obtained results were not worse than the human-tuned parameters for each considered data sample.

Glazer *et al.* [58] proposed a genetic algorithm-based framework for automatic defect classification of wafers used for semiconductor manufacturing. To date, the industry standard defect classifiers are manually designed with use of expert knowledge. The proposed method, when compared to the manual design, showed significant improvement of accuracy of classification of defective parts. Moreover, the method automatically adapts to a changes of technology and manufacturing process, while the human-designed detectors need to be adjusted manually after every technological change.

Chapter 3

Genetic Programming

In this chapter we introduce genetic programming, describe solution representation, and thoroughly discuss the underlying components of the genetic programming algorithm. Next we review the non-canonical, but widespread variants of genetic programming, and discuss the differences between genetic programming and other variants of evolutionary algorithms. The chapter is concluded by survey of applications of genetic programming.

3.1 Introduction

Genetic Programming (GP) is a domain-independent evolutionary approach intended for induction of discrete structures using discrete, atomic parts. The parts are often instructions and the structures are computer programs, hence the name of the method. The basic methodology and assumptions on genetic programming were initially developed by John Koza in early 1990s [80]. In Sections 3.2 — 3.8 we discuss Koza's vision of GP which, augmented with only a few improvements through the years, became the canon of contemporary GP. For brevity we narrow our description to GP applied to program induction.

3.2 Program Representation

In GP, programs are made of instructions:

Definition 3.1. An instruction is an function of an arbitrary arity, domain and codomain.

Instructions can be divided into *terminals*, i.e., nullary instructions, and *non-terminals*, i.e., instructions that take one or more arguments. Terminals include constants, which return the same value in every execution or named variables that read program arguments upon execution . Non-terminal instructions are typically domain-dependent and can be divided into application instructions, e.g., math or Boolean operators, and flow-control instruction, e.g., jumps, conditional ifs etc. Notice the implicit assumption coming from Definition 3.1 that each instruction is required to return exactly one output value.

Definition 3.2. Instruction set Φ has the *closure* property if all instructions in it are type-consistent and safe to evaluate [80, 141].

Instructions in a set Φ are type-consistent if all of them accept outputs produced by any other instruction from Φ. Although type consistency may be considered limiting, it is always satisfied when a domain and codomain of instructions represent the same type. This is the case when evolving arithmetic or Boolean expressions. However when mixing types a little fiddling with the

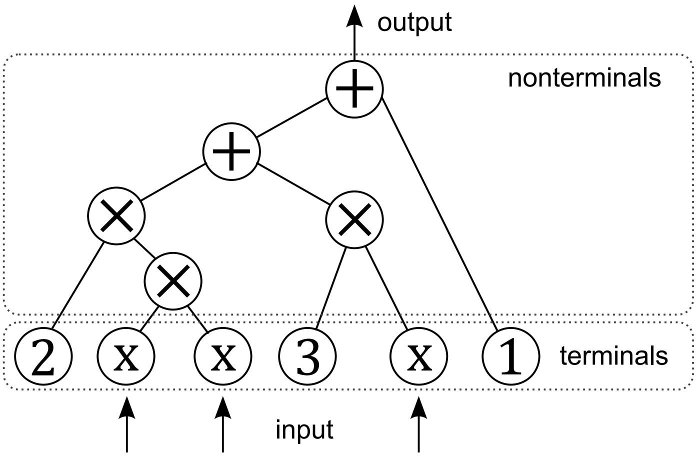


Figure 3.1: Exemplary abstract syntax tree for the program $2x^2 + 3x + 1$. Terminal and nonterminal nodes are marked by dotted frames. Control flow is denoted by arrows.

definitions of instructions is required. For instance we can employ instructions with automatic conversion of types, e.g., to convert real values to Boolean ones, treat non-positive real values as logical zero and positive real values as logical one. Conversion in the other way can assign arbitrary numeric values to logical zero and one. An alternative to requiring type consistency is using type-aware operators (cf. Definitions 2.4, 2.5, 2.10, 2.11) that do not violate the constraints imposed by an instruction set. This is the case for strongly-typed GP [113], where type system of instructions and a grammar resulting from this system are respected.

An instruction is safe to evaluate if it must not fail at runtime. Instruction may fail due to an invalid argument, e.g., division supplied with zero divisor. In GP it is quite common to handle such cases by protected versions of instructions. For instance, division is usually protected by assuming that it returns either 0 or 1 if divisor is 0, depending on implementation [80, 141, 168].

Definition 3.3. Program $p : I \to O$ is composition of instructions from Φ, where I is a set of program inputs and O is a set of program outputs.

In canonical GP solutions are typically represented at genotypic level by abstract syntax trees (AST). AST as used in GP is a tree, where each node contains an instruction. A number of children of a node reflects the arity of the consisting instruction, hence leaf nodes are terminals and intermediate nodes consist of non-terminals. Terminals representing named variables read program input, each intermediate node reads the outputs of its children nodes and computes an output. Root node of the tree returns program output. An example of AST is shown in Figure 3.1. From Definition 3.3 it follows also that the considered programs have no side effects nor memory persistent across executions.[1] Also, all such programs by definition halt (since program is actually a function).

It should be clear by now that genotype (cf. Definition 2.1) p in GP takes the form of program p and a set of all genotypes $\mathcal{P}$ becomes a set of all programs $\mathcal{P}$ that can be produced from a given instruction set Φ, i.e., in terms of formal languages p is a word under alphabet Φ and $\mathcal{P}$ is a language under alphabet Φ, $p \in \mathcal{P}$.

3.3 Problem statement and fitness assessment

Program induction problem as approached in GP is an optimization problem $\Pi = (\mathcal{P}, f)$ where one is supposed to construct a program p that maximizes (or minimizes) a fitness function f. The

[1] GP with persistent memory and side effects can be easily implemented, however for the scope of this thesis, it is enough to consider programs devoid of these features.

Algorithm 3.1: Full and grow initialization operators for program tree. $h \in \mathbb{N}_{\geq 1}$ is maximum height of the tree, $\Phi_t \subset \Phi$ and $\Phi_n \subset \Phi$ are subsets of all terminals and nonterminals in Φ, respectively, Random$(\cdot)$ returns a random element from the given set, Arity$(\cdot)$ returns arity of the supplied instruction.

```
function Full($h$,$\Phi_t$,$\Phi_n$)
    if $h = 1$ then
        return Random($\Phi_t$)
    $r \leftarrow$ Random($\Phi_n$)
    for $i = 1..$Arity($r$) do
        $r_i \leftarrow$ Full($h - 1$,$\Phi_t$,$\Phi_n$)
    return $r$
```

```
function Grow($h$,$\Phi_t$,$\Phi_n$)
    if $h = 1$ then
        return Random($\Phi_t$)
    $r \leftarrow$ Random($\Phi_n \cup \Phi_t$)
    for $i = 1..$Arity($r$) do
        $r_i \leftarrow$ Grow($h - 1$,$\Phi_t$,$\Phi_n$)    :: Assign child node
    return $r$
```

solution space $\mathcal{P}$ is constrained to programs obtainable using given instruction set Φ. This statement is consistent with Definition 2.2. Fitness function typically measures some kind of an error, amount of used resources (e.g., time) or accuracy of a program on a set of training examples. Actually a program in GP has to be run to assess its behavior and total cost of evaluating a program in GP involves multiple executions of the program with different inputs or in a different environment states. In fact each program in the population has to be executed to obtain its fitness and to make the programs comparable.

Over the years many techniques for limiting the evaluation cost were proposed, however they are mainly technical tricks that seem to be natural for every experienced software developer are beyond the scope of this thesis. An interested reader is referred to [141, 101].

Definition 3.4. Instruction set Φ is *sufficient* if the set of all programs $\mathcal{P}$ that can be constructed from Φ contains at least one satisfactory solution, i.e., a solution with fitness above (below) a given threshold.

Naturally by extending the instruction set by an additional instruction, the solution space may be also extended, hence more satisfactory solutions may be included in it, allowing GP find such solution more easily. On the other hand the inclusion of unnecessary or redundant instructions in the set seems to not negatively affect GP's performance in most cases [141, Ch 3]. Notice lack of sufficiency allows GP to induce only an approximate solution, hence it is usually better to include an additional instruction in the set, than not. Unfortunately sufficiency is hard to be satisfied and can be guaranteed only for those problems, where theory or experience state that a satisfactory solution can be obtained from the given instruction set [141, Ch 3]. For instance when inducing a continuous differentiable arithmetic expression, a set of polynomial instructions, e.g., $\{+, -, \times, x\} \cup \mathbb{R}$ is sufficient, since polynomials are able to express any C^{∞}-class function e.g., using Taylor series [173]. For induction of Boolean programs it is sufficient to use any functionally complete set of instructions, e.g., $\{\text{nand}, \text{in}_1, \text{in}_2, ..., \text{in}_n\}$, where in_i reads ith component (argument) of program input. In general it may be necessary to use a set of instructions that is known to be Turing complete [175].

3.4 Population initialization

Like all EAs, GP involves an initialization stage that determines the initial search points (candidate solutions). The earliest and to our knowledge the most common initialization operators are Full and Grow operators [80] presented in Algorithm 3.1. The modes of operation of both of them are very similar. Each of them takes as an argument the maximum tree height h_{max}[2] to be generated and recursively creates instruction nodes to be attached in the respective nodes of the tree being

[2] In this thesis, we strictly differentiate between the notions of *height* and *depth*, which are often confused in other works. Height is property of tree — length of the longest path from the tree root to leaf, while depth is property of node — length of the path from the tree root to the node.

created. The *full* operator always produces full trees, i.e., at depths from 1 to $h_{\max} - 1$ nodes are randomly chosen from the set of nonterminals, while the remaining nodes are chosen from the set of terminals, hence leafs are always at the maximum depth. The *grow* operator creates trees of variable height and shape by using all instructions for the choice at each node having depth less than the maximal and the set of terminals for the remaining nodes.

Although both operators guarantee that the produced tree does not violate the constraint on tree height, they do not explicitly constrain the size of produced trees, i.e., number of nodes in the tree. The actual tree size depends on arity of nonterminals, $h_{\max}$, and for grow operator also on the proportions of terminals to nonterminals. A high fraction of terminals leads to small trees, as they are then more likely to be selected before h_{max} is reached. Otherwise, the grow operator creates big trees. Note that greater $h_{\max}$ usually leads to greater overall size of initialized programs. On average, the full operator creates bigger trees than the grow operator.

In practice neither full nor grow operators provide satisfactory variety of tree shapes and sizes on their own [80]. Therefore, they are typically paired together, which is known as the Ramped Half and Half method [80] (RHH). RHH initializes population by running, with h_{max} varying in a user-specified range, the full operator for half of the population and the grow operator for the other half. Syntactic duplicates are discarded.

3.5 Selection

To our knowledge most researchers use tournament selection (cf. Definition 2.8) as a preferred selection operator in GP [141, Ch 2]. Selection pressure of μ-TS is insensitive to absolute values of fitness, can be fine tunable and kept roughly constant during GP run. These features make μ-TS perfectly fit to most GP applications. The tournament size typically used in GP practice varies from three [178, 179], through four [70, 144, 149], five [115, 167], six [70] to seven [101, 87, 133, 19].

3.6 Crossover

The canonical approach to cross-over two programs in GP is *Tree-Swapping Crossover* (TX) [80, 141]. TX is presented in Algorithm 3.2 and its exemplary application to a pair of programs is presented in Figure 3.2. Given two parent program trees p_1 and p_2 TX begins with randomly drawing from them nodes p'_1 and p'_2, respectively. The chosen nodes p'_1 and p'_2 are called *crossover points*. Typically, a node is chosen from the tree leafs with probability 10% and from the intermediate nodes with probability 90%. The root node is never chosen. These probabilities were originally proposed by Koza [80] to overcome a strong bias for choosing leafs by an uniform choice of nodes. In a full binary tree of height h, a leaf is chosen uniformly with probability ${}^{2^{h-1}}/_{(2^h-1)} \approx {}^1/_2$, effectively causing half of the crossover points to be leafs. Next TX replaces the subtree rooted at p'_1 in parent p_1 with subtree rooted at p'_2 in parent p_2, creating so the offspring. The remaining parts of parent trees are wasted.

Alternatively the second offspring may be created from the remaining parts, which is quite common approach [101]. To be consistent with Definition 2.11 we assume that TX is run twice to produce two offspring: the second time with swapped parents and with fixed crossover points in both runs.

TX only mixes code fragments given in parents, and as such may lead to lose of some code fragments, especially when run under strong selection pressure. This in turn may lead to loss of diversity and poor results, as programs may permanently lose access to some input variables. This is sometimes worked around by using large populations, e.g., $|P| \geq 1000$ [80, 81] or even $|P| \geq 100000$ [11, 82]. The argument is that complete loss of them is unlikely.

Algorithm 3.2: Tree-swapping crossover operator. RandomNode$(\cdot)$ picks a random node from the given tree, Replace(p_1, p_1', p_2') replaces subtree rooted at p_1' in p_1 with subtree p_2'.

```
function TX($p_1$,$p_2$)
    $p_1' \leftarrow$ RandomNode($p_1$)
    $p_2' \leftarrow$ RandomNode($p_2$)
    return Replace($p_1$,$p_1'$,$p_2'$)
```

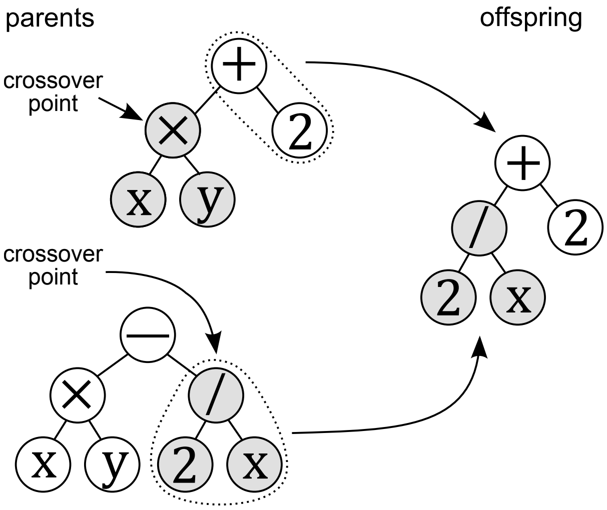


Figure 3.2: Exemplary application of the tree-swapping crossover. Given two parent program trees $p_1 \equiv xy + 2$ and $p_2 \equiv xy - 2/x$, crossover points are picked at $p_1' \equiv \times$, $p_2' \equiv /$, respectively, splitting each parent into two parts. The gray part of p_1 is replaced by gray part of p_2, producing so the offspring $2/x + 2$.

3.7 Mutation

Tree Mutation (TM) [80] is a unary counterpart of tree-swapping crossover. TM is presented in Algorithm 3.3 and its exemplary run is shown in Figure 3.3. The operator acts similarly to tree-swapping crossover. Given a single parent program p, TM first picks a mutation point p' using the same rules as TX. Then it generates a random tree r using, e.g., grow operator (cf. Algorithm 3.1). Finally, TM replaces the subtree rooted at p' in p by r, producing so the offspring.

Due to similarity of TM and TX, the mutation is often technically realized by crossover with a random tree. This technique is known as headless chicken crossover [13].

Unlike TX, TM introduces new code fragments into population and is used as a common workaround for lost of diversity and stagnation, especially in small populations. Note that a proper proportion of mutation and crossover is essential to successful optimization [102, 103].

Algorithm 3.3: Tree mutation operator. RandomNode$(\cdot)$ and Replace$(\cdot,\cdot,\cdot)$ are defined in Algorithm 3.2, Φ_t, Φ_n and Grow$(\cdot,\cdot,\cdot)$ are defined in Algorithm 3.1.

```
function TM($p$)
    $p' \leftarrow$ RandomNode($p$)
    $r \leftarrow$ Grow($h$,$\Phi_t$,$\Phi_n$)
    return Replace($p$,$p'$,$r$)
```

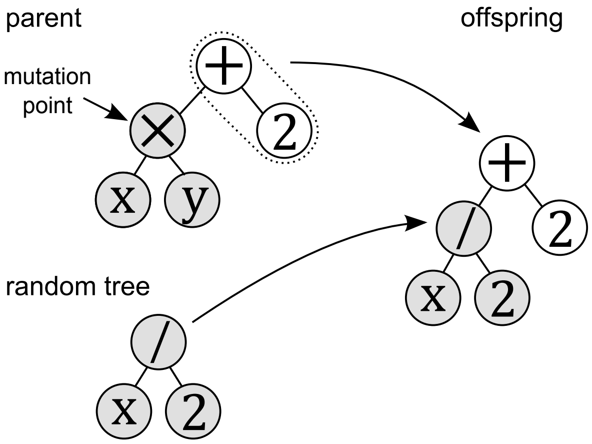


Figure 3.3: Exemplary run of tree mutation. Given parent program $p \equiv xy + 2$, a mutation point $p' \equiv \times$ is picked, next random tree $r \equiv x/2$ is generated and replaces tree rooted at p', producing offspring $x/2 + 2$.

$$\begin{tikzcd}
 & & TX \arrow[dr, Rightarrow] & \\
P \arrow[r] & \mu\text{-}TS \arrow[ur, Rightarrow, "\Pr(X)"] \arrow[r, "\Pr(M)"] \arrow[dr, "\Pr(R)"'] & TM \arrow[r] & P' \\
 & & R \arrow[ur] &
\end{tikzcd}$$

Figure 3.4: Breeding pipeline. Individuals from population P are subjected to selection, then a probabilistic choice is made which operator executes next. The offspring produced by a selected operator is added to population P' of the next generation. Double arrows indicate two individuals traversing a particular path at once.

3.8 Breeding pipeline

The default configuration of GP involves both mutation and crossover. In contrary to most EAs where mutation is applied after crossover, in GP it is common to perform crossover and mutation in parallel. In each iteration of the loop in lines 6 – 9 of Algorithm 2.1, evolutionary algorithm in line 8 executes variation operators. In GP, the operator is picked at random. More formally crossover is executed with probability $\Pr(X)$ and mutation with probability $\Pr(M)$, $\Pr(X) + \Pr(M) \leq 1$. If neither of them is picked GP reproduces (cf. Definition 2.9) the selected parent individual(s) with probability $\Pr(R) = 1 - \Pr(X) - \Pr(M)$.

Typically in GP $\Pr(X) \approx 0.9$ and $0.01 \lesssim \Pr(M) \lesssim 0.1$ [80, 141, 101]. These setting plus tournament selection of size in range $\langle 3, 7\rangle$ (cf. Section 3.5) constitute the typical so-called 'breeding pipeline' of genetic programming, i.e., a structure that defines the flow of individuals from the current population P to the next generation population P'. The pipeline is visualized in Figure 3.4.

3.9 Other variants of Genetic Programming

In this section we briefly introduce the non-canonical variants of genetic programming.

3.9.1 Linear Genetic Programming

Though conventional computer programs usually have tree-like structure, i.e., code units are organized into methods, classes, namespaces and binaries, at the level of a single method the code

Table 3.1: Execution of Push program `(2 4 INTEGER.* 5.1 6.2 FLOAT.- FALSE TRUE BOOLEAN.AND)`.

Step	Stacks	Comment

1	`EXEC: (2 4 INTEGER.* 5.1 6.2 FLOAT.- FALSE TRUE BOOLEAN.AND)`
	`BOOLEAN: ()`
	`FLOAT: ()`
	`INTEGER: ()`
		Program is loaded onto execution stack

2	`EXEC: (INTEGER.* 5.1 6.2 FLOAT.- FALSE TRUE BOOLEAN.AND)`
	`BOOLEAN: ()`
	`FLOAT: ()`
	`INTEGER: (4 2)`
		2 and 4 are popped from execution stack and pushed into integer stack

3	`EXEC: (5.1 6.2 FLOAT.- FALSE TRUE BOOLEAN.AND)`
	`BOOLEAN: ()`
	`FLOAT: ()`
	`INTEGER: (8)`
		Integer multiplication is executed by popping 4 and 2 from the integer stack and pushing resulting 8

4	`EXEC: (FLOAT.- FALSE TRUE BOOLEAN.AND)`
	`BOOLEAN: ()`
	`FLOAT: (6.2 5.1)`
	`INTEGER: (8)`
		5.1 and 6.2 are popped from execution stack and pushed onto float stack

5	`EXEC: (FALSE TRUE BOOLEAN.AND)`
	`BOOLEAN: ()`
	`FLOAT: (1.1)`
	`INTEGER: (8)`
		Floating point subtraction is executed by popping 6.2 and 5.1 from float stack and pushing resulting 1.1

6	`EXEC: (BOOLEAN.AND)`
	`BOOLEAN: (TRUE FALSE)`
	`FLOAT: (-1.1)`
	`INTEGER: (8)`
		FALSE and TRUE are popped from execution stack and pushed onto Boolean stack

7	`EXEC: ()`
	`BOOLEAN: (FALSE)`
	`FLOAT: (-1.1)`
	`INTEGER: (8)`
		AND is executed by popping TRUE and FALSE from Boolean stack and pushing FALSE

usually forms a linear sequence of instructions. This statement is true for almost all industrial-standard programming languages, like e.g., C++, C#, or Java. Industry-standard computer architectures, like e.g., x86 or ARM, execute programs instruction by instruction, in a sequence, except the control flow instruction like e.g., conditionals and loops. Thus even if a software developer prepares a tree-like program structure, compiler translates it into a sequence of instructions.

These observations inspired research on induction of programs represented as sequences. The first works on *Linear Genetic Programming* (LGP) date back to the first half of 1990s, to the works of Banzhaf [17], Perkis [136] and Nordin [125]. Another popular linear GP paradigm is embodied by *Push* programming language and the *PushGP* evolutionary system [167, 164, 165, 166]. Although PushGP is often classified as LGP system, Push features nesting of instructions and strictly speaking Push programs are not linear as whole, but in parts.

Push programs run on an interpreter equipped with several stacks, each assigned to different data type. There is a Boolean stack, float stack, integer stack, code stack and execution stack. Common instructions consist of pushing and popping values onto and from a stack, peeking the values on top of the stack, duplicating of peeked value, swapping of two peeked values or emptying the whole stack. There are also instructions dedicated to each stack-type separately. For instance, primitive logic instructions, e.g., not, and, or etc., are available for top elements of Boolean stack, and e.g., $+, -, \times$,... for float and integer stacks. There are also instructions for type conversion and move data from stack to stack.

A program in Push is simply a sequence of instructions. Thus, the complete LR grammar[3] of Push consists of two lines:

```
PROGRAM  := (SEQUENCE)
SEQUENCE := ∅ | instruction SEQUENCE | literal SEQUENCE | (SEQUENCE)
```

where instruction belongs to the set of available instructions, literal is an arbitrary value not being an instruction and parenthesis are used to group parts of program together, e.g., to indicate boundaries of subroutine.

A Push program is executed in the following manner. First, the interpreter pushes the program onto execution stack. Then, it repeats the following steps until the stack becomes empty:

- Pop the first element from the execution stack,
- If it is a single instruction then execute it,
- Else if it is a literal then push it onto the appropriate stack,
- Otherwise split it into separate items[4] and push onto execution stack in a reverse order.[5]

An example of Push program and its execution are presented in Table 3.1. A complete description of Push programming language is presented in [166].

Push programs are to be evolved by PushGP evolutionary system. To perform this task PushGP employs standard variation operators of linear genetic programming. LGP crossover given two parent programs randomly chooses two subprograms (crossover points), one in each parent, then exchanges a subprogram chosen from the first parent with the one chosen from the second parent producing so the offspring. LGP mutation identifies a random subprogram in the parent program, and replaces it by a randomly generated one, producing so the offspring. An interested reader is referred to [145] for more details on linear GP and PushGP.

3.9.2 Grammatical Evolution

Another tree-based genre of genetic programming is *Grammatical Evolution* (GE) [128, 153]. GE makes use of context-free grammars expressed in Backus–Naur Form [15] that constrain the way in which instructions can be combined into a program. A genotype in GE encodes a derivation tree of a given grammar (list of productions) using an integer vector that denotes a sequence of 0-base indexed productions from the list of productions that can be applied at the current state of derivation. Hence GE involves an additional level of abstraction that separates encoding of a program from program code.

In original formulation [128, 153] integers in the genotype are encoded as bitstrings and evolved using standard mechanisms of genetic algorithms (cf. Section 2.3.3). Since these mechanisms are unaware of the actual grammar and numbers of valid productions, they may produce genotypes that consist of integers out of the range. The common workaround involves modulo a number of valid productions to map them into the range.

Later the other approach was proposed where homologous crossover operator works at the level of integer vectors, instead of bitstrings [129]. The operator given two parent genotypes, derives their derivation trees, calculates a common region of productions rooted at the starting production. Next it draws randomly a production that is in the common region (first crossover point), the same in both parents, and the other production outside the common region (second crossover point), separately in both parents.[6] Finally the operator performs two-point crossover (cf. Section 2.3.3) adopted to integer vectors using the drawn points to produce an offspring. The analysis in [129] has

[3] A grammar where parser reads input from **L**eft to right and produces reversed **R**ightmost derivation.

[4] The popped element must be a list.

[5] Thus the first element is going to be placed at the top of the stack and the execution order will have been maintained.

[6] There were also proposed a version where the second crossover point is common in both parents [129].

shown that both above described approaches are statistically equivalent in terms of performance and achieved results.

3.9.3 Cartesian Genetic Programming

In *Cartesian Genetic Programming* (CGP) proposed by Miller [111, 112] programs (genotypes) are sequences of integers that encode planar graphs, with graph nodes being instructions. The nodes are arranged into a discrete grid. The graph is encoded as a sequence of numbers, grouped in triples. The first number indicates the opcode of instruction and the following integers indicate the grid coordinates of instruction arguments (ancestors in the control flow graph). To prevent cycles, an instruction is allowed to point only to coordinates smaller than its own coordinates. Program inputs have designated coordinates, hence can be fetched in the same manner. Program output is obtained by calling the instruction in the designated output location, which in turn recursively calls its predecessors. Hence a program in CGP acts like an electronic circuit, which by the way is quite common application of CGP [54, 73, 159].

Graphic structure of CGP programs allows reuse of subprograms many times, which is not possible for tree programs in canonical GP.

Recently there was proposed Recurrent Cartesian Genetic Programming (RCGP) [176], where the graph may contain cycles. To effectively handle the cycles, instructions are executed in the predefined order and each instruction reads its arguments from the most recent outputs of other instructions or program inputs. If an instruction attempts to read output value of an other instruction that has not been executed yet, the output value is considered to be 0.[7] The program stores its own state in instructions' outputs, thus if executed again with the same input, may result in different output. RCGP is useful in applications where a memory persistent across program executions or feedback information are required, e.g., in agent controller scenarios, where an agent is supposed to explore and discover its operating environment [176].

CGP typically employs point mutation as the only variation operator. Given a parent genotype, mutation first randomly draws an integer in genotype. Then it determines the function of that integer: if the integer is an opcode, then it is replaced by a random integer (and thus random opcode). Otherwise, if the integer refers to a coordinate in the grid, a new valid coordinate is drawn.

3.9.4 Estimation of Distribution Algorithms in Genetic Programming

In order to use EDA (cf. Section 2.4) with tree-based GP, EDA must be augmented to model probability distribution of tree-like structures, i.e., programs. To our knowledge the first successful attempt to EDA in GP was presented by Sałustowicz *et al.* [154] under the name of *Probabilistic Incremental Program Evolution* (PIPE). In PIPE, the probability distribution is modeled using a tree of an arbitrary size and shape, where each node holds a probability distribution of instructions. Distributions in leafs are limited to terminals only, however the ones in the intermediate nodes involve all instructions. Programs are sampled by recursively traversing the tree from tree root to leafs and drawing in each node an instruction according to the probability distribution stored there. Obviously the choice of an instruction in a node determines the number of child nodes that are actually visited, and the choice of a terminal instruction terminates a given branch of tree.

PIPE models a different probability distribution in each tree node, however it does not store information about conditional probability distributions. These observations where confirmed experimentally [127], where the performance of standard GP and PIPE were statistically equivalent.

[7] Or other 'neutral' symbol depending on the domain.

This drawback of PIPE has been addressed simultaneously by Sastry *et al.* [155] and Yannai *et al.* [190]. In the former work a method called *Extended Compact Genetic Programming* (ECGP) was proposed, that is an extension to PIPE, which for each node and each instruction in probability table of this node stores entire subtree of probability tables instead of a single value. In this way, the probabilities in a given subtree are conditional with regard to the choice of the root instruction of that subtree. The latter work introduces *Estimation of Distribution Programming* (EDP) method, that models probability distribution with a Bayesian network, hence the probability table in each tree node consists of conditional probabilities w.r.t. to instructions in a subset of other nodes (cycles in the network are forbidden). Note that EDP is able to model interdependencies between nodes located in different branches of the tree, while ECGP can only model probability w.r.t. the ascendants. ECGP requires thus (on average) less storage for the probabilistic model.

EDAs in GP are not limited only to the tree-based representation. In the work of Poli *et al.* [142] an N-Gram GP system was proposed that combines a linear program from n-grams of instructions and models the probability distribution using a multi-dimensional matrix. Each dimension of the matrix corresponds to the specific index in an n-gram and enumerates all instructions. Each matrix element contains the probability of choosing a particular sequence of instructions as an n-gram to be appended to a partial program. The N-Gram GP was compared with a simple form of linear GP and resulted in a noticeable improvements of performance, especially for more difficult problem instances [142].

3.10 Challenges for Genetic Programming

Almost 30 years of research on genetic programming identified a few issues inherent to GP methodology. Below we briefly discuss the most important of them, explain their origins and name possible countermeasures and remedies.

The behavior of a program, meant as the output it produces for some input, is not only determined by a particular set of instructions it is composed of, but also by how those instructions are arranged together. Even a minute modification of this combination may have dramatic impact on the program's behavior. For instance negation of a loop condition, or swapping of arguments of an asymmetric arithmetic operation may result in completely different program behavior. On the other hand a complete reorganization of program parts may result in exactly the same behavior. This observation is known as *ruggedness of genotype-phenotype mapping* in GP (cf. Definition 2.1), which has been investigated in studies on fitness-distance correlation [37] and locality of representation [148]. The complexity comes from the way how instructions interact: one instruction may enhance or negate the effects of another or be neutral in certain syntactic structures. Additionally, the number of combinations of instructions is exponential w.r.t. the instruction number and a program size, which makes exhaustive search of the program set infeasible. What is more a number of program (or instruction) input values is huge, or even infinite in some domains, which reflects in difficult evaluation of the program behavior (that incorporates e.g., all corner cases).

In addition most random programs implement simple behaviors. For instance, Langdon [90] showed that almost half of Boolean programs sampled uniformly out of all Boolean programs built upon instructions $\{\mathrm{and}, \mathrm{nand}, \mathrm{or}, \mathrm{nor}, \mathrm{in}_1, \mathrm{in}_2, ..., \mathrm{in}_8\}$ implement contradiction (a program that returns 0 for all inputs) and almost other half tautology (a program that returns 1 for all inputs). Of the remaining programs, about 1% rewrite inputs in straight or negated form and approximately 1‰ implement multi-input $\mathrm{nor}, \mathrm{nand}, \mathrm{and}, \mathrm{or}$ depending on the considered maximal program size. A similar distribution but with a less bias of tautology and contradiction was observed by Beadle *et al.* [21] in the initial populations initialized by RHH. On the other hand, we showed in [133] that some behaviors of arithmetic expressionsbuilt upon instructions $\{+, -, \times, /, \sin, \cos, \exp, \log, x\}$ are more common than others and most of longer expressions are in fact linear combinations of simpler ones. The conclusion from these works is that the distribution of behaviors of programs that come from uniform sampling of program space or are generated by conventional initialization operators is highly non-uniform, with high fraction of behaviors that humans consider as simple. This observation is important, because simple behaviors are usually not interesting from the perspective of real-world program induction problems.

Another challenge for GP are *introns*. Intron is a fragment of a program that does not influence program's final output, i.e., is not executed at all (like *dead code* in conventional programming), or its outcome is being cancelled by processing in another program fragment. The former case occurs when a control flow is unable to reach some instructions, e.g., when a conditional instruction evaluates to a constant value. The latter case occurs, e.g., when multiplying an expression by 0 or calculating a logical alternative with 1; in such cases, the outcome of the second argument of the instruction does not matter. Note that these situations not only happen when a part of a statement is a constant literal, it may also be an entire subprogram that evaluates to a constant value. Therefore the exact way to detect introns involves static analysis of a program code and boils down to solving Boolean satisfiability problem that is known to be NP-complete [38, 97]. Thus it is usually not practiced in GP, except where domain-dependent features enable us to perform static analysis efficiently [46]. Instead it is common to verify whether code fragments have impact on fitness [126] or on program outputs [34].

Opinions on usefulness of introns are divided in GP community. Some of the researchers protect introns as a mechanism of redundancy in code that prevents destruction of useful code fragments by variation operations [126, 56], while others [12, 46] criticize introns for detrimental influence on GP's performance. Moreover a recent study indicates no relation between existence of introns and GP's performance [34]. Disagreements here clearly stem from the number of factors and parameters that influence overall GP's performance in a complex manner.

Nevertheless it is not subject to doubt that introns contribute to program growth during an evolutionary run. The growth of program size during evolution is a natural consequence of adaptation of programs to a given problem: with the growing size of programs, the number of programs that solves the given problem is non-decreasing. Unfortunately the growth of introns has no influence on program outcome and the effort put into evolving such a growing code may be virtually lost.

The uncontrolled growth of code size that does not reflect in fitness improvement is called *code bloat*, or *bloat* for short [141, Ch 11]. Bloat negatively affects computation costs of fitness evaluation and variation operators. In addition oversized programs are difficult to interpret for humans, usually overfit to a training data and feature poor generalization abilities.

Langdon [89] showed that bloat is not only essential to GP, instead it comes from variable-length representation of genotype and may be encountered also by other optimization techniques. This is because there is no correlation between program size and its fitness. And since there are more big programs than the small ones, big programs are more likely to occurs in a population.

On the other hand Soule *et al.* [160] experimentally showed that in canonical GP introns are more likely to occur in deep parts of abstract syntax trees than in the shallow ones, thus are typically smaller than an average size of a subtree. Moreover TX and TM are more likely to select crossover (or mutation) point in these deep regions than in shallow ones, hence are very likely to occur in the intron parts. This plus observation that no matter what code is inserted into intron part the entire part still remains intron, lead to conclusion that there is no bias for inserting highly fit code parts in variation operations occurring in introns. Juxtaposing this with the Langdon's observation results in that big subtrees are more likely to be inserted into intron part of program, than the remaining part.

Quite different explanation was provided by McPhee *et al.* [108]. They claim that standard variation operators in GP are biased in producing offspring similar to their parents. The existence of similar individuals in population leads to better average fitness and lower variance, however mixing of similar individuals decreases diversity and causes bloat.

A natural way to cope with bloat is imposition of limits on size of a program, i.e., for canonical GP operators are not allowed to produce individuals of greater height or size than the limit, and if such an individual is created, it gets discarded. The particular way to handle this case is operator and implementation-dependent, the operator may be run again with the same or other arguments, or its argument is returned as the operator's output. Crane *et al.* [40] showed that imposing either height or size limits on abstract syntax trees result in similar distributions of tree sizes in a population, with a remark that height limits appear to bias a bit distribution towards lower heights than the corresponding size limits. A similar analysis has been conducted by McPhee *et al.* [107] for linear GP, where the authors show that low limits on program length cause quick convergence of an average program length in population to a certain value, while for high limits no convergence occurs, instead the average length of programs oscillates over time in a certain range.

The other way to fight bloat involves bloat-aware operators. The common method here is *parsimony pressure* [80] that subtracts (or adds) a scaled program size from a program's fitness. Typically the scaling coefficient is constant across evolutionary run, however Zhang *et al.* [192] showed that dynamic adaptation of this coefficient may be beneficial. Nevertheless the method takes into account only the absolute size of programs and does not employ information on distribution of program sizes in population. From that reason Poli proposed a Tarpeian method [139] that randomly assigns the worst fitness to some of the above-average sized individuals and then runs a selection operator of a choice.

Langdon proposed size-fair crossover [91] for canonical GP that acts like TX, however with the second crossover point in the second parent restricted to the points where the size of an underlying subtree is no more than as twice as big than the subtree rooted at the first crossover point in the first parent. Similarly Langdon proposed size-fair mutation [93] that restricts the size of subtree to be inserted into parent w.r.t. the size of subtree to be deleted in the parent. Later Crawford-Marks *et al.* [41] moved the idea of size-fair operators to linear GP.

3.11 Genetic Programming in contrast to other Evolutionary Algorithms

All evolutionary algorithms presented in Section 2.3, except genetic programming have one common feature: all of them optimize a given model of a problem. That is, each genotype determines a particular combination of values of free variables of the model, and the interpretation of those variables within a model is fixed. In genetic programming the model is not known in advance, and only the constraints of the model are given, e.g., instruction set, the maximum size of a genotype etc. Hence GP creates and optimizes entire model of the problem.

In other words, GP evolves an executable code that inputs a given problem instance and outputs a solution to the instance, while conventional EAs evolve solutions directly. In this sense GP's solutions are active, i.e., adapt to the given problem instance, and other EA's solutions are passive, i.e., when applied to a different problem instances, they often turn out to be infeasible or obtain much worse fitness.

It should be clear by now that executable code evolved by genetic programming (in fact a variant of a heuristic search algorithm) can be an other heuristic search algorithm, where the other heuristic search algorithm is a solution to the given problem and the other heuristic search algorithm's output is a solution to a particular problem instance. The heuristic synthesis of a heuristic search algorithm is known under the name *hyper-heuristic optimization* [147, 29].

From technical perspective the above properties are mainly reflected in the length of genotype and mode of its fitness evaluation. In conventional EA genotype has usually a fixed length, i.e., number of free variables is fixed in the model, while in GP genotype changes over time, since a particular program structures are added and/or removed. Because genotype in EA solves only a particular instance of a particular problem, it must be applied only once to the problem to assess the solution's fitness. In contrast, an objective[8] fitness assessment of GP's program requires multiple executions of the program on different problem instances. Therefore, fitness assessment in GP is by design computationally much more demanding than in other variants of EA.

3.12 Applications

In this section we present some of the most remarkable achievements and applications of genetic programming presented in the past literature. As when reviewing other EAs (cf. Section 2.5) we focus here on works that were awarded in the Humies competition [8]. From a wide range of biologically-inspired, not limited only to EAs, techniques included in the competition, GP scored about 54% of gold awards and 52% of total awards, including gold, silver and bronze, in the years $2004 - 2014$. This indicates how important and how powerful GP is in solving real-world problems when compared with the other techniques.

Hornby *et al.* [67, 99, 100] used genetic programming to automatically design an antenna that meets the demanding characteristics specified by NASA for use in spacecraft. First, the authors automatically evolved an antenna that was 100% compliant with the NASA's requirements. Then, the initial requirements were modified due to a change of a launch vehicle, thus within four weeks a new design of antenna was developed and fabricated using the proposed technique. The new antenna again was compliant with the requirements and during tests in an anechoic chamber it outperformed the human-designed one, scoring 93% efficiency[9] in contrast to 38% of the human-designed one, which naturally reflects in higher range and stronger signal.

Forrest *et al.* [53] applied genetic programming to automatically fix errors in C programs created by human software developers. The authors took source codes or their fragments of 11 open-source and proprietary applications with known errors in their code. The software included, among others, operating system of Microsoft Zune media player [10], Null httpd web-server [5] and Flex lexical analyzer [4]. Genetic programming supplied with a small set of positive tests that verify if the software behaves according to the specification, and even a smaller set of negative tests[10] that cause failures, fixed all the errors, each in on average three minutes. The same authors in a more recent work [96] used GP to fix errors in a more complex applications, like PHP interpreter [6], Python interpreter [7] and Wireshark network sniffer [9]. This time 52% of errors were successfully fixed. The authors employed Amazon EC2 cloud computing infrastructure [3] to estimate costs of automatic software repair, which resulted in $7.32 per error on average.

Spector *et al.* [163] evolved discriminator [182], Pixley [137], majority [16] and Mal'cev [71] terms of finite algebras that were many orders of magnitude smaller than the forms created by previously known methods. Moreover, the authors compared the efficiency of GP system with to the date the most time-efficient algebraic method by Clark *et al.* [36] that turned out to be a few orders of magnitude slower than GP.[11]

In another work Spector [162] used PushGP to evolve programs for quantum computers. The proposed approach was experimentally verified with success on a series of quantum computing problems, e.g., unstructured database search, a general gate of a certain function that scales with the size of a problem instance, and a quantum communication without a direct physical link between communicating objects.

[8] Truly objective fitness assessment would embody all problem instances, however in most problems the number of instances would cause the objective assessment to be computationally infeasible, thus it is common practice to conduct fitness assessment only on a sample thereof.

[9] Amount of energy emitted divided by amount of energy consumed.

[10] Cardinalities of sets of positive and negative tests were in $\langle 2, 6\rangle$ and $\langle 1, 2\rangle$, respectively, depending on the application.

[11] Months vs minutes for small algebras and years (estimate) vs minutes for bigger algebras.

Widera *et al.* [184] created energy functions for prediction of folding of protein ternary structures that achieve predictive power better than an expert-designed ones and optimized by Nelder-Mead downhill simplex algorithm [121].

Schmidt *et al.* [157] proposed a genetic programming approach to solve iterated function problems, i.e., problems in form $g(g(g(x))...) = h(x)$, where $h(x)$ is known and $g(x)$ is to be found. The iterated functions are in general useful in revealing functions producing fractals [24], image compression [23] and modeling of dynamical systems [26]. According to the authors they obtained a brand new solution to the problem of finding formula for $g(x)$ in equation $g(g(x)) = x^2 - 2$ that is the first non-trigonometric solution to this problem published in the literature.

Chapter 4

Semantic Genetic Programming

In this chapter we introduce the notion of semantics and show how it can be used in genetic programming. Then we define related to the semantics properties of effectiveness and neutrality for GP operators.

4.1 Introduction

Semantic Genetic Programming (SGP) is a branch of GP that involves semantic information about evolved programs. In theory of formal languages, semantics is a rigorous specification of meaning or behavior of programs [124]. There are three general ways to describe semantics: operational semantics that describes how a program is to be executed by an abstract computational machine [1, 138], denotational semantics that models behavior with mathematical objects that represent effects of computation [158], and axiomatic semantics that expresses behavior by means of constraints and assertions to be met by a program before and after execution [124].

In recent research in GP a kind of denotational semantics has been widely adopted. The GP semantics involves representation typical to data sets in supervised learning problems.

Formally, training data is a list of fitness cases:

Definition 4.1. A *fitness case* is a pair of program input and the corresponding correct output, i.e., $(in, out^*) \in F \subseteq I \times O$, where F is a list of fitness cases, each of them having unique input, i.e., $\forall_{i=1..|F|} in_i \in (in_i, out^*) \in F, \forall_{j=1..|F|, j \neq i} in_j \in (in_j, out^*) \in F : in_j \neq in_i$, I and O come from Definition 3.3.

In other words, a fitness case is a sample of a correct program behavior: it describes the outcome of a single execution of a hypothetical program given an input in and returning a correct output out^*. Given a specific program p and a fitness case (in, out^*), p executed with an input in returns its (not necessary correct) output out, which we denote as $out = p(in)$.

From above formulation it follows that for each input there is exactly one correct output. This is also a common assumption in machine learning, which GP can be seen a form of.

Definition 4.2. A *sampling semantics* (*semantics* for short) $s \in \mathcal{S}$ is a tuple of $|F|$ values from O, such that its elements correspond to fitness cases in F, i.e., $s = (out_1, out_2, ..., out_{|F|})$, $\forall_{i=1..|F|} out_i \in O$. *Semantic mapping* $s : \mathcal{P} \to \mathcal{S}$ is a function with property $s(p_1) = s(p_2) \Leftrightarrow \forall_{i=1..|F|} in_i \in F : p_1(in_i) = p_2(in_i)$, where $p_1, p_2 \in \mathcal{P}$.

In other words, sampling semantics consists of a finite *sample* of outputs provided for each fitness case in F. Thus, sampling semantics is not a complete description of a program behavior and depends on the choice of fitness cases in F.

The definition of semantics does not refer to a program, which allows manipulating semantics directly without knowing if a program having that particular semantics exists. Therefore, we can even synthesize an 'abstract' semantics if needed.

Since a part of a program is a valid program itself (cf. Definition 3.3), a semantics can be calculated for any subprogram of a given program to describe its behavior more thoroughly than just by its outputs.

If elements of O are numbers, semantics takes the form of a vector [178, 86, 186, 87, 179, 55, 133, 34]. However in this thesis we assume that O holds any objects (e.g., Booleans), hence we prefer to use the term 'tuples'.

Definition 4.3. Given a set of fitness cases F, a *target* $t \in \mathcal{S}$ is a semantics composed of the correct outputs for each fitness case, i.e., $t = (out_1^*, out_2^*, ..., out_{|F|}^*), \forall_{i=1..|F|} out_i^* \in F$.

The divergence between the correct output out^* and the one out returned by a program p is an error committed by p on this particular fitness case, e.g., a binary error. In the following we assume that fitness function $f(\cdot)$ is monotonous[1] with respect to the errors obtained for each fitness case in F. However, at this point we do not need to specify yet how the errors are aggregated. The fitness of a program p can be calculated directly from its semantics $s(p)$. A program p having semantics $s(p) = t$ is optimal with respect to $f(\cdot)$. In terms of the bio-inspired terminology introduced in Section 2.1.2, the domain of the fitness function is set of phenotypes (cf. Definition 2.2), which here is a set of semantics $\mathcal{S}$, and semantic mapping $s(\cdot)$ is a genotype-phenotype mapping from Definition 2.1. Because fitness function involves F, F implicitly becomes a part of the optimization problem $\Pi = (\mathcal{P}, f)$ from Definition 2.2.

To express similarity between semantics we use semantic distance:

Definition 4.4. *Semantic distance* between two semantics is a function $d : \mathcal{S} \times \mathcal{S} \to \mathbb{R}_{\geq 0}$ with the following properties: $d(s_1, s_2) = 0 \Leftrightarrow s_1 = s_2$ (identity of indiscernibles), $d(s_1, s_2) = d(s_2, s_1)$ (symmetry), $d(s_1, s_3) \leq d(s_1, s_2) + d(s_2, s_3)$ (triangle inequality).

Properties of semantic distance cause it to be a metric. The simplest function that meets the above requirements is discrete metric, i.e.,

$$d(s_1, s_2) = \begin{cases} 0 & s_1 = s_2 \\ 1 & \text{otherwise.} \end{cases} \tag{4.1}$$

In the following we commonly use semantic distance on semantics of specific programs, hence we allow for an abuse of a notation and use a shorthand $d(p_1, p_2) \equiv d(s(p_1), s(p_2))$.

4.2 Semantic neutrality and effectiveness

GP searches program space $\mathcal{P}$ using variation operators, whose action may result in move in $\mathcal{P}$ that is not reflected in move in semantic space $\mathcal{S}$. This, however, is undesirable from the perspective of achieving the goal, i.e., finding the optimal program.

Below we introduce the notions of semantic neutrality and effectiveness for particular operators introduced in Section 2.2.

Definition 4.5. An application of an initialization operator is *neutral* w.r.t. population P iff semantics $s(p')$ of an initialized program p' is equal to a semantics of any program p already in the population P, i.e., $\exists_{p \in P} s(p) = s(p')$. Otherwise the application is *effective*, i.e., $\forall_{p \in P} s(p) \neq s(p')$. Initialization operator is *effective* iff all its applications are effective.

[1] $f(\cdot)$ must be non-decreasing w.r.t. errors for fitness minimization, and non-increasing for maximization.

The role of initialization operator is to provide initial sampling of solution space. If a population consists of programs having duplicated semantics, the number of actually sampled distinct solutions is smaller than a number of individuals in the population, hence the computational effort invested in neutral initializations is lost. On the other hand population that consists exclusively of semantically distinct programs fully exploits its sampling potential and is more likely to reveal beneficial solutions.

In analogy to initialization operators, an n-mate selection operator from Definition 2.6 can be effective or not:

Definition 4.6. An application of an n-mate selection operator to mate programs $\{p_1, p_2, ..., p_{n-1}\}$ and a population P is *neutral* iff semantics of the selected nth program $p' \in P$ is equal to a semantics of any of the mate programs, i.e., $\exists_{i=1..n-1} : s(p_i) = s(p')$. Otherwise the application is *effective*, i.e., $\forall_{i=1..n-1} : s(p_i) \neq s(p')$. n-mate selection operator is *effective* iff all its applications are effective.

According to Algorithm 2.1, the programs resulting from selection and mate selection operators become input to variation operators, and as such become parents to offspring created there. The parents are representatives of good solutions in the current population and should constitute a semantically diverse sample.

Definition 4.7. An application of a variation operator to a set of parents $\{p_1, p_2, ..., p_n\}$ is *neutral* iff semantics $s(p')$ of the produced offspring p' is equal to a semantics of any of its parents, i.e., $\exists_{i=1..n} : s(p_i) = s(p')$. Otherwise the application is *effective*, i.e., $\forall_{i=1..n} : s(p_i) \neq s(p')$. Variation operator is *effective* iff all its applications are effective.

That is, we say that a variation operator is effective if all produced offspring are semantically different from their parents. An offspring is effective w.r.t. its parents iff it is produced by an effective application of a variation operator, otherwise we attribute the offspring *neutral*.

Note that from properties of genotype-phenotype (semantic) mapping (cf. Section 2.1.1) two programs may have different code, but the same semantics, thus *syntactic* difference of an offspring and its parents is not sufficient for an application to be effective.

Nevertheless, it is challenging to design a variation operator that guarantees effectiveness, in particular in the extreme case of all parents being semantically identical. An exception is mutation, because it takes only one parent. Therefore, the natural way is to provide semantically different parents to the variation operator by an effective mate selection operator. Use of this idea for all $n \geq 2$-ary variation operators together with effective mutations makes breeding effective, i.e., no offspring in the next population is semantically equal to any of its parents. Note that this still does not guarantee that the next population contains no programs that are semantically equal to the ones in the current population, since the produced offspring is not compared to the semantics of programs in the current population.

The notion of neutrality is not new in GP, however its meaning outside SGP is quite different. In GP, the concept of neutrality occurs in two contexts: (i) neutral code, i.e., introns [79, 49], and (ii) neutral mutation that does not lead to phenotypic [79] or fitness [49] change. The latter meaning is somehow similar to the one presented in Definition 4.7, however these early studies use bitstrings [79] or integer vectors [49] for genotypes and abstract syntax trees [79, 49] for phenotypes. In contrast, we, and the majority of researchers in the field, consider now AST to be a genotype and semantics to be a phenotype. Recently the topic of semantic neutrality (in some studies called ineffectiveness [21, 131]) and effectiveness was addressed by a few papers on SGP, with some of the works defining neutrality and effectiveness of operators using semantics of entire programs [21, 20, 22, 133, 131, 55], while others using semantics of parents' and offspring's subprograms [178, 123, 179].

Chapter 5

Geometric Semantic Genetic Programming

The main aim of this chapter is to show how geometry of semantic space can be utilized by GP to improve search performance and solve difficult program induction problems. We discuss geometry of metric spaces and show how the choice of a metric for fitness function influences the shape of a fitness landscape. Next, we define geometry-aware semantic operators for GP and describe their properties. Then, we outline the main differences between the geometric operators and other semantic methods.

5.1 Metrics and geometries

We focus on three commonly used Minkowski metrics: L_1 (taxicab metric), L_2 (Euclidean metric), L_∞ (Chebyshev metric), where

$$L_z(x, y) = \sqrt[z]{\sum_i |x_i - y_i|^z}.$$

Definition 5.1. *A segment* $S(x, y)$ in a space X between x and y, $x, y \in X$, is the set of all $v \in X$ that satisfy $\|x, y\| = \|x, v\| + \|v, y\|$, where $\|\cdot, \cdot\|$ is a metric in X.

Note that we defined a segment using only a metric, thus we do not impose any restrictions on the objects x, y, v, except that they come from a metric space. Figure 5.1 presents the segments under the L_1, L_2, L_∞ metrics in the two-dimensional real space, i.e., $x, y, v \in \mathbb{R}^2$. An L_1-segment has the shape of a (hyper-)rectangle,[1] with x and y in the opposite corners and sides parallel to the

[1] Assuming that by hyperrectangle we mean the shape under L_2 metric, in the common sense. Formally it is L_1-segment. This comment applies also to other figures described in this section.

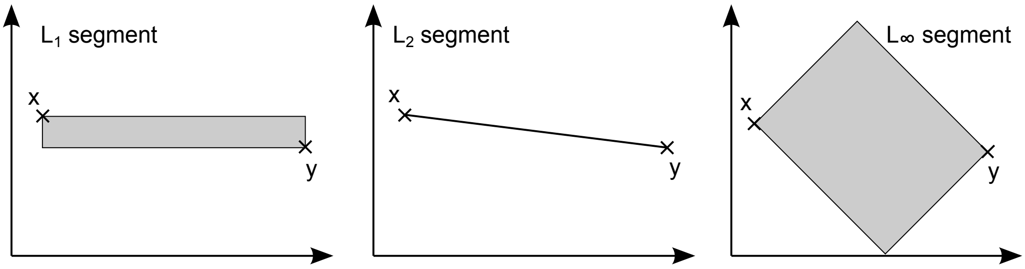


Figure 5.1: Segments in two dimensions under L_1, L_2, L_∞ metrics.

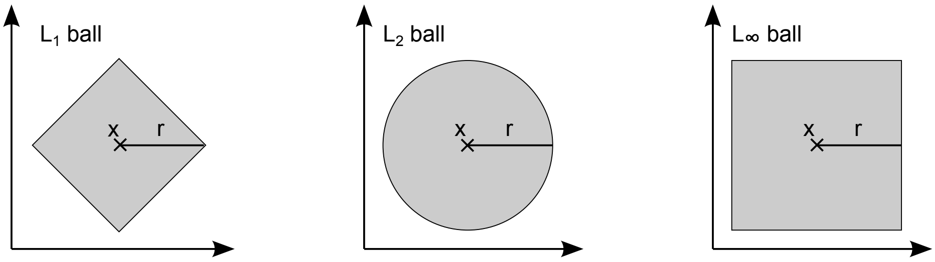


Figure 5.2: Balls in two dimensions under L_1, L_2, L_∞ metrics.

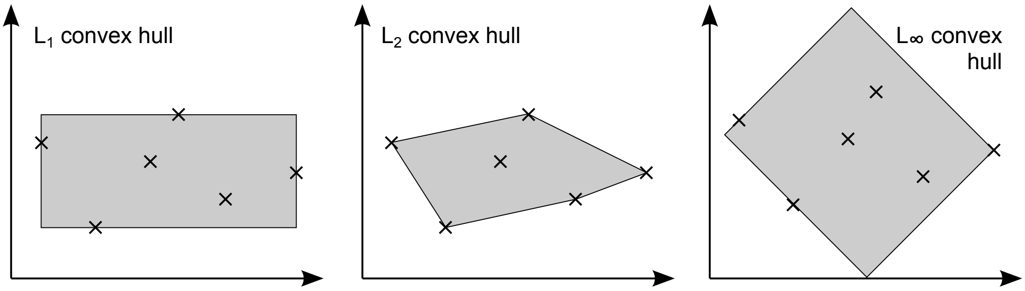


Figure 5.3: A convex hulls in two dimensions under L_1, L_2, L_∞ metrics.

axes of coordinate system. L_2-segment is just a section of a straight line passing trough x and y and delimited by them. L_∞-segment is a polyhedron (rectangle in 2D) with x and y in opposite corners and sides angled at 45° to adequate axes of the coordinate system.

Definition 5.2. *A ball* $B(x, r)$ of radius $r \in \mathbb{R}_{>0}$ with a center in x in a space X, $x \in X$, is set of all $v \in X$ that satisfies $\|x, v\| \leq r$, where $\|\cdot, \cdot\|$ is a metric. *Sphere* is an analogous set that satisfies $\|x, v\| = r$.

Figure 5.2 shows balls of radius r centered in x under the L_1, L_2, L_∞ metrics in two dimensional real space, i.e., $x, v \in \mathbb{R}^2$. L_1-ball takes form of regular polyhedron with edges inclined by 45° degrees with respect to the adequate axes of the coordinate system. L_2-ball is a filled sphere and L_∞-ball is a (hyper-)cube with sides parallel to axes of the coordinate system.

Definition 5.3. *A convex set*[2] is a set X where the segments between all pairs of elements in X are contained in X. *Convex hull* $C(Y)$ of set Y is the intersection of all convex sets that contain Y.

In other words, a convex hull of Y is the smallest convex set that contains Y. In Figure 5.3 we visualized the convex hulls for the L_1, L_2, L_∞ metrics in $\mathbb{R}^2$. L_1-convex hull is a (hyper-)rectangle with edges parallel to the axes of coordinate system, L_2-convex hull is a polygon where each vertex contains an element from the enclosed set Y and L_∞-convex hull is a polygon without the above property, but with sides angled by 45° to adequate axes of the coordinate system.

We will use shorthand notation $L_z(p_1, p_2) \equiv L_z(s(p_1), s(p_2))$, where $p_1, p_2 \in \mathcal{P}$, to indicate L_z distance between programs.

5.2 Fitness landscape

In Section 4.1 we assumed that fitness function measures an error committed by a program on a set of fitness cases, which is a common practice in GP. This idea is often implemented by means of *root mean square error* (RMSE) or *mean absolute error* (MAE). The key observation is that fitness

[2] Typically convex set is defined for L_2 geometry and its generalization to other metrics is referred to as *geodesically convex set*, however we do not separate these two notions, as our further considerations apply to both of them.

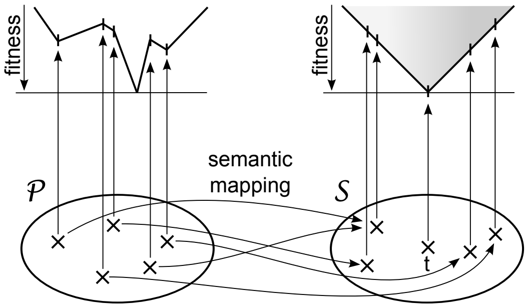


Figure 5.4: Transformation of solution space from $\mathcal{P}$ to $\mathcal{S}$ and impact on fitness landscape.

functions defined in such ways are metrics, where the first metric argument is semantics $s \in \mathcal{S}$ under assessment (possibly, but not necessarily, a semantics of an actual program p, $s = s(p), p \in \mathcal{P}$) and the second is target t: $f(s) = \|s, t\|$. Furthermore, a semantic distance is a metric too and may be used as the fitness function: $f(s) = d(s, t)$. Note that the above does not necessarily restrict semantic distance $d(\cdot, \cdot)$ and fitness function $f(\cdot)$ to be the same metric.

Fitness function plotted against a set of all programs $\mathcal{P}$ forms a surface, termed *fitness landscape* [188], where elevation of each point equals to a value of fitness of an underlying program. When an arrangement of underlying programs depends on their purely syntactic features, the fitness landscape is likely to be rugged with lots of valleys and hills and no clear area of good solutions e.g., so-called deep valley.

We redefine the arrangement of programs by employing semantic set $\mathcal{S}$ as a proxy between program set $\mathcal{P}$ and fitness landscape. That is, semantic mapping $s(\cdot)$ maps $\mathcal{P}$ onto $\mathcal{S}$ and semantic distance $d(\cdot, \cdot)$ imposes a certain structure onto $\mathcal{S}$ that reflects distances between program semantics and makes $\mathcal{S}$ a space.

The observation that is crucial here is that the fitness function is a metric and the fitness landscape proxied by semantic space is a cone with target in the apex, i.e., assigned the optimal zero fitness. Transformation of solution space and its impact on fitness landscape is visualized in Figure 5.4.

A particular shape of a fitness landscape cone depends on the metric. Figure 5.5 shows fitness landscapes under L_1, L_2, L_∞ metrics, i.e., $f(s) = L_z(s, t)$. Level sets of fitness landscape are spheres under the corresponding metric and dimensionality of semantic space, located at various levels of *fitness*. In L_1, fitness landscape cone takes the form of a regular pyramid with edges rotated by 45° to the fitness axis and level sets rotated by 45° to corresponding axes of the coordinate system. Under L_2 fitness landscape is just an ordinary cone. In L_∞ it is again a pyramid with edges inclined at 45° to the fitness axis, however with level sets parallel to corresponding axes of the coordinate system.

It should be obvious that searching semantic space under such fitness landscapes should be easy: in each point of the fitness landscape, gradient clearly indicates the direction towards the global optimum. There are also no plateaus, i.e., gradient does not zero anywhere. However, let us not forget that $\mathcal{S}$ is not the space being searched. The entire search process takes place in the program space $\mathcal{P}$, and $\mathcal{S}$ is only an image of it. Therefore, unless one designs specialized operators (see next section), the moves in $\mathcal{P}$ translate into moves in $\mathcal{S}$ in a very complex and hard to predict way. What is more semantics in $\mathcal{S}$ that are infeasible in $\mathcal{P}$ constitute *holes* in the fitness landscape — barriers that forbid specific moves in the semantic space and may be conflicting with direction of improvement indicated by the gradient of the fitness landscape, thus difficult to avoid. These two facts constitute the main challenges in searching the solution space.

In the following we attempt to exploit the properties of conic fitness landscapes by introducing the notion of geometric operators.

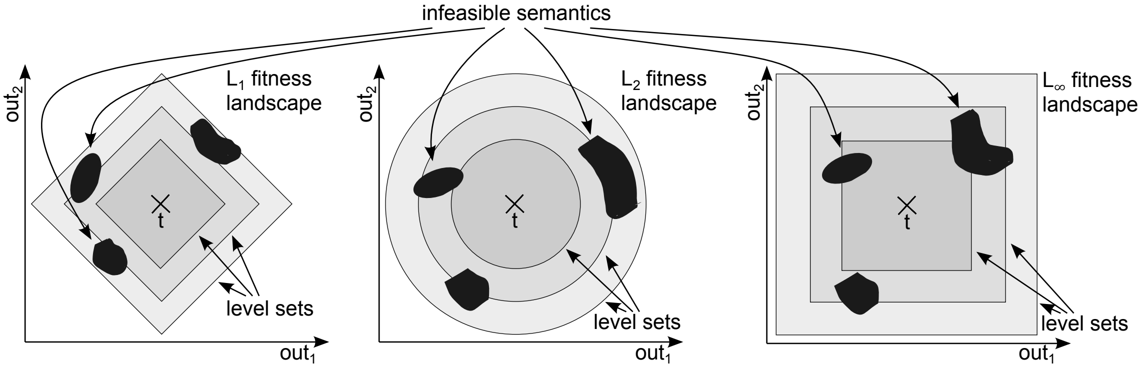


Figure 5.5: Fitness landscape under L_1 (left), L_2 (center), L_∞ (right) metrics. Black areas represent (exemplary) regions of semantics without counterpart in program set.

5.3 Geometric operators

We present and extend previously formulated by Moraglio *et al.* [115] framework of *Geometric Semantic Genetic Programming* (GSGP). The original work defines only geometric mutation and crossover, however we extend it for other variation operators, initialization and mate selection, thus making it a complete framework of operators for GP. In the following, each time we refer to GSGP, we mean a GP algorithm employed with geometric operators introduced in this extended framework, instead of the Moraglio *et al.* one.

In this section we introduce GSGP operators, and in the next one we discuss how the operators interact with each other and how they influence the evolutionary search.

As defined in Section 2.2, an operator is a random function, thus in below definitions we distinguish between properties of the operator and properties of its application.

Definition 5.4. An application of an initialization operator is *geometric* w.r.t. population P iff the convex hull of semantics $s(p)$ of programs p already present in P and semantics $s(p')$ of the initialized program p' includes the target, i.e., $t \in C(\{s(p) : p \in P\} \cup \{s(p')\})$. An initialization operator is *n-geometric* iff n is minimum size of population P for which all applications of this operator are geometric.

Left part of Figure 5.6 presents in $\mathbb{R}^2$ semantic space under L_2 metric two exemplary applications of an initialization operator given p_1 and p_2 are already present in P. An application of the operator that produces p' is not geometric, since the convex hull of semantics $s(p')$ and $s(p_1)$ and $s(p_2)$ (the dashed one) does not include the target t. An application that produces p'' is geometric, since the corresponding convex hull (the dotted one) includes t.

Note that 1-geometric initialization is an idealized case, where the convex hull must be built from a single program semantics, thus 1-geometric initialization must produce a program having target semantics. For real-world problems, such an operator is virtually unrealizable, and if it existed, it would solve the program induction problem in just one step.

Definition 5.5. An application of an n-mate selection operator to mate programs $\{p_1, p_2, ..., p_{n-1}\}$ and population P is *geometric* iff the convex hull of semantics $s(p')$ of the selected parent p' and semantics of mate programs $\{s(p_1), s(p_2), ..., s(p_{n-1})\}$ includes target, i.e., $t \in C(\{s(p'), s(p_1), s(p_2), ..., s(p_{n-1})\})$. An n-mate selection operator is *geometric* iff all its applications are geometric.

Left part of Figure 5.6 (the same like for initialization) presents in $\mathbb{R}^2$ under L_2 two exemplary applications of a 3-mate selection operator. Given that p_1 and p_2 are mate programs here, an application that selects p' is not geometric, because the convex hull of semantics $s(p')$ and $s(p_1)$ and $s(p_2)$ does not include t. An application that selects p'' is geometric, because the convex hull includes t.

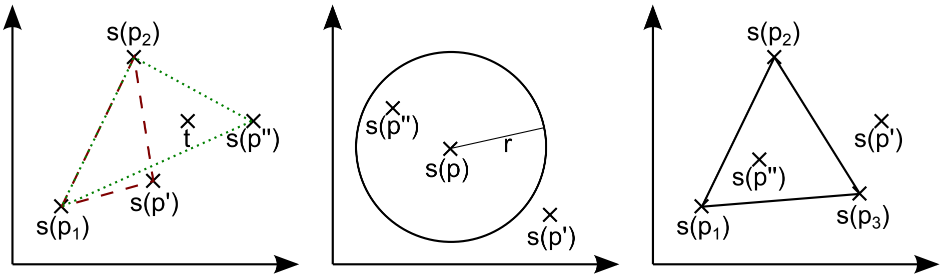


Figure 5.6: Exemplary applications of an initialization (left), a 3-mate selection (left too), a mutation (center) and an ternary variation operator (right) in $\mathbb{R}^2$ under L_2 metric: p's are results of non-geometric applications, p''s of geometric.

A necessary condition for an application of an n-mate selection operator to be geometric is existence of at least one program in the population that satisfies the above requirement, i.e., $\exists p' \in P : t \in C(\{s(p'), s(p_1), s(p_2), ..., s(p_{n-1})\})$. After Carathéodory's theorem [30], the necessary condition is satisfied in an $\mathbb{R}^{|F|}$ semantic space for $n \geq |F| + 1$ if t is in the convex hull of P, e.g., if P was initialized by a geometric initialization. Otherwise, an n-mate selection may not exist.

After [114, 115], we define geometric mutation operator:

Definition 5.6. An application of a mutation operator to a parent p is r-*geometric* iff the semantics $s(p')$ of a produced offspring p' belongs to $B(s(p), r)$ ball, i.e., $s(p') \in B(s(p), r)$. A mutation operator is r-*geometric* iff all its applications are geometric.

Center part of Figure 5.6 presents exemplary applications of a mutation operator given parent p. An application that produces p' is not r-geometric, because the distance of t to $s(p')$ is greater than r. An application that produces p'' is r-geometric, because the distance of t to $s(p'')$ is less than r.

Below we present the definition of $n \geq 2$-ary geometric variation operator that is a natural generalization of definition of geometric crossover introduced in [114, 115]:

Definition 5.7. An application of $n \geq 2$-ary variation operator to parents $\{p_1, p_2, ..., p_n\}$ is *geometric* iff the semantics $s(p')$ of a produced offspring p' belongs to the convex hull of semantics of its parents, i.e., $s(p') \in C(\{s(p_1), s(p_2), ..., s(p_n)\})$. An $n \geq 2$-ary variation operator is *geometric* iff all its applications are geometric.

Right part of Figure 5.6 presents exemplary applications of a ternary variation operator given parents p_1, p_2, p_3. An application that produces p' is not geometric, because $s(p')$ is outside the convex hull of the parents' semantics. An application that produces p'' is geometric, because $s(p')$ belongs to the convex hull of the parents' semantics.

In the following we adopt the naming introduced in this section also to programs and call them geometric if they were obtained by a geometric application of an operator.

5.4 Discussion on design of geometric operators

The general idea behind the extended framework of GSGP presented in this thesis, is to employ a convex hull of semantics of programs in a population (a convex hull of population for short) to geometrically restrict part of solution space to be searched, such that this convex hull incorporates the target t.

Figure 5.7 visualizes this idea in $\mathbb{R}^2$ semantic space under L_2 metric, and presents how the convex hull of the population changes over generations.

GSGP by means of geometric initialization operator begins from creating a population, which convex hull includes t (left part of Figure 5.7).

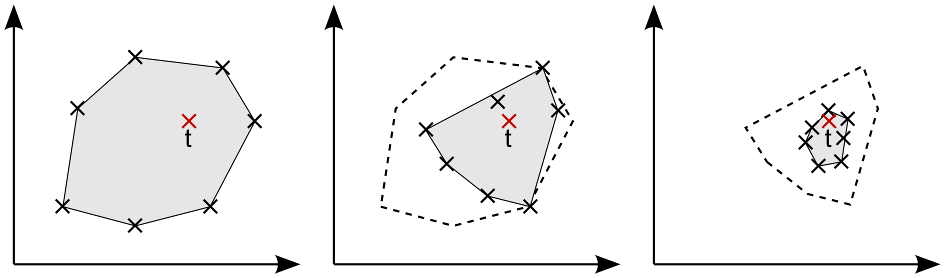


Figure 5.7: Visualization of the idea behind the extended framework of GSGP in $\mathbb{R}^2$ under L_2 metric. Left: initial population, center: second generations, right: third generation. Dashed polygons are convex hulls from previous generation.

In each iteration of evolutionary loop (cf. Algorithm 2.1) GSGP gradually shrinks the population's convex hull. This routine begins from selection of parents such that, a convex hull of their semantics is subset of the convex hull of the population. Then, variation operators are employed to produce offspring, such that a convex hull of an offspring's population is included in the convex hull of the parents' (center part of Figure 5.7).

In effect, the whole population is concentrated around the target (right part of Figure 5.7).

We indicate two necessary conditions allowing efficient search by $n \geq 2$-ary variation operators that come from that definition.

Theorem 5.8. *A geometric $n \geq 2$-ary variation operator can produce in one application a program having semantics t iff a convex hull of semantics of parent programs includes t.*

Proof. The proof comes from Definition 5.7, i.e., a geometric n-ary variation operator is unable to produce offspring that lies outside the convex hull of the parents, hence if t lies outside the convex hull, the operator is unable to reach t. □

The condition in the above theorem can be met by supplying variation operator with parents selected by geometric n-mate selection. Note that for crossover (2-ary operator) the convex hull in Definition 5.7 of n-ary geometric variation operator boils down to a segment between parents' semantics.

Theorem 5.9. *If all variation operators involved in search are geometric $n \geq 2$-ary variation operators, a program having semantics t can be acquired iff the convex hull of a population includes t.*

Proof. The proof comes from observation that convex hulls of subsequent populations are subsets of previous ones (a proof for crossover that can be generalized to $n \geq 2$-ary operators is to be found in Theorem 3 in [114]). □

In case the convex hull of a newly created population does not include the target, it can be expanded in two ways. First, by means of a geometric mutation operator, that is the only operator in GSGP able to expand the convex hull in the subsequent generation. Second, by adding some programs from the previous population to the new one, e.g., using elitism or a reproduction biased towards expansion of the convex hull.

Note that choice of a certain metric for semantic distance affects set of solutions that is obtainable by a single application of a geometric operator. In the next section we show how particular metrics influence properties of geometric operators and bound performance of the GSGP search.

5.5 Properties of geometric operators

From practical perspective, the key property of a variation operator is whether the moves it conducts in the search process improve the quality of solutions. In other words, whether a variation operator causes a search process to make progress. Extending our previous work [132] we define following properties of variation operators:

Definition 5.10. A variation operator is *weakly progressive* (WP) iff all the produced offspring p' are not worse than the worst of their parents $\{p_1, ..., p_n\}$, i.e., $f(p') \leq \max_{i=1..n} f(p_i)$.

In other words if an operator is WP it must not deteriorate the fitness of the produced offspring with respect to its parents. If all variation operators involved in search are WP, the worst fitness in the next population P' will be not worse than the worst fitness of parents selected for recombination, which in turn must be not worse than the worst fitness in the current population P, i.e., $\max_{p' \in P'} f(p') \leq \max_{p \in P} f(p)$. Note WP does not guarantee that the operator produces a solution strictly better than any of its parents. To overcome this deficiency, we define next property:

Definition 5.11. A variation operator is *potentially progressive* (PP) iff for every set of parents $\{p_1, ..., p_n\}$ an image of the operator $Im(OP(\cdot))$ (a subset of the operator's codomain containing offspring that can be created from the given set of parents) applied to this set of parents contains an offspring p' that is not worse than the best of its parents, i.e., $\exists p' \in Im(OP(\{p_1, ..., p_n\})) : f(p') \leq \min_{i=1..n} f(p_i)$.

In other words PP means that an operator is able to produce offspring not worse than all of its parents. In practice an operator that is both WP and PP has the potential of improving on average the worst fitness in a population every generation, thus in turn positively contribute to the search progress. Note that the operator being solely PP is not guaranteed to not worsen the fitness. From this reason we strengthen PP below:

Definition 5.12. A variation operator is *strongly progressive* (SP) iff all the produced offspring p' are not worse than the best of their parents $\{p_1, ..., p_n\}$, i.e., $f(p') \leq \min_{i=1..n} f(p_i)$.

SP property implies both WP and PP, hence the corresponding discussions apply here. SP property refers only to a single application of a variation operator and as such does not constrain a population wide characteristics of progress in a higher extent than WP property solely, i.e., $\max_{p' \in P'} f(p') \leq \max_{p \in P} f(p)$. Note also that we do not require SP operator to produce strictly better offspring, because operator like that is formally unrealizable: when supplied with an optimal parent program, it cannot produce a better offspring.

One may ask why we care of the properties defined above, given that GP, like other metaheuristics, occasionally accepts deterioration of working solutions, as a mechanism of escaping local optima and revealing alternative search paths towards the global optimum. First, these properties characterize variation operators in abstraction from a search algorithm, and in this sense are general. Second, because fitness landscape is unimodal, (cf. Section 5.2), the gradient in each place indicates the path towards the global optimumand there is no reason to worsen the best-so-far solution. However we admit that if regions of infeasible semantics exist in the landscape they may hinder reach of the global optimum, although independently on whether an operator has or not the above properties. Third, these properties refer only to variation operators, hence given a poorly fit parents the operator, even being SP, would still produce a poorly fit offspring. Therefore a convex hull of succeeding population is unlikely be much less than the convex hull of the current population.

Theorem 5.13. *The r-geometric mutation under a semantic distance $d = L_{z_d}$ and fitness function $f = L_{z_f}$ is potentially progressive.*

Proof. Consider a parent p and its fitness under L_{z_f} metric $f(p) = L_{z_f}(s(p), t)$. By definition of the ball, the L_{z_d} distance between p and its offspring p' located in an L_{z_d}-ball $B(s(p), r)$ amounts to $d(p, p') = L_{z_d}(p, p') \leq r$. From the Cauchy-Schwarz [169] and Hölder's [146] inequalities, the L_{z_f} distance of p to p' is bounded by

$$L_{z_d}(p, p') \leq \quad L_{z_f}(p, p') \quad \leq |F|^{\frac{1}{z_f} - \frac{1}{z_d}} L_{z_d}(p, p') \leq |F|^{\frac{1}{z_f} - \frac{1}{z_d}} r$$

if $0 < z_f \leq z_d \leq \infty$, otherwise

$$L_{z_f}(p, p') \quad \leq L_{z_d}(p, p') \leq r.$$

Triangle inequalities $f(p) \leq f(p') + L_{z_f}(p, p')$ and $f(p') \leq f(p) + L_{z_f}(p, p')$ set bounds for offspring's fitness $f(p')$, which we compactly write as $f(p') = f(p) \pm L_{z_f}(p, p')$. This in turn is bounded by

$$f(p') = \begin{cases} f(p) \pm r|F|^{\frac{1}{z_f} - \frac{1}{z_d}} & z_f \leq z_d \\ f(p) \pm r & \text{otherwise.} \end{cases}$$

In conclusion, the offspring can be better than its parent, thus an r-geometric mutation is PP for all combinations of L_{z_f} and L_{z_d} metrics. Moreover, an r-geometric mutation is not WP and SP, since the offspring may be also worse than the parent. □

Theorem 5.14. *A geometric $n \geq 2$-ary variation operator under the semantic distance $d = L_{z_d}$ and fitness function $f = L_{z_f}$ is weakly progressive if the L_{z_f}-ball is a convex set under L_{z_d}.*

Proof. Let $p_1, p_2, ..., p_n$ be the parents. Assume without loss of generality that $f(p_1) = \max_{i=1..n} f(p_i)$. Let $B(t, r)$ (B for short) be an L_{z_f}-ball, where $r = f(p_1) = L_{z_f}(s(p_1), t)$. By definition, $\forall_{i=1..n} f(p_i) \leq r$ and $\{s(p_1), s(p_2), ..., s(p_n)\} \subseteq B$. By the non-decreasing property of convex hull,[3] L_{z_d}-convex hull $C(B)$ is a superset of L_{z_d}-convex hulls of all subsets of B, i.e., $C(\{s(p_1), s(p_2), ..., s(p_n)\}) \subseteq C(B)$. If B is a convex set under L_{z_d} then $B = C(B)$ and $C(\{s(p_1), s(p_2), ..., s(p_n)\}) \subseteq B$ holds. Otherwise $B \subset C(B)$, thus there may be a pair of parents' semantics in B, such that an L_{z_d}-segment connecting them is not contained in B. In conclusion, if B is a convex set under L_{z_d}, the offspring $p' : s(p') \in C(\{s(p_1), s(p_2), ..., s(p_n)\})$ is not worse than p_1, i.e., $f(p') \leq f(p_1) = r$, and the geometric $n \geq 2$-ary variation operator in question is WP. □

Theorem 5.15. *The geometric $n \geq 2$-ary variation operator under the semantic distance $d = L_{z_d}$ and fitness function $f = L_{z_f}$ is potentially progressive.*

Proof. Let $p_1, p_2, ..., p_n$ be the parents. Assume without loss of generality that $f(p_1) = \min_{i=1..n} f(p_i)$. Let $B(t, r)$ (B for short) be an L_{z_f}-ball, where $r = f(p_1) = L_{z_f}(s(p_1), t)$. By definition, $s(p_1) \in B$ and an L_{z_d}-convex hull of parents has a nonempty intersection with B, i.e., $s(p_1) \in C(\{s(p_1), s(p_2), ..., s(p_n)\}) \cap B$. In conclusion, there exists an offspring $p' : s(p') \in C(\{s(p_1), s(p_2), ..., s(p_n)\})$ that is not worse than the best of the parents, i.e., $f(p') \leq f(p_1) = r$ and the geometric $n \geq 2$-ary crossover operator is PP. □

Theorem 5.16. *A geometric $n \geq 2$-ary variation operator under a semantic distance $d = L_{z_d}$ and fitness $f = L_{z_f}$ is strongly progressive if the L_{z_f}-ball is a convex set under L_{z_d} and all parents have the same fitness.*

Proof. Consult the proof of Theorem 5.14, and change the constraint on parents' fitness from $f(p_1) = \max_{i=1..n} f(p_i)$ to $\forall_{i=2..n} f(p_i) = f(p_1)$, so that it requires all parents to have the same fitness. Then the proof shows that offspring's fitness must not be worse then the fitness of the worst (and simultaneously the best) parent. □

[3] $X \subseteq Y$ implies that $C(X) \subseteq C(Y)$

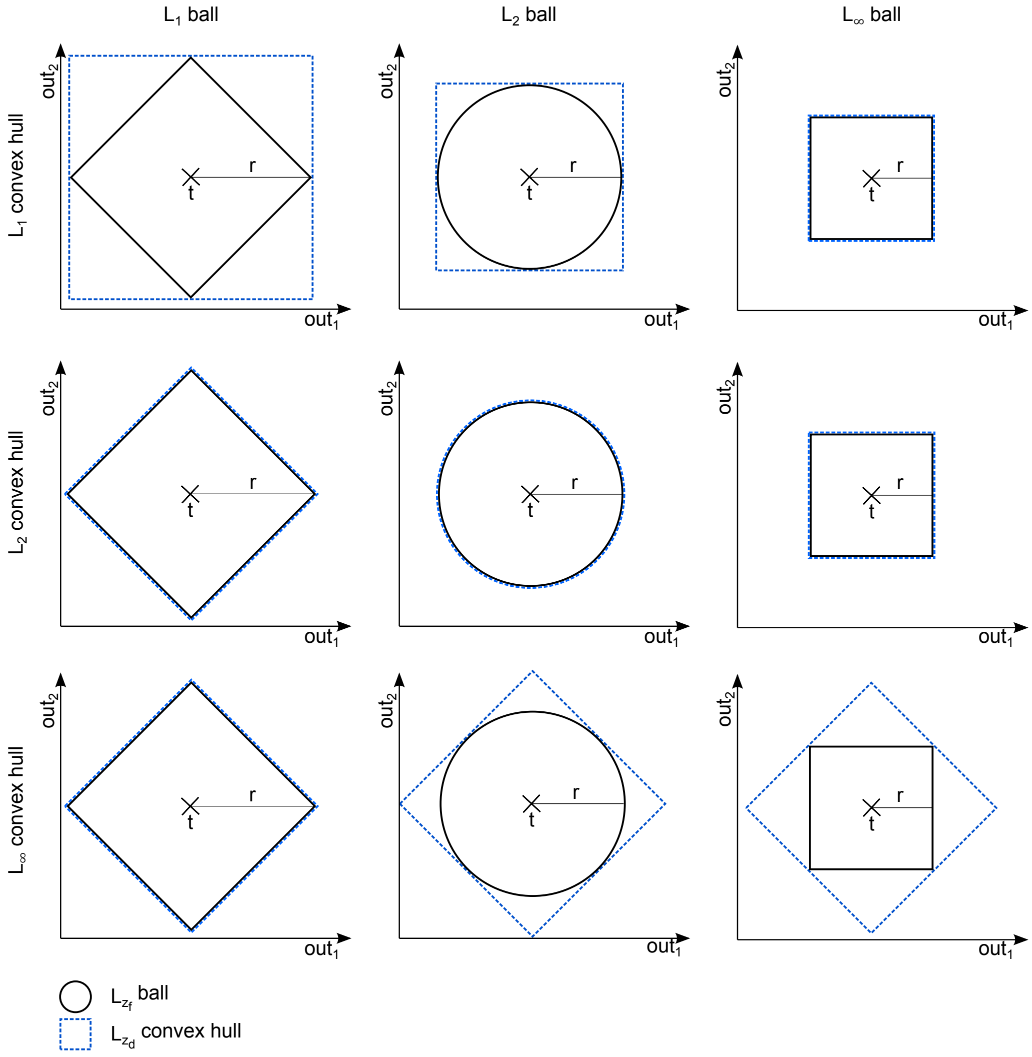


Figure 5.8: L_{z_f}-balls and their L_{z_d}-convex hulls for $z_f, z_d \in \{1, 2, \infty\}$.

The semantics of parents in Theorem 5.16 are equidistant from t, however they are not necessary the same. Actually the parents lie in the L_{z_f}-sphere centered in t with radius equal to parents' fitness.

Note that Theorems 5.14 – 5.16 can be easily generalized to other metrics, since their proofs do not refer to specifics of L_z, rather come from general properties of metrics. However other metrics are out of scope of this thesis, thus for consistency we continue using L_z throughout the rest of this text.

A question arises for which combinations of metrics' parameters, i.e., z_f and z_d, L_{z_f}-ball is a convex set under L_{z_d}. The theorems below answer this question and Figure 5.8 visualizes L_{z_f}-balls and their L_{z_d}-convex hulls in $\mathbb{R}^2$ to help understand the proofs.

Theorem 5.17. *An L_∞-ball is a convex set under L_1.*

Proof. L_1-convex hull is the smallest hyper-rectangle (cf. Section 5.1) enclosing an L_{z_f}-ball $B(t, r)$. The diameter of the ball is $2r$, thus the convex hull's edge length is also $2r$ in each dimension, and each coordinate of the point of the convex hull that is most distant from t differs from the coordinate of t by r. Therefore the distance of this point from t under any L_{z_f} metric is $\sqrt[z_f]{|F|r^{z_f}}$. Since $\lim_{z_f\to\infty} \sqrt[z_f]{|F|r^{z_f}} = r$, L_∞-ball is a convex set under L_1. If $z_f \neq \infty$, then $\sqrt[z_f]{|F|r^{z_f}} > r$ and L_{z_f}-ball is not a convex hull under L_1. □

Table 5.1: (a) Properties of progress and (b) upper bounds of ratios of offspring's fitness to the worst parent fitness for geometric $n \geq 2$-ary variation operator w.r.t. semantic distance $d = L_{z_d}$ and fitness function $f = L_{z_f}$, $z_d, z_f \in \{1, 2, \infty\}$.

(a)

$d \backslash f$	L_1	L_2	L_∞
L_1	PP	PP	WP, PP
L_2	WP, PP	WP, PP	WP, PP
L_∞	WP, PP	PP	PP

(b)

$d \backslash f$	L_1	L_2	L_∞
L_1	$\lvert F\rvert$	$\sqrt{\lvert F\rvert}$	1
L_2	1	1	1
L_∞	1	$\sqrt{2}$	2

Corollary 5.18. *For an $n \geq 2$-ary variation operator, the L_1 semantic distance and the L_{z_f} fitness function, the upper bound of ratio of offspring's fitness to its the worst parent is $\sqrt[z_f]{|F| r^{z_f}}/r = \sqrt[z_f]{|F|}$.*

Theorem 5.19. *An L_{z_f}-ball is a convex set under L_2 for $z_f \geq 1$.*

Proof. L_2-convex hull is a convex polygon that is tangential to L_{z_f}-ball $B(t, r)$ unless $z_f < 1$. Hence the L_{z_f}-ball and its L_2-convex hull are the same sets. For $L_{z_f<1}$, $B(t, r)$ is not a convex set since the L_2-convex hull of the $L_{z_f<1}$-ball is equivalent to L_1 ball of radius r. Under $L_{z_f<1}$, the point of the L_2-convex hull that is the most distant from t is located in the center of (any) side of the L_2-convex hull, i.e., at the distance of $\sqrt[z_f]{|F| \times ({}^{r}/_{|F|})^{z_f}}$ from t. □

Corollary 5.20. *For $n \geq 2$-ary variation operator, L_2 semantic distance and $L_{z_f<1}$ fitness function the upper bound of ratio of an offspring's fitness to its the worst parent is $\sqrt[z_f]{|F| \times ({}^{r}/_{|F|})^{z_f}}/r = |F|^{1/z_f - 1}$.*

Theorem 5.21. *L_1-ball is a convex set under L_∞.*

Proof. L_∞-convex hull takes the form of an L_1-ball that encloses the entire L_{z_f}-ball $B(t, r)$. The walls of L_∞-convex hull that are tangential to the L_{z_f}-ball $B(t, r)$ are at the distance r from t under L_{z_f}. The edges of L_∞-convex hull have length of $2r$ under L_{z_f} and their tangent points are in the middle. Thus, the L_{z_f} distance to the point of L_∞ convex hull that is the most distant from t (i.e., a vertex of the L_∞ convex hull polygon) is $r \times 2^{1-1/z_f}$. For $z_f = 1$, this distance amounts to r, hence the L_1-ball is a convex set under L_∞. □

Corollary 5.22. *For $n \geq 2$-ary variation operator, L_∞ semantic distance and L_{z_f} fitness function, if $z_f \geq 1$, then the upper bound of ratio of an offspring's fitness to its the worst parent is $r \times 2^{1-1/z_f}/r = 2^{1-1/z_f}$. If $z_f < 1$, the upper bound is the same as for L_2 semantic distance, i.e., $|F|^{1/z_f - 1}$.*

The upper bounds provided by proof to Theorem 5.13 and Corollaries 5.18, 5.20, 5.22 are derived for continuous semantic space. In discrete space that is in fact a subset of the continuous space, semantics (programs) reaching these bounds may not exist. Thus the actual upper bounds would be tighter in discrete space.

To sum up, in Table 5.1a we present how particular progress properties hold for geometric $n \geq 2$-ary variation operators. Note that SP holds for the same cases where WP, provided all parents have the same fitness (cf. Theorem 5.16). However, we omitted SP in Table 5.1a due to this additional requirement. Table 5.1b presents the maximum relative deterioration of offspring's fitness to the worst parent's under semantic distance $d = L_{z_d}$ and fitness $f = L_{z_f}$ for $z_f, z_d \in \{1, 2, \infty\}$.

From practical perspective, the above relationships between metrics are useful if definition of a program induction problem features a fitness function that is a scaled L_z metric, like RMSE or MAE. The choice of semantic distance and fitness function has critical importance for the properties of progress, and can be used as a means towards improving the performance of GSGP.

All fitness function metrics return 0 for the optimal program, hence the particular choice of a metric impacts only gradient of the fitness landscape, and in turn, the indicated way to the optimum. Therefore if fitness function is not fixed in problem definition, but, e.g., from technical reasons, semantic distance is a fixed metric, one may choose fitness function metric that enables GSGP to find the global optimum more smoothly.

Our results are consistent with the previous results of Moraglio [114], where he showed that if semantic distance and fitness function are the same metric and that metric satisfies Jensen's inequality [77] (e.g., L_2), geometric crossover is weakly progressive. Our result is more general, as the proof can be applied to different metrics for semantic distance and fitness function that may or may not fulfill Jensen's inequality on its own. Also, the reasoning above holds for all geometric $n \geq 2$-ary variation operators. Moraglio showed also that, given the same assumptions, a geometric crossover produces an offspring p' that on average has better fitness than an average fitness of its parents p_1, p_2, i.e., $\overline{f(p')} \leq \overline{(f(p_1)+f(p_2))/2}$.

5.6 Runtime analysis of Geometric Semantic GP

Let us outline runtime analysis of GSGP. We start with evolution of Boolean programs, where semantics are strings of bits, i.e., $\mathcal{S} = \{0,1\}^{|F|}$, like genotypes in genetic algorithms in Section 2.3.3. If semantic distance is Hamming distance [63], which for bitstrings is equivalent to L_1, a convex hull of $\{s_1, s_2, ..., s_n\}$ is a set of all points with coordinates being combinations of all values occurring on particular bits in s_is.

By merging all points comprising Hamming convex hull such that we store 0 and 1 on dimension where all semantics are respectively 0 and 1, and # otherwise, we obtain schema (cf. Definition 2.12) for all semantics contained in the convex hull.

For crossovers known from genetic algorithms we prove:

Lemma 5.23. *One-point, two-point and uniform crossovers are geometric crossovers under Hamming distance.*

Proof. Since one-point, two-point and uniform crossover exchange corresponding parents' bits, each bit is inherited in the offspring from one of its parents. Thus, the offspring belongs to the Hamming convex hull of its parents (an L_1 segment for two parents). □

Corollary 5.24. *Assuming fitness proportionate selection, one-point crossover and no mutation, schema theorem (cf. Theorem 2.14) holds for GSGP evolving Boolean programs. Consequently, the number of programs having semantics consistent with above average fit, short and low-order schemata increases over the course of evolution.*

In related work Moraglio *et al.* [119] derived probability of finding global optimum using a kind of generalization of GSGP and GA with uniform crossover and convex fitness function (e.g., L_2). Moreover, they showed that on average GSGP requires polynomial number of generations and polynomial size of a population w.r.t. number of fitness cases $|F|$, hence polynomial expected computation time w.r.t. $|F|$ to find the optimum.

In a more recent work Moraglio *et al.* [116] applied GSGP equipped only with mutation to evolution of programs operating on real values. For semantic distance and fitness function being L_2, they derived expected computation times $\Theta(|F| \log |F|)$ and $\Theta(\log {}^{1}/_{\epsilon})$ w.r.t. the number of fitness cases $|F|$ and a value $\epsilon > 0$ of the fitness function to be achieved. Moreover, they experimentally verified that GSGP with mutation on average requires fewer generations to solve a regression problem, than an equivalent variant of evolutionary strategy (cf. Section 2.3.2).

5.7 Comparison with other Semantic GP methods

A quite distinctive feature of geometric semantic GP, not found in other semantic GP methods, is directed and precisely designed search strategy. That is, each search step performed by operators of GSGP, as defined in Section 5.3, is devoted to shrink convex hull of a population and concentrate programs' semantics around the target.[4] In contrast most of other semantic methods (reviewed in the next chapter) are trial-and-error approaches that repeatedly apply standard syntactic operators until certain conditions are met. Therefore, the search performed by them is indirect, random, and in most cases blind to the structure of solution space. Actually, most of these methods are designed specifically to reduce amount of semantically neutral moves and to ensure effectiveness of the search (cf. Definitions 4.5 – 4.7), however with no bias in any specific direction.

Geometric operators are not effective by definition: a newly initialized program may be semantically equal to any other program already present in a population, parents selected for reproduction may have the same semantics, and a produced offspring may be semantically equal to any of them. Indeed, this can be considered as a drawback of GSGP methodology, however use of geometric operators does not preclude making them effective. These two features may coherently co-exist in a single operator.

The next difference between GSGP and other SGP methods lies in use of and requirements for semantic distance and fitness function. Most semantic non-geometric methods only verifies whether two programs are semantically equal or distinct, which can be done without use of semantic distance and fitness function. In contrast, GSGP requires both semantic distance and fitness function to be metrics, since metric is what establishes a geometry of the search space. Providing a metric for some domains may be difficult, but for semantics composed from program's outputs, i.e., sampling semantics (cf. Definition 4.2), a metric can be provided always, e.g., by means of Hamming distance.

[4] Except initialization, which in fact creates the initial population and its convex hull.

Chapter 6

Overview of past works on Semantic Genetic Programming

We review past works related to the semantics in GP, focusing but not limiting ourselves to tree-based GP. We divide them into contributions on semantic non-geometric GP and semantic geometric GP, and present them in Sections 6.1 and 6.2, respectively. The algorithms presented in the following are unified and/or generalized w.r.t. their original formulations to make them independent on semantic representation and/or domain, while keeping the original ideas intact. Every modification is indicated and discussed in text. While reviewing these algorithms, we pay special attention to their effectiveness (cf. Definitions 4.5 – 4.7) and geometry (cf. Definitions 5.4 – 5.7), as they are ones of the key concepts of this thesis.

6.1 Works on Semantic Non-geometric Genetic Programming

Albeit sampling semantics from Defintion 4.2 is the most common way of defining semantics in GP works, below we describe alternative representations of semantics found in past GP literature. Then we divide the works according to the part of GP workflow, and finally present some of the operator-independent research.

6.1.1 Alternative representations of semantics

Beadle *et al.* used *Reduced Ordered Binary Decision Diagrams* (ROBDDs) [27] in a series of papers [21, 20, 22] to represent semantics of Boolean programs. ROBBD is indeed a canonical form of Boolean program i.e., it expresses complete knowledge on program behavior, and for a particular behavior and variable ordering, ROBDD is unique. Hence ROBDDs are useful for equivalence checking of two Boolean programs.

Note that ROBBDs are not suitable for non-Boolean domains while sampling semantics is a domain-independent data structure. Actually both ROBDD and sampling semantics store equivalent amount of information on program behavior if sampling semantics enumerates all combinations of program inputs. Otherwise, ROBDD cannot be created, while sampling semantics can.

Both sampling semantics and ROBBD are useful in modeling program behavior where the target behavior is known. However in applications like induction of a program for a robot controller, the optimal behavior is not known in advance.

Algorithm 6.1: Semantically Driven Initialization algorithm. Before running the algorithm first time, all terminals Φ_t from instruction set Φ are copied to P, i.e., $P = \Phi_t$. $\epsilon_1 \geq 0$ is similarity threshold, const is semantics of any constant program, Random$(\cdot)$ and Arity$(\cdot)$ are defined in Algorithm 3.1.

```
Require: $\Phi_t \subseteq P$
1: function SDI($P$, $\Phi_n$)
2:     repeat
3:         $r \leftarrow$ Random($\Phi_n$)
4:         for $i = 1..$Arity($r$) do
5:             $r_i \leftarrow$ Random($P$)    :: Assign child node
6:     until $\forall p \in P : \epsilon_1 \leq d(p, r) \wedge s(r) \neq$ const
7:     return $r$
```

Algorithm 6.2: Behavioral Initialization algorithm. $h \in \mathbb{N}_{\geq 1}$ is maximum height of the tree, $\epsilon_1 \geq 0$ is similarity threshold, Grow$(\cdot)$ and Full$(\cdot)$ are defined in Algorithm 3.1.

```
function BI($h$, $P$, $\Phi_t$, $\Phi_n$)
    repeat
        $r \leftarrow$ Grow($h$, $\Phi_t$, $\Phi_n$)    :: Or Full($h$, $\Phi_t$, $\Phi_n$) with 0.5 probability
    until $\forall p \in P : \epsilon_1 \leq d(p, r)$
    return $r$
```

To express semantics of a robot controller, Beadle *et al.* [21, 22] used a sequence of robot moves and orientations in an environment obtained by executing its program in loop a limited number of times. In other works semantics there is a trace of the robot (or its program).

Similar representations to the Beadle's one were used by Jackson [74, 75] and Galvan-Lopez *et al.* [55], however Jackson used solely a sequence of moves of a robot, while Galvan-Lopez *et al.* solely a sequence of orientations.

6.1.2 Initialization operators

Beadle *et al.* [21] proposed *Semantically Driven Initialization* (SDI). Originally, they used ROBDD and sequence of robot movements as representations of semantics, and their algorithm is specific to Boolean and controller domains. In Algorithm 6.1 we present the generalized version of SDI that could be used in other domains and with other representations of semantics. To Algorithm 6.1 be equivalent to the original one, $d(\cdot, \cdot)$ has to be a discrete metric from Eq. (4.1) and const be semantics of tautology and contradiction.

SDI is a trial-and-error approach that combines a randomly drawn instruction with the already existent in the population. If the created program is semantically distinct from all other programs in the population, it is accepted. Otherwise the algorithm makes a new attempt to create a program. Note that Algorithm 6.1 is not guaranteed to halt, thus in practical implementation it needs a limit on number of iterations of the loop in lines 2–6.[1]

SDI creates semantically more diverse populations than RHH (cf. Section 3.4), however the relative performance of SDI when compared to RHH is highly problem-dependent, and none of these methods is ultimately better.

Jackson [74] experimented with initialization operators that promote structural and behavioral diversity and showed that the former slightly and the latter noticeably improve GP's performance on a set of relatively easy problems. We present in Algorithm 6.2 Jackson's *Behavioral Initialization* (BI) operator from that work. It is actually a wrapper on RHH, that discards semantically duplicated programs. For $\epsilon_1 = 0$, BI is equivalent to RHH, for $\epsilon_1 > 0$ BI is effective (cf. Defintion 4.5).

[1] A fixed limit ceases the algorithm to be strictly effective, however it may be considered *effective in the limit* of iteration count approaching ∞. For clarity we do not distinguish these two terms. This comment actually applies to the most of algorithms presented in this chapter.

Algorithm 6.3: Semantic Tournament Selection algorithm. p is mate parent, $\epsilon_1 \geq 0$ is similarity threshold, μ comes from Definition 2.8, Random($\cdot$) is defined in Algorithm 3.1.

```
Require: $P = \{p_1, p_2, ..., p_{|P|}\}$
1: function STS($p$,$p_1$,$p_2$,...,$p_{|P|}$)
2:     $p' \leftarrow$ Random($P$)
3:     for $i = 1..\mu$ do
4:         $r \leftarrow$ Random($P$)
5:         if $f(r) > f(p') \wedge \epsilon_1 \leq d(r,p)$ then
6:             $p' \leftarrow r$
7:     return $p'$
```

Algorithm 6.4: Semantically Driven Mutation algorithm. $\epsilon_1 \geq 0$ is similarity threshold, TM($\cdot$) is defined in Algorithm 3.3.

```
1: function SDM($p$)
2:     repeat
3:         $p' \leftarrow$ TM($p$)
4:     until $\epsilon_1 \leq d(p', p)$
5:     return $p'$
```

Originally there is a limit on the number of iterations of loop in lines 2 – 4,[1] which is required for the algorithm to halt.

6.1.3 Selection operators

Galvan-Lopez *et al.* [55] proposed a 2-mate selection operator, called *Semantic Tournament Selection* (STS), presented in Algorithm 6.3. Given a first parent selected by TS (cf. Definition 2.8), STS samples the population μ times and selects the best of the μ sampled individuals that has semantics different than the first parent; μ is here an analog to tournament size in TS. If none of the sampled individuals has different semantics, a random one is returned. Note that if the random program p' drawn in line 2 is semantically equivalent to first parent p and not worse than all programs r drawn in line 4, STS produces a neutral second parent, thus STS is not effective.

The main advantage of STS is that its computation cost is nearly the same as TS, given that fitness values and semantics of individuals are already calculated (cf. lines 3 and 11 of Algorithm 2.1). The performance of GP employed with STS is statistically better than the performance of the canonical GP algorithm.

6.1.4 Mutation operators

Beadle *et al.* [22] proposed *Semantically Driven Mutation* (SDM). Originally, SDM was designed for Boolean domain and semantics represented by ROBDD, in Algorithm 6.4 we generalized SDM to more domains and representations of semantics by introducing semantic distance in comparison of semantics. The algorithm is equivalent to the original one when $\epsilon_1 = 1$ and $d(\cdot,\cdot)$ is a discrete metric from Eq. (4.1). SDM applies TM in a loop until the produced offspring is semantically more distant from its parent than a given threshold ϵ_1. For $\epsilon_1 > 0$, SDM is effective, for $\epsilon_1 = 0$ SDM is equivalent to TM (cf. Section 3.7). Note that the algorithm may not halt, hence implementation it requires a limit on number of iterations of the loop in lines 2 – 4.[1]

Experimental analysis indicated that GP equipped solely with SDM synthesizes more fit programs than combination of TX and TM. Moreover, SDM produces programs of similar size to those evolved using combination of TX and TM.

Algorithm 6.5: Semantic Aware Mutation (left) and Semantic Similarity-based Mutation (right) algorithms. ϵ_1, ϵ_2 are distance thresholds, $0 \leq \epsilon_1 \leq \epsilon_2$, RandomNode($\cdot$) and Replace($\cdot,\cdot,\cdot$) are defined in Algorithm 3.2. Φ_t, Φ_n and Grow($\cdot,\cdot,\cdot$) are defined in Algorithm 3.1

```
1: function SAM($p$)
2:     repeat
3:         $p' \leftarrow$ RandomNode($p$)
4:         $r \leftarrow$ Grow($h$,$\Phi_t$,$\Phi_n$)
5:     until $\epsilon_1 \leq d(r,p')$
6:     return Replace($p$,$p'$,$r$)
```

```
1: function SSM($p$)
2:     repeat
3:         $p' \leftarrow$ RandomNode($p$)
4:         $r \leftarrow$ Grow($h$,$\Phi_t$,$\Phi_n$)
5:     until $\epsilon_1 \leq d(r,p') \leq \epsilon_2$
6:     return Replace($p$,$p'$,$r$)
```

Algorithm 6.6: Semantically Driven Crossover algorithm. $\epsilon_1 \geq 0$ is similarity threshold, TX is defined in Algorithm 3.2.

```
1: function SDX($p_1$,$p_2$)
2:     repeat
3:         $p' \leftarrow$ TX($p_1$, $p_2$)
4:     until $\epsilon_1 \leq d(p',p_1) \wedge \epsilon_1 \leq d(p',p_2)$
5:     return $p'$
```

Quang *et al.* [123] *Semantic Aware Mutation* (SAM) and *Semantic Similarity-based Mutation* (SSM). SAM works like TM, except that a newly created random subprogram is accepted only if it is semantically more distant from the subprogram to be replaced than a given threshold ϵ_1. SSM accepts a new subprogram only if its distance is within a range $[\epsilon_1, \epsilon_2]$. Note that the original formulation of SAM involves only one iteration of the loop in lines 2 – 5, however to unify SAM with SSM we actually allow more iterations.[2] The second difference is in condition in line 5 of SSM, where we use non-strict inequalities ($\leq$), while the authors use two strict inequalities ($<$). Different types of inequalities in SSM and in SAM were probably not intended by the authors. To guarantee that both algorithms halt, a limit on a number of iterations of the loop in lines 2 – 5 is required. Neither SAM nor SSM is effective, because even if a random subprogram r is semantically distinct from the replaced one p', the distinction may not hold for entire offspring and its parent.

Experiments reported in [123] showed that SAM is inferior for GP performance when compared to TM, and SSM is superior to both TM and SAM. We believe that the main reason for much weaker results of SAM than SSM lies in the lack of repetitions in the original SAM algorithm, what gives SAM fewer chances (than in case of SSM) to meet the semantic distance condition

6.1.5 Crossover operators

Beadle *et al.* [20] proposed *Semantically Driven Crossover* (SDX). SDX is actually a trial-and-error wrapper on TX (cf. Section 3.6) that repeats TX application to the parents until the created offspring is semantically different from both of the parents. SDX is originally developed for evolution of Boolean programs and ROBDD semantics, however we present in Algorithm 6.6 a natural generalization of the algorithm that handles more domains and forms of semantics. To be consistent with the original formulation the comparison in line 5 has to involve $\epsilon_1 = 1$ and discrete metric from Eq. (4.1) as $d(\cdot,\cdot)$. For real domain ϵ_1 would be a small positive number and $d(\cdot,\cdot)$ be Euclidean metric. Given $\epsilon_1 > 0$ SDX is effective, for $\epsilon_1 = 0$ SDX is equivalent to TX.

The loop in lines 2 – 5 guarantees that the produced offspring is effective, however it may not halt. To overcome this drawback technical realization should impose a limit on the number of iterations.[1]If the limit exceeds, the neutral offspring is to be returned or crossover attempt is to be repeated with different parents, which is recommended approach by the authors.

[2] Originally SAM performed TM if the first attempt to create semantically different subprogam failed, which lowers the overall probability of accepting semantically equal subprogram, but not eliminate this case.

Algorithm 6.7: Semantic Aware Crossover (left) and Semantic Similarity-based Crossover (right) algorithms. ϵ_1, ϵ_2 are distance thresholds, $0 \leq \epsilon_1 \leq \epsilon_2$, RandomNode($\cdot$) and Replace($\cdot,\cdot,\cdot$) are defined in Algorithm 3.2.

```
1: function SAX($p_1$,$p_2$)
2:     repeat
3:         $p'_1 \leftarrow$ RandomNode($p_1$)
4:         $p'_2 \leftarrow$ RandomNode($p_2$)
5:     until $\epsilon_1 \leq d(p'_1, p'_2)$
6:     return Replace($p_1$,$p'_1$,$p'_2$)
```

```
1: function SSX($p_1$,$p_2$)
2:     repeat
3:         $p'_1 \leftarrow$ RandomNode($p_1$)
4:         $p'_2 \leftarrow$ RandomNode($p_2$)
5:     until $\epsilon_1 \leq d(p'_1, p'_2) \leq \epsilon_2$
6:     return Replace($p_1$,$p'_1$,$p'_2$)
```

Algorithm 6.8: The Most Semantic Similarity-based Crossover algorithm. c is number of candidate pairs of crossover points, $\epsilon_1 \geq 0$ is similarity threshold, RandomNode($\cdot$) and Replace($\cdot,\cdot,\cdot$) are defined in Algorithm 3.2.

```
function MSSX($p_1$,$p_2$)
    $\Gamma = \emptyset$
    repeat
        $p'_1 \leftarrow$ RandomNode($p_1$)
        $p'_2 \leftarrow$ RandomNode($p_2$)
        if $\epsilon_1 \leq d(p'_1, p'_2)$ then
            $\Gamma = \Gamma \cup \{(p'_1, p'_2)\}$
    until $|\Gamma| = c$
    $(p'_1, p'_2) \leftarrow \arg\min_{(p'_1,p'_2)\in\Gamma} d(p'_1, p'_2)$
    return Replace($p_1$,$p'_1$,$p'_2$)
```

Experiments verified that GP equipped with SDX on average achieves better fitness and produces smaller programs than with TX. Use of SDX accompanied with SDM results in better fitness than use of SDM solely, however they produce substantially bigger trees than TX and TM [22].

Quang *et al.* proposed in [178] two algorithms based on TX: *Semantic Aware Crossover* (SAX) and *Semantic Similarity-based Crossover* (SSX). In case of SAX they attempted to generalize the ideas that founded SDX to the domain of real function synthesis by introducing semantic distance in comparison of semantics. However their approach is not completely compatible with SDX, because in contrary to SDX, they compare semantics at the level of crossover points, while SDX does it at entire programs. Note that even when the semantics of exchanged subprograms are different, the entire crossover application may be neutral, hence SAX is not effective crossover operator. For $\epsilon_1 = 0$ SAX is equivalent to TX.

SSX operates similarly to SAX, however it involves a second threshold on semantic distance that bounds from up the distance between subprograms to be exchanged. The authors argue use of the upper bound by that not only the subprograms to be exchanged should be semantically different but also not too distant from each other to perform efficient GP search. SSX is not effective from the same reason as SAX. For $\epsilon_1 = 0$ and $\epsilon_2 = \infty$ SSX is equivalent to TX.

Note that technical realization of both algorithms requires a limit on number of iterations of loops in lines 2 – 5, to guarantee halting.

Performance reports on SAX and SSX found in the literature are divergent. Quang *et al.* in a series of papers [122, 177, 178] report that both SAX and SSX noticeably overcome TX, while in Krawiec *et al.* [87] we showed that the performances of SAX and TX are nearly the same, while the performance of SSX is noticeably worse. We [131] confirmed on a wider set of problem domains than previous papers that SAX is statistically better than TX. Moreover in the same paper we experimentally verified theoretical observation that SAX is not effective operator.

On the basis of SSX, Quang *et al.* [179] proposed *The Most Semantic Similarity-based Crossover* (MSSX). Their algorithm is trial-and-error approach that begins from drawing a set of pairs of candidates for crossover points, one point per pair from each parent. Pairs having semantics less

Algorithm 6.9: Semantic Control Crossover algorithm. c is number of candidate pairs of crossover points, $\alpha = \alpha_{\max} - (\alpha_{\max} - \alpha_{\min}) \times g/G$, where $\alpha_{\mathsf{min}}, \alpha_{\mathsf{max}} \in \mathbb{R}$ are bounds for scaling coefficient α, $\alpha_{\mathsf{min}} \leq \alpha_{\mathsf{max}}$, g is current generation number and G is total generations number, $R(\Gamma)$ is probability distribution with probability density function $\Pr((p'_1, p'_2) : (p'_1, p'_2) \in \Gamma) = \frac{d^{\alpha}(p'_1,p'_2)}{\sum_{(p''_1,p''_2)\in\Gamma} d^{\alpha}(p''_1,p''_2)}$, RandomNode$(\cdot)$ and Replace$(\cdot)$ are defined in Algorithm 3.2.

```
function SCX($p_1$,$p_2$)
    $\Gamma = \emptyset$
    repeat
        $p'_1 \leftarrow$ RandomNode($p_1$)
        $p'_2 \leftarrow$ RandomNode($p_2$)
        $\Gamma = \Gamma \cup \{(p'_1, p'_2)\}$
    until $|\Gamma| = c$
    $(p'_1, p'_2) \sim R(\Gamma)$    :: Select pair proportionally to scaled distance between nodes in this pair
    return Replace($p_1$,$p'_1$,$p'_2$)
```

distant than a threshold are discarded. Once the set is filled by c candidates,[3] MSSX chooses the pair with the smallest distance between semantics of subprograms to be exchanged. Note the loop in lines 3 – 8 may never end, so the limit on the number of iterations is required. Argumentation behind MSSX is similar to that behind SSX: exchange of semantically as similar as possible subprograms is beneficial, but not when the subprograms are semantically equal to each other, because then the computation effort is virtually lost. MSSX is not effective operator.

Quang *et al.* reported that GP equipped with MSSX finds significantly better solutions than GP equipped with TX or SSX on a series of symbolic regression benchmark problems.

Hara *et al.* [64] proposed *Semantic Control Crossover* (SCX) operator with similar mode of operation to MSSX, however with a fine-grained automatic shaping of probability distribution of accepting particular pairs of crossover point candidates. The distribution is tuned depending on the semantic distance within a candidate pair and a current stage of evolution. SCX is presented in Algorithm 6.9. Given two parent programs SCX first draws from them c pairs of candidates for crossover points, one point in each parent per pair. Then SCX draws from the set of candidates Γ according to probability distribution $R(\Gamma)$, defined in the caption of Algorithm 6.9, i.e., proportionally to scaled semantic distance between subprograms in each pair. $\alpha \in \mathbb{R}$ is scaling coefficient that controls trade-off between exploration and exploitation of a solution space. For $\alpha \geq 1$ there is significant bias towards exploration of far subprograms, for $0 < \alpha < 1$ the bias for exploration is compressed, for $\alpha = 0$ there is no preference at all[4] and for $\alpha < 0$ there is significant bias towards exploitation of semantically close subprograms. Value of α varies over evolutionary run in range $\langle \alpha_{\min}, \alpha_{\max} \rangle$, according to formula presented in the caption of Algorithm 6.9, i.e., initially promotes exploration, then gradually moves bias towards exploitation. SCX is not effective because all candidate pairs in Γ may consist of semantically equal subprograms.

Hara *et al.* evaluated SCX only on two relatively easy benchmarks and reported a slight improvement w.r.t. TX. However a deeper analysis of Hara's results suggests that SCX is very sensitive to values of $\alpha_{\min}$ and $\alpha_{\max}$ and on average performs as good as TX.

6.1.6 Other work

Castelli *et al.* [34] proposed changes to GP scheme that enable modeling and shaping of distribution of semantics in the population. The algorithm starts with a population of semantically distinct programs. Then, in every generation a desired distribution of semantics is modeled, depending on

[3] Actually the original formulation in [179, Alg. 2] misses the line where a counter is updated and does not explain this in text, so we hypothesize that the counter refers to the number of candidates, not just iterations of the loop.

[4] For the scope of SCX we assume that $0^0 = 1$ and $0/0 = 1$ to handle indeterminate forms in definition of $R(\Gamma)$. For $\alpha = 0$ SCX is equivalent to TX.

distribution and fitness values of semantics in the previous generation. Newly created offspring are accepted to join the population only if they match the desired semantics distribution or have fitness better then the best program found so far.

Castelli's approach lead to improvements in fitness obtained by GP, but also increased bloat. The authors proposed to explicitly find and delete introns in every generation, which did not significantly degraded the fitness-based performance of the approach, but decreased an average program size.

Jackson [75] conducted an analysis on structural, semantic (behavioral)[5] and fitness-based diversity in populations of GP. He found out that population initialization affects only a few first generations of GP algorithm, then the diversity of programs depends mainly on variation operators. For canonical GP employed only with TX the structural diversity typically increases in the initial generations, then becomes constant over time. On the other hand, semantic diversity drops rapidly and after few generations stabilizes at low levels. Dynamics of fitness-based diversity is problem-dependent. Jackson also confirmed experimentally that use of semantic diversity-promoting methods in initialization and crossover operators noticeably improves GP performance.

Vanneschi *et al.* [180] prepared an extensive survey of semantic methods in GP that briefly sums up recent papers. The authors divided semantic methods into those that care only on semantic diversity in a population, manipulate semantics indirectly e.g., by trial-and-error modifications of syntax, or directly i.e., by making modifications in syntaxthat influenceprogram's semantics in a certain way. Nevertheless the authors do not present specifics of the algorithms or comparison of them.

6.2 Works on Geometric Semantic Genetic Programming

Below we briefly characterize past works on geometric semantic genetic programming, mainly focusing on specific algorithms comply with or approximating principles of geometric operators.

6.2.1 Exact geometric variation operators

Moraglio *et al.* [115] proposed exact Semantic Geometric Mutation (SGM) and Semantic Geometric Crossover (SGX) for programs working in Boolean, classifier and real domains. Both operators are constructive approaches that build an exact geometric offspring 'on top' of the code of its parent(s) (cf. Definitions 5.6 and 5.7). SGM prepends a given parent program p with a domain dependent formula that moves its semantics in a semantic space:

$$p' = \begin{cases} p \vee p_x & \text{with}\,0.5\,\text{probability} \\ p \wedge \neg p_x & \text{otherwise} \end{cases} \quad \text{(Boolean domain)}$$

$$p' = p + \epsilon \times p_x \qquad \text{(real domain)}$$

where p' is an offspring, and p_x is a random subprogram. In the Boolean domain, p_x is a random minterm that incorporates all named variables in a considered set of instructions Φ. In the real domain, p_x is any random program created from Φ. To alleviate bias of (possibly) non-uniform distribution of programs to be created from Φ (cf. Section 3.10), p_x can be replaced by $(p_{x_1} - p_{x_2})$, where p_{x_1} and p_{x_2} are random programs created from Φ, i.e., from the same probability distribution.

[5] Notion 'behavior' in Jackson's works is equivalent to sampling semantics here.

SGM is a 1-geometric mutation (cf. Definition 5.6) in Boolean and classifier domains for Hamming distance as semantic distance, and r-geometric in real domain for any L_z metric as semantic distance, where r is proportional to ϵ.

SGX combines two given parents p_1, p_2 in a formula that returns a random point in the segment between their semantics:

$$\begin{aligned} p' &= (p_1 \wedge p_x) \vee (\neg p_x \wedge p_2) && \text{(Boolean domain)} \\ p' &= p_1 \times p_x + (1 - p_x) \times p_2 && \text{(real domain)} \end{aligned}$$

where p' is an offspring. p_x is a random program created from the set of instructions Φ. For real domain, if semantic distance is L_2 then p_x is constrained to a constant from $\langle 0, 1\rangle$ else if L_1 then p_x outputs values in range $\langle 0, 1\rangle$ for each fitness case, i.e., $\forall_{(in,out^*)\in F}\, 0 \leq p_x(in) \leq 1$.

SGX is geometric in Boolean and classifier domains for Hamming distance, and in the real domain for L_1 and L_2 distances.

Note that the construction of SGM and SGX assume that the set of instructions Φ contains instructions that allow constructing an appropriate formula, e.g., for SGX the formula that defines a point on a segment under given metric.

Unfortunately SGM and SGX may result in an offspring that is semantically equal to one of its parents. Therefore, both SGM and SGX are not effective.

These operators cause also substantial bloat by design. SGM adds an extra code piece to a parent, resulting in a bigger offspring each time. SGX combines entire parent programs, producing an offspring that is roughly two times greater than the average size of the parents. The formulas for average program size in a population after g generations of evolution using either SGM or SGX are

$$\begin{aligned} &\overline{p_0} + g \times (\overline{p_x} + c) && \text{(SGM)} \\ &2\overline{p_0} + (2^g - 2)(\overline{p_0} + \overline{p_x} + {}^{c}\!/_{2}) && \text{(SGX)} \end{aligned}$$

where $\overline{p_0}$ is the average program size in the initial population, $\overline{p_x}$ is the average size of p_x, c is the number of instructions required to combine p_x with the parent(s), i.e., a domain dependent constant, e.g., for SGM and real domain two instructions: + and ×. Note that the formula for SGX is exponential in g, what was also verified experimentally in [134].

The memory and computational time requirements for evolving and/or running bloated programs are challenging, what was already signaled by authors of the method. They proposed to use a domain-dependent program simplification procedure. Unfortunately problem of program simplification in general is NP-hard [44], which in turn makes precise simplification futile for large programs created by SGM and SGX.

The problem of bloat in SGX and SGM was also addressed by Vanneschi *et al.* [181], who formalized an obvious observation that SGM and SGX never modify programs once they have been created. The idea is to store in an offspring the pointer(s) to parent(s) instead of the entire parent(s). Thus one instance of parent's code may be used multiple times in multiple offspring, their offspring and so on. This, together with caching of parents' semantics lead to a memory- and time-efficient implementation of SGM and SGX.

The performance of SGX was compared to other algorithms for geometric semantic crossover operators (presented in subsequent sections) in our previous work [134], where SGX turns out to achieve the best results of all compared operators in evolution of Boolean programs, however in real domain it causes an entire population to converge to a single point in semantic space and in consequence underperforms.

Despite the challenges discussed above , SGX and SGM found use with success in symbolic regression, function prediction [31, 32, 33] and classification [193].

Algorithm 6.10: Locally Geometric Semantic Crossover algorithm. CommonRegion$(\cdot,\cdot)$ returns set of common nodes in the given programs, Midpoint$(\cdot)$ is defined by Eq. (6.1), RandomNode$(\cdot)$ and Replace$(\cdot,\cdot,\cdot)$ are defined in Algorithm 3.2, LibrarySearch$(\cdot)$ returns from a library a program as close as possible to the supplied semantics.

```
function LGX($p_1$, $p_2$)
    $c \leftarrow$ CommonRegion($p_1$, $p_2$)
    $p'_1, p'_2 \leftarrow$ RandomNode($c$)    :: Select nodes at the same locus in each parent
    $s_m \leftarrow$ Midpoint($p'_1$, $p'_2$)
    $p' \leftarrow$ LibrarySearch($s_D$)
    return Replace($p_1$, $p'_1$, $p'$)
```

6.2.2 Locally geometric semantic crossover

In our previous work [84] we proposed and later deeply investigated [85, 87] the *Locally Geometric Semantic Crossover* (LGX) that approximates the geometric recombination of parent programs at the level of homologously chosen subprograms.

Algorithm 6.10 presents the pseudocode of LGX. Given two parent programs p_1, p_2 LGX starts from calculating a syntactic *common region* of them [140], the tree constructed by simultaneously descending both parent trees from their roots until reaching nodes of different arities. Intuitively, the common region is the subset of program nodes that overlaps when program trees are superimposed on each other. Next, LGX uniformly picks two homologous nodes p'_1, p'_2 (at the same locus) in the common region, one in each parent. Then, a desired semantics s_D that is semantically geometric with respect to these subprograms, i.e., a point on the segment $s(p'_1)$, $s(p'_2)$, is calculated by domain-dependent Midpoint$(\cdot,\cdot)$ formula:

$$\begin{aligned} s_m &= (s(p'_1) \wedge s_x) \vee (\neg s_x \wedge s(p'_2)) && \text{(Boolean domain)} \\ s_m &= \tfrac{1}{2}(s(p'_1) + s(p'_2)) && \text{(real domain)} \end{aligned} \tag{6.1}$$

where s_x is a random semantics, such that the numbers of bits in s_m coming from each parent's semantic are the same concerning only bits that differ. Next, a library of known programs is searched for the program p' that minimizes the semantic distance to s_m. Finally, p' replaces p'_1 in p_1, producing so the offspring.

To produce the second offspring, we assume LGX is run with the parents swapped and the same nodes are picked from the common region.

Eq. (6.1) is intended to calculate the center of the segment between semantics of parents' subtrees, i.e., a point such that $d(s(p'_1), s_m) = d(s_m, s(p'_2))$. For the Boolean domain and Hamming distance there may be many points like that, or none at all, depending on the distance $d(p'_1, p'_2)$ is even or odd, respectively. For an odd distance, we allow violating the above equality by 1. In the real domain and for L_2 there is exactly one s_m that satisfies the equality; for other L_zs there may be more such points, and Eq. (6.1) produces one of them for any $z \geq 1$.

The source of the programs for the library is parameter of the method. It can be the set of all (sub)programs available in the current population, a precomputed set of programs, or mix of these two. The way how the library is constructed is not critical, however the number of the programs in the library increase the costs of searching, which may so become the main computation cost of the method.

LGX is not a geometric operator in strict sense for two reasons. First, it does not geometrically recombine parent programs; rather than that, it assumes that geometric recombination performed at deep parts of parents' trees would propagate trough program structures to the root, which is obviously not guaranteed due to, e.g., non-monotonous characteristics of instructions (see [86] for analysis). Second, in general the library is not guaranteed to contain a program of desired semantics, and even a slight mismatch between the desired semantics and its best match in the library may be amplified when propagated to the program tree root.

Algorithm 6.11: Krawiec & Lichocki Crossover algorithm. c is number of offspring candidates, MostGeometric$(\cdot,\cdot,\cdot)$ is given in Eq. (6.2), TX$(\cdot,\cdot)$ is defined in Algorithm 3.2.

```
function KLX($p_1$, $p_2$)
    $\Gamma \leftarrow \emptyset$
    repeat
        $\Gamma \leftarrow \Gamma \cup \{$ TX$(p_1, p_2)$ $\}$
    until $|\Gamma| = c$
    return MostGeometric$(\Gamma, p_1, p_2)$
```

LGX is not an effective operator, since library search in line 5 of Algorithm 6.10 is allowed to return a program having the same semantics as $s(p_1')$. Such a program, when put in place of p_1' in line 6, does not change semantics in any node up on the path to p_1's root.

Experimental analysis in [87] showed that fitness-based performance and generalization ability of LGX is superior to TX (cf. Section 3.6), homologous one-point crossover [140], and two semantic crossovers: SAX and SSX (cf. Section 6.1.5). However, the same analysis showed that LGX suffers from noticeable bloat and significantly higher computation cost than the other operators.

6.2.3 Approximately geometric semantic crossover

Krawiec and Lichocki [83] proposed a KLX operator to approximate geometric crossover in semantic space that combines principles of trial-and-error and brood selection (see [172]). In the original paper, the authors considered four variants that share common principles. Below we focus on only the one that achieved the best performance in that work.

KLX, presented in Algorithm 6.11, runs in a loop an other crossover operator, e.g., TX, and stores all generated candidate offspring in a set Γ (often called *brood* in analogous works in EC). Then KLX selects the candidate that is the closest to being geometric with respect to the parents, while possibly balancing the distances from them:

$$\mathsf{MostGeometric}(\Gamma, p_1, p_2) = \arg\min_{p' \in \Gamma} \big(\underbrace{d(p_1, p') + d(p', p_2)}_{\text{distance sum}} + \underbrace{|d(p_1, p') - d(p', p_2)|}_{\text{penalty}} \big) \tag{6.2}$$

This formula is composed of two parts. The first of them is the sum of semantic distances between the parents and the candidate offspring p'. For an offspring lying in the segment between the parents, that distance is equal to the semantic distance between the parents. The closer $s(p')$ is to the segment, the smaller this term. The second term is the penalty for producing an offspring that is non-equidistant from the parents; for an offspring located in center of the segment, it amounts to 0.

KLX is not effective operator, since the criterion of offspring selection does not deny candidate offspring that are semantically equal to any of their parents. This issue was addressed in our previous work [131], where we proposed to replace the MostGeometric$(\cdot,\cdot,\cdot)$ call in Algorithm 6.11 with:

$$\mathsf{MostGeometric}^{+}(\Gamma, p_1, p_2) = \begin{cases} \mathsf{MostGeometric}(\Gamma', p_1, p_2) & \Gamma' \neq \emptyset \\ \mathsf{MostGeometric}(\Gamma, p_1, p_2) & \Gamma' = \emptyset \end{cases} \tag{6.3}$$
$$\text{where } \Gamma' = \{p' : p' \in \Gamma, d(p_1, p') > \epsilon, d(p', p_2) > \epsilon\}$$

where $\epsilon > 0$ is a threshold on semantic distance between a candidate and the parent. Eq. (6.3) discards from the set of candidates those that are semantically equal to any parent, making the choice of the offspring effective. However if all candidates in Γ are neutral, it falls back to Eq. (6.2), which is equivalent to a random selection of one of the offspring candidates. KLX augmented with Eq. (6.3)will be referred to as KLX^{+} in the following.

The results presented by Krawiec *et al.* [83] show that KLX performs as well as TX. This was later confirmed in our study [134], where we also identified the main cause for it: on a typical Boolean problem, nearly 90% of offspring produced by KLX are neutral. On the other hand, less than 0.1% of offspring produced by KLX$^+$ is neutral in an experiment reported in [131], and in consequence KLX$^+$ indeed performs significantly better than KLX, TX and TX augmented to be effective.

6.2.4 Geometric crossovers for syntactic metrics

Moraglio *et al.* in their early works [117, 118] analyzed properties of geometric operators defined for syntactic metrics of program code. Those studies do not refer to semantics and provide the equivalents of Definitions 5.6 and 5.7 for a set of programs $\mathcal{P}$ using syntactic metrics instead of semantic ones.

In [117] they proved that TX is not geometric under any metric, and that homologous crossover [140] is not geometric under any graphic metrics.[6] They defined structural Hamming distance for abstract syntax trees, proved its metric properties and showed that homologous crossover is geometric under that metric.

In [118] Moraglio *et al.* carried out a similar analysis for edit-distance and linear programs. They proved that all homologous crossovers for linear programs are geometric under edit distance and derived bounds on edit distance between parents and offspring produced by homologous crossover.

[6] Graphic metric is metric defined for pairs of graphs.

Chapter 7

Competent Algorithms for Geometric Semantic Genetic Programming

We present general vision followed by precise description of competent algorithms for GP operators for each stage of evolution: initialization, selection, mutation and crossover that combine features of effective (cf. Section 4.2) and geometric (cf. Section 5.3) operators.

7.1 The vision

Semantic GP is a relatively recent thread in genetic programming. The studies conducted in this area usually focus on individual components of evolutionary algorithm, most commonly on variation operators. Most attention is usually paid to performance (program error), with the other qualities treated as of secondary importance. Though many of the results delivered in course of such efforts are valuable, the past research seems a bit fragmented and lacking an overall vision.

In response to that state of affairs, in this thesis we come up with the concept of *competent algorithms for geometric semantic* GP. By competent algorithms we mean algorithms implementing particular operators of evolutionary workflow: initialization, selection, mutation and crossover operators that:

- Together constitute a complete and consistent algorithmic suite of GP operators (cf. research goals in Section 1.3),
- Address (possibly all) challenges identified in SGP: particularly neutrality, poor generalization of programs and bloat (cf. Sections 3.10 and 4.2),
- Take into account possibly many theoretical consequences of the geometric structure of semantic space (e.g., the conic fitness landscape in Section 5.2),
- Follow GSGP's definitions from Section 5.3 and directly manipulate a convex hull of a population,
- Explicitly decompose big program induction problems into smaller ones,
- Are easily applicable to different domains, e.g., in not making assumptions about the nature of a domain and programming language of consideration – unlike, e.g., the exact geometric crossover (cf. Section 6.2.1) that requires the semantic 'mixing' of parents to be expressible in a given programming language,
- And, as a consequence of the above, are robust and solve a wide range of problems quickly, reliably and accurately (cf. Section 1.4).

Algorithm 7.1: Competent Initialization algorithm. SDI$(\cdot,\cdot,\cdot)$ is defined in Algorithm 6.1.

```
Require: $\Phi_t \subseteq P$
1: function CI($P$, $\Phi_t$, $\Phi_n$)
2:     repeat
3:         $p' \leftarrow$ SDI($P, \Phi_t, \Phi_n$)
4:     until $s(p') \notin C(\{s(p) : p \in P\})$    :: $p'$ is not in the convex hull of the population
5:     return $p'$
```

By designing such competent algorithms and using them together, we hope to substantially boost the performance of semantic GP methods. We are particularly interested in developing a competent variant of GSGP where *all* abovementioned components are competent in the above sense. We begin from presenting competent algorithms for population initialization and mate selection. Next, we introduce a helper algorithm for program inversion that is used by competent variation operators presented in the subsequent sections: mutation and crossover. Then, we discuss the design of the operators and propose a way to efficiently store and search a library of programs that is used by the variation operators. Finally, we show how the use of the proposed variation operators influences fitness landscape.

7.2 Competent Initialization

Algorithm 7.1 presents *Competent Initialization* (CI). The algorithm is based on semantically driven initialization (Algorithm 6.1). The key novelty of CI is that it calls SDI in a loop and accepts the returned program p' if its semantics $s(p')$ is not a part of a convex hull of the population being initialized (as opposed to SDI, which was interested in semantic uniqueness only). In other words, CI in each call returns a program that expands the convex hull of the population in an attempt to include the target t in the hull, because as we showed in Theorem 5.9 this is necessary condition for $n \geq 2$-ary geometric variation operators to reach t. CI inherits SDI's requirement to include all terminals in the population before calling the CI function for the first time, i.e., $P = \Phi_t$

A semantics $s(p')$ of a new candidate program p' belongs to the L_1-convex hull of the population if $s(p')$ can be expressed as a linear combination of semantics present in the population. Let us arrange the semantics of programs in a population into a matrix, where $s_{i,j}$ is value of semantics of ith program in the population for the jth fitness case. Then, $s(p')$ belongs to the L_1-convex hull of the population iff system of equations

$$\begin{bmatrix} s_{1,1} & s_{2,1} & \cdots & s_{|P|,1} \\ s_{1,2} & s_{2,2} & \cdots & s_{|P|,2} \\ \vdots & \vdots & \ddots & \vdots \\ s_{1,|F|} & s_{2,|F|} & \cdots & s_{|P|,|F|} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{|P|} \end{bmatrix} = s(p')$$

has solution(s) for $\forall x_i \in \langle 0, 1 \rangle$. This can be verified, e.g., by means of Gaussian elimination [57, Ch 3] or linear programming [187]. The implementation involved in an experiment use the latter one.

For L_2-convex hull, an additional equation that ensures the combination is convex has to be added to that system:

$$\sum_i x_i = 1.$$

Note that the loop in lines 2 – 4 of Algorithm 7.1 is not guaranteed to halt. SDI may be unable to produce a program that expands the convex hull of the population. Another possibility is that the convex hull may already incorporate the entire semantic space, i.e., $C(\{s(p) : p \in P\}) = \mathcal{S}$, in which case it cannot be further expanded. A natural choice to handle the first case is to limit

Algorithm 7.2: Competent Tournament Selection algorithm. p is mate parent, μ comes from Definition 2.8, Random($\cdot$) is defined in Algorithm 3.1.

```
Require: $P = \{p_1, p_2, ..., p_{|P|}\}$
1: function CTS($p$,$p_1$,$p_2$,...,$p_{|P|}$)
2:     $\Gamma = \emptyset$
3:     repeat
4:         $\Gamma \leftarrow \Gamma \cup \{$ Random($P \setminus \Gamma$) $\}$
5:     until $|\Gamma| = \mu$
6:     return $\arg\min_{p' \in \Gamma} \frac{\overbrace{d(s(p'),t)}^{\text{distance to target}}}{\underbrace{d(p',p)}_{\text{distance to mate program}}} \times \underbrace{(1 + |d(s(p'),t) - d(s(p),t)|)}_{\text{penalty for unequal distances of programs to target}}$
```

the number of iterations of the loop. Concerning the second case, one needs to verify whether $C(\{s(p) : p \in P\}) = \mathcal{S}$ holds, and if that happens, accept any program subsequently created by SDI.

Because all programs initialized by CI come from SDI and SDI is effective in sense of Definition 4.5, CI is effective too.

CI strives to include the target t in the convex hull by expanding it to cover as much as possible semantic space. This does not guarantee that t gets included in the convex hull in a certain fixed number of applications. However, in the limit of population size (and number of applications) approaching infinity, t gets included in the convex hull. Hence, CI can be considered geometric in the limit or ∞-geometric (see Definition 5.4).

7.3 Competent Selection

Algorithm 7.2 presents *Competent Tournament Selection* (CTS), which is a 2-mate selection operator based on tournament selection (cf. Definition 2.8). CTS is motivated by the desired properties of geometric selection defined in Definition 5.5. CTS samples uniformly without repetition μ candidate programs from the population, where μ is tournament size. Next, from the set of candidates it selects and returns the one that minimizes the formula in line 6.

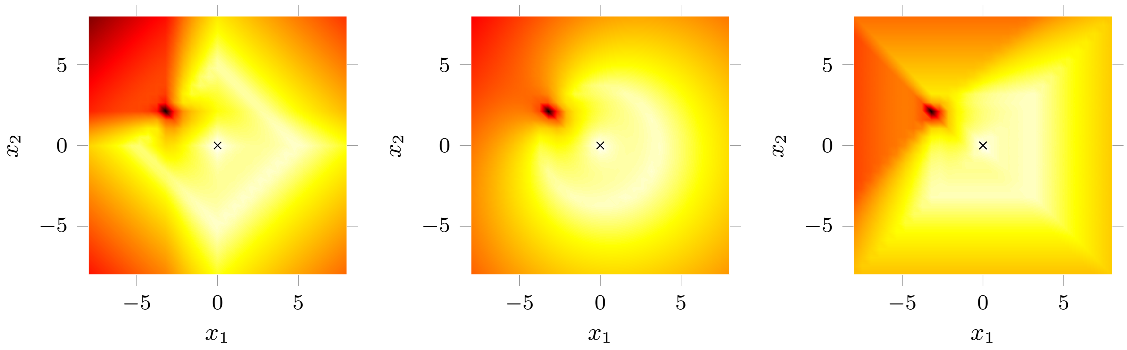
This formula involves three terms:

- The distance of a candidate p' to the target t – to be minimized to impose selection pressure towards better solutions,
- The distance of a candidate p' to the mate program p – to be maximized to penalize neutral candidates and ensure effectiveness of selection (cf. Definition 4.6),
- The penalty for selecting a candidate p' with a different distance to the target t than the mate program p – to prefer candidates in a certain perimeter of t, i.e., so that when p' is selected, t is located close to the center of the convex hull of $s(p)$ and $s(p')$.

In Figure 7.1, we provide visualizations of the formula for L_1, L_2, L_∞ semantic distances in the $\mathbb{R}^2$ semantic space. The formula attains the minimal value of 0 for the candidate having target semantics t, which is an effect of bias towards good candidates; infinity for semantics equal to the mate program $s(p)$; and 2 for semantics equidistant to $s(p)$ and t; and $1/2$ for semantics with the same distance to t as $s(p)$ but on the opposite side of t, i.e., $d(p,p') = 2d(s(p),t)$.

Due to probabilistic choice of candidates, inclusion of t in the selected programs' convex hull cannot be guaranteed and CTS is approximately geometric in the sense of Definition 5.5.

From the other perspective, CTS accepts a neutral candidate iff all the candidates are semantically equal to p, i.e., when the formula in line 6 is ∞ for all the candidates.



(a) $d \equiv L_1$ **(b)** $d \equiv L_2$ **(c)** $d \equiv L_\infty$

Figure 7.1: Spatial characteristics of selection formula in CTS under L_1, L_2, L_∞ semantic distances in $\mathbb{R}^2$, $t = (0, 0)$ (denoted by × in plots), mate program semantics $s(p) = (-3, 2)$. White is the minimum of the formula, dark red is the maximum (in the considered range), black is ∞.

7.4 Semantic Backpropagation for competent variation operators

Competent variation operators introduced in the subsequent sections require the concept of desired semantics and a related algorithm. We present them below.

Definition 7.1. A *desired semantics* is a tuple D of $|F|$ sets D_i corresponding to particular fitness cases in F, where each D_i contains the values from a set of program outputs O, i.e., $D = (D_1, D_2, ..., D_{|F|})$, $\forall_{i=1..|F|} D_i \subseteq O$.

Desired semantics is a generalization of sampling semantics in that it stores multiple values for each fitness case. Desired semantics $D = (D_1, D_2, ..., D_{|F|})$ is a compact equivalent (and implicit representation) of a set of semantics resulting from the Cartesian product of all D_is, i.e., $\Pi_{i=1..|F|} D_i$. For instance, $D = (\{2, 3\}, \{1, 7\})$ is equivalent to the set of four semantics $\{(2, 1), (2, 7), (3, 1), (3, 7)\}$. A single semantics s is equivalent to a desired semantics, where each D_i contains corresponding component of s, i.e., $\forall_{i=1..|F|} D_i = \{s_i\}$. In the following, the transformations of semantics from/to desired semantics are implicit. Note that D_is are allowed to be empty, which will come in handy in the following.

We term this formal object *desired* semantics because in the following it will express a set of expected/preferred behaviors of a program.

We say that a desired semantics D is matched by a program p if semantics $s(p)$ is in the set of sampling semantics represented by D, i.e., $s(p) \in \Pi_{i=1..|F|} D_i$.

To proceed, we need to extend Definition 4.4 of semantic distance for desired semantics. Let D^1 and D^2 be desired semantics. Then the distance $d_D(\cdot, \cdot)$ between them is:

$$d_D(D^1, D^2) = \min_{s_1 \in \Pi_{j \in J} D^1_j,\, s_2 \in \Pi_{j \in J} D^2_j} d(s_1, s_2) \tag{7.1}$$
$$J = \{i = 1..|F| : D^1_i \neq \emptyset \wedge D^2_i \neq \emptyset\}.$$

d_D is thus the minimal distance between all pairs of sampling semantics defined by D^1 and D^2, where D^1_i and D^2_i are omitted from the Cartesian product for each i, where D^1_i or D^2_i is empty. If all D^1_i and D^2_i are omitted, the semantic distance is unknown. For D^1 and D^2 representing a single semantics, $d_D(\cdot, \cdot) \equiv d(\cdot, \cdot)$, hence we drop subscript $_D$ in d_D and implicitly use the adequate formula in the following.

Algorithm 7.3: Semantic Backpropagation algorithm. D is desired semantics of program p, r is the node in p to calculate desired semantic for. Child(a, r) returns the child of node a on the path from a to r and Pos(a, r) returns the child's position in the list of arguments of a. Invert(a, k, o) is defined in Table 7.1. $\mathbb{D}$ is entire domain, e.g., $\mathbb{R}$ or $\mathbb{B} = \{0, 1\}$ for real or Boolean domains, respectively.

```
function SemanticBackpropagation($D$, $p$, $r$)
    foreach $D_i \in D$ do    :: For each fitness case
        $a \leftarrow p$
        while $a \neq r \wedge D_i \neq \emptyset \wedge D_i \neq \mathbb{D}$ do
            $k \leftarrow$ Pos$(a, r)$
            $D'_i \leftarrow \emptyset$
            foreach $o \in D_i$ do
                $D'_i \leftarrow D'_i \cup$ Invert$(a, k, o)$
            $D_i \leftarrow D'_i$
            $a \leftarrow$ Child$(a, r)$
    return $D$
```

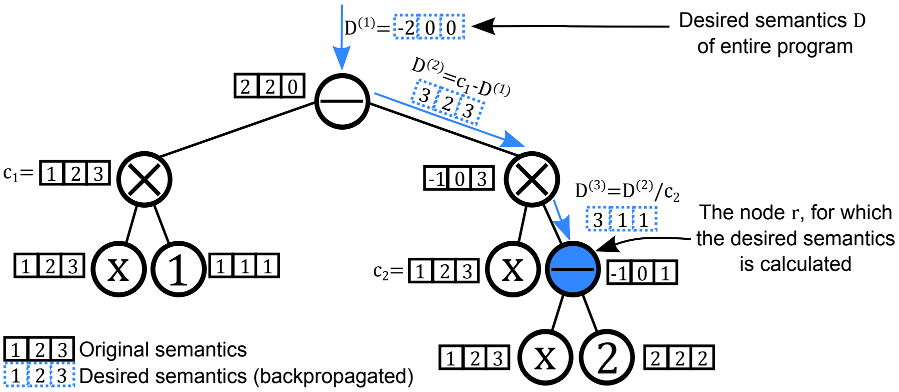


Figure 7.2: Example of semantic backpropagation. A desired semantics of entire program (blue dotted one) is propagated backwards through the path to the designated (blue) node. Black vectors are current semantics of respective subprograms.

Desired semantics is the key formalism used by *Semantic Backpropagation* presented in Algorithm 7.3. Semantic backpropagation was first introduced independently in [86] and [186], then unified and augmented in our joint study [133]. The goal of semantic backpropagation is to determine, given the desired semantics of a given program and the designated node r in this program, the desired semantics D for r such that, when a subprogram matching D replaces r, the entire program will match the program's desired semantics. The rules of conduct of semantic backpropagation are based on the idea of inversion of program execution.

Semantic backpropagation traverses, for each fitness case separately, the path of program nodes (instructions) from the root node to the designated node r. In each node a, where the desired output is o, semantic backpropagation inverts the execution of a w.r.t. the next node c_1 in the path toward r. Technically, an argument of a is calculated using inverted instruction a^{-1} for which a returns o, i.e., $o = a(c_1, c_2, ..., c_n) \implies c_1 = a^{-1}(o, c_2, ...c_n)$, where c_is are outputs returned by children nodes. The algorithm terminates when r is reached.

An exemplary run of semantic backpropagation is presented in Figure 7.2. The input to the algorithm is desired semantics $D^{(1)} = (\{-2\}, \{0\}, \{0\})$, program $p = x \times 1 - x \times (x - 2)$, and the node r for which the desired semantics is to be calculated. The first component of the semantics (-2) is propagated to the right child of root node '$-$' by executing inverted instruction $3 = 1 - (-2)$, where 1 is the first component of semantics of the left child. Next, 3 is propagated trough '$\times$' node to its right child by executing $3 = 3/1$. Finally, 3 becomes the first component of the desired semantics for r. These steps are repeated for each fitness case.

Table 7.1: Definition of Invert(a, k, o) for real and Boolean domains. For a subprogram a (an instruction node with one or two children that returned c_1 and c_2) the formulas determine the desired value for the first ($k = 1$, center column) and the second ($k = 2$, right column) argument, given the desired value o of the whole subprogram a. For clarity we omitted set symbols where there is only a one value.

Subprogram a	$\mathsf{Invert}(a, 1, o)$	$\mathsf{Invert}(a, 2, o)$
	Real domain	
$c_1 + c_2$	$o - c_2$	$o - c_1$
$c_1 - c_2$	$o + c_2$	$c_1 - o$
$c_1 \times c_2$	$\begin{cases} o/c_2 & c_2 \neq 0 \\ \mathbb{R} & c_2 = 0 \wedge o = 0 \\ \emptyset & c_2 = 0 \wedge o \neq 0 \end{cases}$	$\begin{cases} o/c_1 & c_1 \neq 0 \\ \mathbb{R} & c_1 = 0 \wedge o = 0 \\ \emptyset & c_1 = 0 \wedge o \neq 0 \end{cases}$
c_1/c_2	$\begin{cases} o \times c_2 & c_2 \neq \pm\infty \\ \mathbb{R} & c_2 = \pm\infty \wedge o = 0 \\ \emptyset & c_2 = \pm\infty \wedge o \neq 0 \end{cases}$	$\begin{cases} c_1/o & c_1 \neq 0 \\ \mathbb{R} & c_1 = 0 \wedge o = 0 \\ \emptyset & c_1 = 0 \wedge o \neq 0 \end{cases}$
$\exp(c_1)$	$\begin{cases} \log o & o \geq 0 \\ \emptyset & \text{otherwise} \end{cases}$	—
$\log \lvert c_1 \rvert$	$\{-e^o, e^o\}$	—
$\sin c_1$	$\begin{cases} \{\arcsin o - 2\pi, \arcsin o\} & \lvert o \rvert \leq 1 \\ \emptyset & \text{otherwise} \end{cases}$	—
$\cos c_1$	$\begin{cases} \{\arccos o - 2\pi, \arccos o\} & \lvert o \rvert \leq 1 \\ \emptyset & \text{otherwise} \end{cases}$	—
	Boolean domain	
$\text{not } c_1$	$\text{not } o$	—
$c_1 \text{ and } c_2$	$\begin{cases} o & c_2 \\ \mathbb{B} & \text{not } c_2 \text{ and not } o \\ \emptyset & \text{not } c_2 \text{ and } o \end{cases}$	$\begin{cases} o & c_1 \\ \mathbb{B} & \text{not } c_1 \text{ and not } o \\ \emptyset & \text{not } c_1 \text{ and } o \end{cases}$
$c_1 \text{ or } c_2$	$\begin{cases} o & \text{not } c_2 \\ \mathbb{B} & c_2 \text{ and } o \\ \emptyset & c_2 \text{ and not } o \end{cases}$	$\begin{cases} o & \text{not } c_1 \\ \mathbb{B} & c_1 \text{ and } o \\ \emptyset & c_1 \text{ and not } o \end{cases}$
$c_1 \text{ nand } c_2$	$\begin{cases} \text{not } o & c_2 \\ \mathbb{B} & \text{not } c_2 \text{ and } o \\ \emptyset & \text{not } c_2 \text{ and not } o \end{cases}$	$\begin{cases} \text{not } o & c_1 \\ \mathbb{B} & \text{not } c_1 \text{ and } o \\ \emptyset & \text{not } c_1 \text{ and not } o \end{cases}$
$c_1 \text{ nor } c_2$	$\begin{cases} \text{not } o & \text{not } c_2 \\ \mathbb{B} & c_2 \text{ and not } o \\ \emptyset & c_2 \text{ and } o \end{cases}$	$\begin{cases} \text{not } o & \text{not } c_1 \\ \mathbb{B} & c_1 \text{ and not } o \\ \emptyset & c_1 \text{ and } o \end{cases}$
$c_1 \text{ xor } c_2$	$c_2 \text{ xor } o$	$c_1 \text{ xor } o$

Algorithm 7.4: Competent Mutation algorithm. t is target, p is parent. OracleGet$(D, D_{\varnothing})$ returns a program that matches D and does not match $D_{\varnothing}$, RandomNode$(\cdot)$ and Replace$(\cdot,\cdot,\cdot)$ are defined in Algorithm 3.2, SemanticBackpropagation$(\cdot,\cdot,\cdot)$ comes from Algorithm 7.3.

```
function CM($t$, $p$)
    $a \leftarrow$ RandomNode($p$)
    $D \leftarrow$ SemanticBackpropagation($t, p, a$)
    $D_{\varnothing} \leftarrow$ SemanticBackpropagation($s(p), p, a$)
    $p' \leftarrow$ OracleGet($D, D_{\varnothing}$)
    return Replace($p, a, p'$)
```

Invert(a, k, o) performs an inversion of a single instruction a, with respect to its kth child node and the desired output o. This procedure is domain-dependent. In Table 7.1 we present the definition of this function for commonly used instructions in the real and Boolean domains. In general there are four cases that have to handled by Invert(a, k, o).

- The instruction a is bijection, i.e., one-to-one correspondence. In this case the instruction is fully invertible and desired value of the instruction's argument can be unambiguously determined depending on the output and other arguments, thus Invert$(\cdot,\cdot,\cdot)$ returns a single value. Examples of instructions are addition, subtraction, negation and exclusive or.
- a is non-injective instruction. Instructions like that map many input values to the same output, hence their inversion is ambiguous. There may be finite or infinite number of possible inversions. For instance, absolute value $|\cdot|$ results the same output when supplied with two opposite inputs, however $\sin(x)$ has infinite number of inversions in form $\arcsin(o + 2k\pi)$, where $k \in \mathbb{Z}$.
- Other kind of ambiguity comes from instruction arguments that have no impact on the instruction's output. Examples are multiplication by 0 and Boolean 'and' with 0, where the value of the second argument does not matter. In this case Invert$(\cdot,\cdot,\cdot)$ returns the entire domain $\mathbb{D}$, e.g., $\mathbb{R}$ for real domain instructions or $\mathbb{B}$ for the Boolean ones.
- a is a non-surjective instruction. This is the case, where exist values in the instruction's codomain that cannot be produced by that instruction for any input from its domain. If such a value is requested for the instruction's output, the inversion results in inconsistence. Invert$(\cdot,\cdot,\cdot)$ handles this situation by returning $\emptyset$.

It should be clear by now that instruction inversion is contextual: it depends on the value of the requested output and the values of other arguments, which in turn depend on the other parts of the program (in the sibling nodes of the nodes in the considered path). Hence, all above mentioned cases may co-occur in the same instruction, for different values from instruction's domain and codomain.

7.5 Competent Mutation

In Algorithm 7.4 we present *Competent Mutation* (CM), the algorithm inspired by Random Desired Operator (RDO) [133, 186, 185]. Both CM and RDO are based on ideas of program inversion and geometric mutation (cf. Definition 5.6). The main difference between them is that CM is effective operator, while RDO is not.

Given one parent p and the target t, CM starts with picking a random node a in p. Then, it calls SemanticBackpropagation(t, p, a), i.e., uses semantic backpropagation to propagate t to a resulting in desired semantics D. Next, semantic backpropagation is called again to propagate p's semantics $s(p)$ to a, resulting in forbidden semantics $D_{\varnothing}$, i.e., semantics that when put in a would result in no change of $s(p)$ (technically D and $D_{\varnothing}$ are the same structures). Then, an oracle is queried for (sub)program p' that matches D and does not match $D_\varnothing$. Finally, p' replaces a in p to create an offspring.

Oracle is a component that returns a program that meets the constraints imposed by D while not meeting the constraints specified by $D_\varnothing$. In general, this requires solving a separate program induction problem as specified in Section 3.3, the only difference being that the problem is this time defined by desired semantics rather than sampling semantics. In this sense, this approach involves problem decomposition and defines a 'local' subproblem, as declared in Section 7.1. There are several motivations for which we expect solving such subproblems to be easy. It is reasonable to assume that computational cost of solving program induction problem monotonically increases w.r.t. program size. Also, desired semantics represents a set of semantics, not just a single target, and finding a program with semantics in such a set should be in general easier than finding a program with a specific semantics. These observations lead to hypothesis that the problem to be solved by the oracle is on average easier than the original program induction problem, in particular that there are more programs solving such a problem and they are shorter than for the original problem.

Technically, the oracle can be realized by performing a separate GP, or searching a library of known programs. The first option is computationally demanding. On the other hand, library-based approach constitutes fair trade-off between computational costs and quality of the result. Note that none of the options guarantees solving the problem optimally, i.e., the program returned by the oracle is not guaranteed to meet the constraints specified by D. Details of the library-based approach chosen for use in experiment are presented in Section 7.7.

Given a perfect oracle and desired semantics D, where $\forall_{i=1..|F|} D_i \neq \emptyset$, CM is r-geometric, where $r = d(s(p), t)$. However, because the oracle is not guaranteed to return a program that matches the constraints, CM is expected to produce offspring in distance close to $d(s(p), t)$, where the divergence from that value is related to an error conducted by the oracle.

Because the oracle returns a program that is not guaranteed to perfectly match D, such a program may cause CM to produce neutral offspring. For this reason we supply the oracle with forbidden semantics $D_\varnothing$ and so that programs that match $D_\varnothing$ can be ignored (and never returned by it). Given technical realization of the oracle perfectly discards all programs matching $D_\varnothing$, CM is unable to produce neutral offspring, thus CM is effective.

Note that current semantics $s(a)$ of node a cannot be used in place of forbidden semantics $D_\varnothing$, because there may be more semantics (different to $s(a)$) that when put in a may result in the same semantics $s(p)$ of the entire program p. In turn by backpropagating current semantics $s(p)$ of parent p to a, $D_\varnothing$ is obtained that represents all semantics that when put in $a$do not change $s(p)$.

7.6 Competent Crossover

Algorithm 7.5 presents *Competent Crossover* (CX), an operator based on our previously published operator Approximately Geometric Semantic Crossover (AGX) [133, 86]. Both CX and AGX are approximately geometric, but CX is also effective as formalized in Definitions 5.7 and 4.7, respectively.

The main idea behind CX is to calculate the midpoint between semantics of parents p_1 and p_2, using the domain-dependent Eq. (6.1) — the same formula as for LGX, hence the respective discussion applies. Next, a node a is picked from p_1 and the midpoint semantics is backpropagated in p_1 to a, resulting in desired semantics D. Then, the current semantics of p_1 and p_2 are backpropagated to a, which we denote by $D^1_\varnothing$ and $D^2_\varnothing$, respectively, and treat as forbidden semantics. Next, an oracle is queried for program p' that matches D and does not match both $D^1_\varnothing$ and $D^2_\varnothing$. Finally, p' replaces the subprogram rooted at a in p_1, producing the offspring. The second offspring can be created by swapping parents.

Algorithm 7.5: Competent Crossover algorithm. p_1, p_2 are parents. Midpoint$(\cdot,\cdot)$ is defined in Eq. (6.1), RandomNode$(\cdot)$ and Replace$(\cdot,\cdot,\cdot)$ are defined in Algorithm 3.2, SemanticBackpropagation$(\cdot,\cdot,\cdot)$ in Algorithm 7.3, OracleGet$(\cdot,\cdot)$ in Algorithm 7.4.

```
function CX($p_1$, $p_2$)
    $s_m \leftarrow$ Midpoint($p_1, p_2$)
    $a \leftarrow$ RandomNode($p_1$)
    $D \leftarrow$ SemanticBackpropagation($s_m, p_1, a$)
    $D^1_\varnothing \leftarrow$ SemanticBackpropagation($s(p_1), p_1, a$)
    $D^2_\varnothing \leftarrow$ SemanticBackpropagation($s(p_2), p_1, a$)
    $p' \leftarrow$ OracleGet($D, D^1_\varnothing, D^2_\varnothing$)
    return Replace($p_1, a, p'$)
```

CX, similarly to CM, decomposes the original program induction problem and solves the new one using an oracle. For consistency, we use for CX the same technical realization of the oracle as for CM, detailed in Section 7.7.

Given a perfect oracle and desired semantics D, where $\forall_{i=1..|F|} D_i \neq \emptyset$, CX is geometric crossover, i.e., it is guaranteed to produce offspring, whose semantics lies in the midpoint of the segment between parents. However the imperfect match of desired semantics and the program returned by the technical realization of the oracle leads to an approximately geometric result of application.

The oracle in CX, like in CM, is supplied with forbidden semantics of parents to protect the oracle from returning a program that results in a neutral offspring.

7.7 Implementing oracle by library search

We technically realize the oracle by searching a library of prepared in advance programs for the one with semantics as close as possible to the given desired semantics D and not matching the forbidden semantics $D_\varnothing$. Formally,

$$\text{LibrarySearch}(D, D^1_\varnothing, D^2_\varnothing, ...) = \operatorname*{arg\,min}_{p\in L'} d(s(p), D) \tag{7.2}$$

$$L' = \{p : p \in L, d(s(p), D^1_\varnothing) \geq \epsilon, d(s(p), D^2_\varnothing) \geq \epsilon, ...\} \tag{7.3}$$

where $L \subseteq \mathcal{P}$ is a library, D is desired semantics, $D^1_\varnothing, D^2_\varnothing, ...$ are forbidden semantics, $\epsilon \geq 0$ is similarity threshold. For the Boolean domain $\epsilon = 1$, for the real one ϵ is a small value that represents arithmetic precision in a particular system. Setting $\epsilon = 0$ disables verification of forbidden semantics.

To provide even closer match of D, we define a function that calculates a constant program that minimize error w.r.t. D and does not match $D^1_\varnothing, D^2_\varnothing, ...$:

$$\text{OptimalConstant}(D, D^1_\varnothing, D^2_\varnothing, ...) = \operatorname*{arg\,min}_{c\in\mathbb{D}'} d([c], D) \tag{7.4}$$

$$\mathbb{D}' = \{c : c \in \mathbb{D}, d([c], D^1_\varnothing) \geq \epsilon, d([c], D^2_\varnothing) \geq \epsilon, ...\}$$

where $[c]$ is semantics of the constant program that returns c for each input, $\mathbb{D}$ is domain. Note that c calculated this way may not be available in the given instruction set and an appropriate instruction may need to be synthesized on demand.

Finally we combine LibrarySearch with OptimalConstant into the oracle that returns a program from L or the optimal constant depending on which one provides a closer match:

$$\text{OracleGet}(D, D^1_\varnothing, D^2_\varnothing, ...) = \operatorname*{arg\,min}_{p\in\{\text{LibrarySearch}(D, D^1_\varnothing, D^2_\varnothing, ...),\ \text{OptimalConstant}(D, D^1_\varnothing, D^2_\varnothing, ...)\}} d(s(p), D).$$

Equation (7.2) refers to semantic distance from Eq. (7.1) that in turn involves Cartesian product of components of desired semantics. This suggest that to calculate $d(D^1, D^2)$, all combinations of components of desired semantics D^1 and D^2 must be enumerated, which may be a subject for combinatorial explosion. Below we show that for L_z metrics, Eq. (7.2) can be calculated efficiently.

First, in Eq. (7.5) we expand Eq. (7.2) by substituting d with L_z metric. Since raising to the power of $\frac{1}{z}$ is monotonous, we can move it outside inner min term in Eq. (7.6) and drop in Eq. (7.7), because it does not change the result of arg min. Minimum of sums in Eq. (7.7) is equal to minimum of sums of minimums in Eq. (7.8). Because each sum in Eq. (7.8) is equal, we drop the minimum in Eq. (7.9).

$$\begin{aligned}
&\operatorname*{arg\,min}_{p\in L'} \min_{s\in\Pi_i D_i} \left(\sum_{j=1}^{|F|} |s_j(p) - s_j|^z\right)^{\frac{1}{z}} && (7.5)\\
&= \operatorname*{arg\,min}_{p\in L'} \left(\min_{s\in\Pi_i D_i} \sum_{j=1}^{|F|} |s_j(p) - s_j|^z\right)^{\frac{1}{z}} && (7.6)\\
&= \operatorname*{arg\,min}_{p\in L'} \min_{s\in\Pi_i D_i} \sum_{j=1}^{|F|} |s_j(p) - s_j|^z && (7.7)\\
&= \operatorname*{arg\,min}_{p\in L'} \min_{s\in\Pi_i D_i} \sum_{j=1}^{|F|} \min_{s_k\in D_j} |s_j(p) - s_k|^z && (7.8)\\
&= \operatorname*{arg\,min}_{p\in L'} \sum_{j=1}^{|F|} \min_{s_k\in D_j} |s_j(p) - s_k|^z && (7.9)
\end{aligned}$$

The final formula proves that the choice of the optimal s_ks from components D_i of desired semantics D can be done for each fitness case separately, hence the calculation of Cartesian product of D_is is superfluous.

An analogous result can be shown for semantic distances in definition of L' in Eq. (7.3). First, we substitute $D^1 = s(p)$, $D^2 = D_\varnothing$ and Eq. (7.1) with L_z metric for $d(\cdot,\cdot)$, which results in Eq. (7.10). Then we carry out similar transformations to the above, resulting in Eq. (7.11).

$$\min_{s\in\Pi_i D_i} \left(\sum_{j=1}^{|F|} |s_j(p) - s_j|^z\right)^{\frac{1}{z}} \geq \epsilon \qquad (7.10)$$

$$\sum_{j=1}^{|F|} \min_{s_k\in D_j} |s_j(p) - s_k|^z \geq \epsilon^z \qquad (7.11)$$

Assuming that technically D_is are sorted, Eq. (7.9) can be calculated in $O(|L| \cdot |F| \cdot \log |D_i|)$.

Similarly we show how to efficiently calculate an optimal constant from Eq. (7.4). An analogous transformation for Eq. (7.4) results in Eq. (7.12). Since D_is in Eq. (7.12) define their own marginal distance functions without maximums, and with minimums in points from D_is, i.e., the points where $|c - s_i|^z = 0$, their sum cannot have minimums in points lower than the minimum and higher than the maximum point in $\bigcup\limits_{D_i\in D} D_i$, hence we narrow the domain of c in Eq. (7.13).

$$\operatorname*{arg\,min}_{c\in\mathbb{D}'} \sum_{D_i\in D} \min_{s_i\in D_i} |c - s_i|^z \qquad (7.12)$$

$$= \arg\min_{c\in\mathbb{D}''} \sum_{D_i\in D} \min_{s_i\in D_i} |c - s_i|^z \tag{7.13}$$

$$\mathbb{D}'' = \{c : c \in \mathbb{D}',\ \min \bigcup_{D_k\in D} D_k \le c \le \max \bigcup_{D_k\in D} D_k\}$$

Equation (7.13) for L_1 ($z = 1$) boils down to Eq. (7.14). The term under summation in Eq. (7.14) is piecewise linear function with roots in D_i and no plateaus. Since the sum of piecewise linear functions is also piecewise linear function, the sum has minima, in the same points where the summed functions, i.e., $\bigcup_{D_k\in D} D_k$. Thus we narrow the set of c values in Eq. (7.15).

$$\arg\min_{c\in\mathbb{D}''} \sum_{D_i\in D} \min_{s_i\in D_i} |c - s_i| \tag{7.14}$$

$$= \arg\min_{c\in\mathbb{D}''\cap \bigcup_{D_k\in D} D_k} \sum_{D_i\in D} \min_{s_i\in D_i} |c - s_i| \tag{7.15}$$

Given D_is are sorted, Eq. (7.15) can be calculated in $O(|F|^2 \cdot |D_i| \cdot \log |D_i|)$ time.

Unfortunately for $z \ne 1$, the function is non-linear and the whole set $\mathbb{D}''$ is supposed to be searched. For Boolean domain the search can be efficiently conducted, since there are only two values to verify, i.e., $\{0, 1\}$. However for real domain a kind of numerical optimization method may be required to solve this problem optimally.

The source of programs for the library can be arbitrary. In [133] we studied two types of libraries: dynamic library built from all (sub)programs found in the current population, and static library created from programs precomputed prior to the evolutionary run. The main advantage of static library is the cost of creating library to be borne once, since the library can be reused multiple times. In contrary dynamic library has to be built every generation. On the other hand, dynamic library is able to adapt to changing state of evolution, while the static one is immutable. Our previous research confirmed an expected result that from the perspective of fitness-based performance of AGX and RDO, it does not matter how the library is created, but how many distinct semantics are represented by programs inside. We hypothesize that the same conclusion applies CX and CM, since they are effective derivations of AGX and RDO, respectively.

7.8 Discussion on applicability of the operators

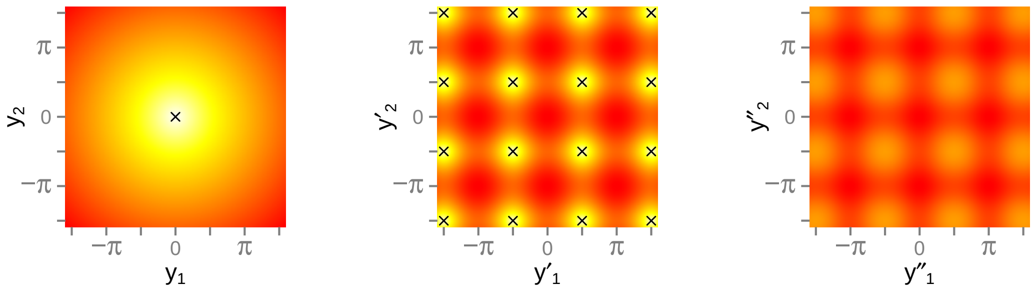
CTS requires the knowledge on the target t. This is not a common requirement, since most of selection operators found in literature do not require access to t.

CM requires access to t too. For both CTS and CM this drawback can be worked around, e.g., by substituting t supplied as argument with an artificially synthesized surrogate target.

On the other hand, CX constructs a surrogate target of recombination using semantics of parents. CX does not require any information on target t of the program induction problem. This may be considered as an advantage of CX over CM, and paves the way for using CX in problems, where CM cannot be used. For instance, in a scenario, where fitness function is evaluated by an external system or a simulator, in which the target is hidden.

7.9 Fitness landscape and program inversion

However fitness landscape seen by GSGP is a cone (see Section 5.2), the search performed by the oracle in CM and CX operators does not benefit from this fact. This is because the conic shape of the fitness landscape is transformed by semantic backpropagation in a non-linear way caused by



(a) $f((y_1,y_2)) = \sqrt{\sum_{i=1}^{2} y_i^2}$ **(b)** $f((y'_1,y'_2)) = \sqrt{\sum_{i=1}^{2} \cos^2 y'_i}$ **(c)** $f((y''_1,y''_2)) = \sqrt{\sum_{i=1}^{2} \cos^2(\sin y''_i)}$

Figure 7.3: Transformation of L_2-fitness landscape for two fitness cases and target $t = (0,0)$ when inverting program $\cos(\sin x)$. Plots present fitness of the entire program depending on semantics of its parts. Crosses mark fitness 0.

(unrestricted) characteristics of instructions on the backpropagation path. The single target of the original fitness landscape may be duplicated or vanish at all depending on approached instructions, current semantics of the program and the original target.

Figure 7.3a visualizes L_2-fitness landscape for two fitness cases and target $t = (0,0)$. Given a program $\cos(\sin x)$, semantic backpropagation inverts it instruction by instruction. Figure 7.3b shows fitness landscape after inverting the root $\cos(\cdot)$ node, i.e., fitness of $\cos(\cdot)$ program in function of semantics of its argument. Target t is replicated here infinite times due to periodicity of $\cos(\cdot)$ and new targets take form of $({}^{\pi}\!/_2 + k\pi, {}^{\pi}\!/_2 + k\pi), k \in \mathbb{Z}$. Hence the oracle when queried for a replacement for $\cos(\cdot)$ argument may return a program matching any of these new targets to result in semantics t of the entire program. Next, when backpropagation descends to the $\sin(\cdot)$ node the fitness landscape is transformed again, which we visualize in Figure 7.3c. This time, however, all targets vanish, since there is no real argument for which $\cos(\sin(\cdot))$ outputs 0. In general target vanishes if semantic backpropagation returns $\emptyset$ for any D_i in desired semantics. In this situation the oracle cannot return a program that causes semantics t of the entire program, however we argument that the fitness of the entire program still can be improved unless all D_is are $\emptyset$. This is the reason why we discard empty sets in Eq. (7.1) for semantic distance, allowing thus library search to operate.

There are two crucial conclusions of the above example. First, the number of replicated targets may grow exponentially in length of the backpropagation path. However desired semantics is designed to handle this situation in the memory-efficient way. For instance 16 targets seen in Figure 7.3b are stored using 8 values: $D = (\{{}^{-3\pi}\!/_2, {}^{-\pi}\!/_2, {}^{\pi}\!/_2, {}^{3\pi}\!/_2\}, \{{}^{-3\pi}\!/_2, {}^{-\pi}\!/_2, {}^{\pi}\!/_2, {}^{3\pi}\!/_2\})$. Second, transformation of the fitness landscape causes non-linear relation between semantic distances calculated inside program and for entire programs. Hence, even a little mismatch between desired semantics and the semantics of the program returned by the oracle may be amplified when measured for the entire program. On the other hand, a high distance measured inside program, may be alleviated for the entire program.

Chapter 8

Design of experiments

This chapter is devoted to present general insight on design of experiments. First, we formulate research questions. Then, we present benchmark problems and the parameters of evolution that are fixed across experiments. Finally, we introduce technical tools and statistical framework to carry out the experiments and analyze the outcomes. The actual experiments will be presented in subsequent chapters.

8.1 Research questions

We want to experimentally evaluate characteristics of competent algorithms presented in Chapter 7 on a broad suite of benchmark problems and in comparison to selected reference algorithms. We analyze the impact and characteristics of initialization, selection, mutation and crossover in separate experiments. Then, we verify how the particular combinations of the parts act together and determine the optimal proportions of mutation and crossover.

More specifically, we execute this experimental plan by answering five questions.

Question 1

Does CI provide more semantically diverse and geometric populations than canonical RHH (cf. Section 3.4) and effective non-geometric SDI (cf. Algorithm 6.1)? We answer this question in Section 9.1.

Question 2

Is the probability of selecting geometric and effective parents higher for CTS than for traditional TS (cf. Definition 2.8) and non-geometric STS (cf. Algorithm 6.3)? The corresponding experiment is presented in Section 9.2.

Question 3

Are the key characteristics of CM (training and test-set fitness, effectiveness, programs size, time-requirements) better than those of the canonical TM's (cf. Algorithm 3.3), effective SDM's (cf. Algorithm 6.4) and exact geometric SGM's (cf. Section 6.2.1)? Does the use of CI in place of RHH influences the characteristics of CM? We answer this question in Section 9.3.

Question 4

Are the key characteristics of CX (training and test-set fitness, effectiveness, geometry, programs size, time-requirements) better than those of the canonical TX's (cf. Algorithm 3.2), effective SDX's (cf. Algorithm 6.6) and exact geometric SGX's (cf. Section 6.2.1)? Does the use of CI and CTS influences characteristics of CX? The corresponding experiment is carried out in Section 9.4.

Table 8.1: Benchmark problems. In single-variable symbolic regression 20 Chebyshev nodes and 20 uniformly picked points in the given range are used for training- and test-sets, respectively. For two-variable these numbers amount to 10 for each variable and Cartesian product of them constitutes F. In Boolean domain training-set incorporates all inputs and there is no test-set.

Symbolic regression benchmarks

Problem	Definition (formula)	Variables	Range	$\lvert F\rvert$
R1	$(x_1+1)^3/(x_1^2-x_1+1)$	1	$\langle -1,1\rangle$	20
R2	$(x_1^5-3x_1^3+1)/(x_1^2+1)$	1	$\langle -1,1\rangle$	20
R3	$(x_1^6+x_1^5)/(x_1^4+x_1^3+x_1^2+x_1+1)$	1	$\langle -1,1\rangle$	20
Kj1	$0.3x_1\sin(2\pi x_1)$	1	$\langle -1,1\rangle$	20
Kj4	$x_1^3e^{-x_1}\cos(x_1)\sin(x_1)(\sin^2(x_1)\cos(x_1)-1)$	1	$\langle 0,10\rangle$	20
Ng9	$\sin(x_1)+\sin(x_2^2)$	2	$\langle 0,1\rangle^2$	100
Ng12	$x_1^4-x_1^3+\frac{x_2^2}{2}-x_2$	2	$\langle 0,1\rangle^2$	100
Pg-1	$1/(1+x_1^{-4})+1/(1+x_2^{-4})$	2	$\langle -5,5\rangle^2$	100
Vl1	$e^{-(x_1-1)^2}/(1.2+(x_2-2.5)^2)$	2	$\langle 0,6\rangle^2$	100

Boolean benchmarks

Problem	Instance	Variables	$\lvert F\rvert$
Even parity	Par5	5	32
	Par6	6	64
	Par7	7	128
Multiplexer	Mux6	6	64
	Mux11	11	2048
Majority	Maj7	7	64
	Maj8	8	128
Comparator	Cmp6	6	64
	Cmp8	8	256

Question 5

Does the combination of CX and CM lead to smaller training and test errors, smaller programs, or less time-requirements? What are the optimal proportions of CX and CM? The experiment is presented in Chapter 10.

8.2 Benchmark problems

We conduct our experiments using symbolic regression and Boolean function synthesis problems of quite different characteristics to determine which observations generalize across problem domain boundaries, and which not.

The aim of symbolic regression problem is to induce a function that minimizes the total approximation error. It differs from typical regression problem in that, there is no a priori given model given to be tuned, but the model is to be expressed with a set of available arithmetic instructions and elementary functions. In other words, symbolic regression is model-free regression, and its formulation exactly matches our definition of program synthesis task as formulated in Section 3.3.

We employ nine symbolic regression benchmarks taken from [106] and [134], listed in Table 8.1. There are five univariate problems, three rational functions, one trigonometric function and one mixed trigonometric-exponential function. There are four bivariate problems, one trigonometric, one polynomial, one rational and one mixed exponential-rational function. GP may produce virtually any function, including polynomial.

For polynomial interpolation, a Runge's phenomenon [151] may occur, a problem of oscillation of the model at the ends of the interpolating range. We employ a common workaround to protect from Runge's phenomenon by choosing Chebyshev nodes [28, Ch 8]:

$$x_k = \frac{1}{2}(a+b) + \frac{1}{2}(b-a)\cos\left(\frac{2k-1}{2n}\pi\right), \; k = 1..n$$

as values for independent variables in the training-set. For the univariate problems, we calculate $n = 20$ Chebyshev nodes in the range $\langle a, b\rangle$ indicated in Table 8.1, for bivariate problems, we calculate $n = 10$ Chebyshev nodes in that range for each variable, next combine them using Cartesian product. For test-set we pick uniformly 20 or 10 values from that range, respectively for one- and two-variable problems. Note that this procedure is in general not applicable to real-world problems, where the data is given 'as-is' and there is no control on the choice of arguments of regressed function.

In Boolean function synthesis problems, a set of fitness cases incorporates all combinations of inputs, i.e., complete knowledge on desired program behavior, hence there are 2^v training fitness cases, where v is the number of input variables. For the same reason, there is no test-set. The benchmarks presented in Table 8.1 come from [106] and are divided into four groups. In *even parity* problems the goal is to return 1 if the number of input variables set to 1 is even, otherwise 0. In *multiplexer*, the first n inputs determine which of the remaining 2^n inputs should be copied to program output. The total number of input variables in these problems is thus constrained by $v = n + 2^n, n \in \mathbb{Z}$. In *majority*, the aim is to return 1 if more than a half of inputs is 1, otherwise 0. In *comparator* problem v input variables are divided in two groups of $^v/_2$ variables, which encode two integers. The aim is to return 1 if the first number is greater than the second, otherwise 0.

8.3 Parameter setup

Table 8.2 presents the setting of parameters of GP used in experiments. The upper part of the table groups the generic parameters of evolutionary search, while the lower one the parameters specific for particular components (initialization, selection, mutation, crossover). Parameters not presented there are set to ECJ's defaults [101]. The presented parameters were tuned in preliminary experiments.

A justification is required for the choice of fitness functions and semantic distances. For symbolic regression L_2, metric is actually a scaled root mean square error, which is a common error measure used in regression problems. Analogously, L_1 metric is equivalent to Hamming metric for Boolean problems, which is a typical measure of error in Boolean domain. Given these fitness functions, the natural choice of semantic distance metrics comes from the rationale presented in Section 5.5. We use L_2 semantic distance for symbolic regression, since under this metric geometric crossover has the weak property of progress (cf. Definition 5.10), and the strong property of progress (cf. Definition5.12) for parents of the same fitness. For Boolean domain we use L_1 semantic distance that does not provide the above properties, because of the discrete nature of Boolean space. That is, a segment between two distinct points in Boolean space under L_z metrics other than L_1 contains only these two points. Therefore for Boolean domain the crossover can be simultaneously geometric and effective only for L_1 semantic distance.

$\mathsf{Invert}(a, k, o)$ function from Table 7.1 if there are more than one inversions of o available, returns at most two representative values, e.g., one positive and one negative for trigonometric functions. We did this to prevent semantic backpropagation from combinatorial explosion.

The optimal constant in Eq. (7.4) is calculated using formula

$$\operatorname*{arg\,min}_{c \in (\bigcup_{D_k \in D} D_k)} \sum_{D_i \in D} \min_{s_i \in D_i} |c - s_i|^z \tag{8.1}$$

Table 8.2: Parameters of evolution.

Generic parameters		Symbolic regression	Boolean domain
Number of runs		30	
Fitness function		L_2 metric	L_1 metric
Semantic distance		L_2 metric	L_1 metric
Similarity threshold on semantic distance		$2^{-52} \approx 2.22 \times 10^{-16}$	1
Population size		1000	
Termination condition		At most 100 generations or find of a program with fitness 0	
Instructions Φ		x_1, x_2, $+$, $-$, $\times$, $/$, sin, cos, exp, log, ERC[a]	$x_1, x_2, ..., x_{11}$,[b] and, or, nand, nor, ERC
Algorithm-specific parameters			
RHH, SDI, CI	Max initial tree height	6	
RHH	*Syntactic* duplicate retries	10	
SDI	*Semantic* duplicate retries	10	
CI	Convex hull expansion retries	10	
TS, STS, CTS	Tournament size μ	7	
CM, CX	Library	Dynamic library composed of all subprograms in the current population	
SGM, SGX	Random tree p_x source	ERC in range $\langle 0, 1\rangle$	Grow with tree height in range $\langle 1, 3\rangle$
SDM, SDX	Neutral offspring retries	10	

[a] log is defined as $\log |x|$; / returns 0 if divisor is 0.
[b] The number of inputs depends on a particular problem instance

that restricts the range of constants to be verified in Eq. (7.13) to values that exist in components D_ks of desired semantics D. The formula can be efficiently calculated in $O(|F|^2|D_i| \cdot \log |D_i|)$ time, however may result in a suboptimal constant in symbolic regression (in Boolean domain the formula is optimal).

The similarity thresholds in Table 8.2 are the minimal distances between semantics to consider them distinct, respectively to the domains, e.g., in real domain the threshold equals to difference between two subsequent floating point numbers, where one of them is 1, in Boolean domain the threshold comes from distance between two semantics that differ in just one bit. We use these thresholds in comparisons of semantics involved in the considered algorithms and to provide statistics of effectiveness and neutrality.

The set of instructions Φ used by GP is domain-dependent. To satisfy its closure property (cf. Definition 3.2) for symbolic regression we use protected versions of logarithm and division. Protected logarithm returns the logarithm of the absolute value of its argument, i.e., $\log |x|$, while division returns 0 for divisor equal to zero. Our choice of this particular way of protecting division comes from [168], where this form of protection led to the smallest training-set errors of produced programs.

Instruction sets include Ephemeral Random Constants (ERCs). For symbolic regression, ERCs are random constants in range $\langle -1, 1\rangle$ or for operators that are supported by a library, the constants returned by Eq. (8.1). In Boolean domain, to maintain consistency with the symbolic regression one, we use set of constants $\{0, 1\}$.

Instruction sets are sufficient to express an optimal solution for each problem (cf. Definition 3.4).

$$\underbrace{\textsc{Ci}}_{\text{initialization}}\underbrace{\textsc{Cts}}_{\text{selection}}\underbrace{\textsc{Cx}75}_{\text{crossover}}\underbrace{\textsc{Cm}}_{\text{mutation}}$$

Figure 8.1: An exemplary configuration encoded using the naming convention for experimental setups. The configuration employs competent initialization, competent tournament selection implicitly accompanied with tournament selection, competent crossover applied with probability 75% and competent mutation applied with probability $100\% - 75\% = 25\%$.

We employ the naming convention presented in Figure 8.1 to succinctly identify experimental configurations. The name of each configuration consists of four components:

- The name of initialization operator,
- The name of selection operator,
- The name of crossover operator, optionally followed by its probability $\Pr(X)$,
- The name of mutation operator, applied with probability $\Pr(M) = 1 - \Pr(X)$ (cf. breeding pipeline, Section 3.8).

Probability is omitted if there is only one variation operator. Mate selection algorithms are accompanied with tournament selection, which is implicit in the name.

8.4 Experimental tools

We implemented all compared algorithms in a unified environment of Evolutionary Computation in Java (ECJ) software suite [101]. We use Java 1.8 and Linux running in x64 mode on Intel Core i7-950 processor and 6GB DDR3 RAM. The CPU times obtained in this setup exclude calculation of statistics.

The results were subject to thorough statistical analysis. For each statistical aggregate, e.g., average, median, fraction or correlation, we provide corresponding 95% confidence intervals.

Each time we present results in a plot of an aggregate over generations, to calculate the aggregate in generations where some of the runs terminated (e.g., found the optimum earlier), we use a value from the generation where the optimum has been found.

Each time we present results in tabular form, we rank the algorithms on each benchmark problem independently and present also the average ranks. Significance of differences between algorithms is determined by means of Friedman's test for multiple achievements of multiple subjects (Friedman's test for short) [78], a non-parametric test suitable for a wide range of statistical aggregates. In case a null hypothesis of a Friedman's test is rejected, i.e., the outcome (a.k.a. *achievements* in this context) of at least one algorithm is statistically different than at least one other algorithm, we conduct a post-hoc analysis using symmetry test[68] to determine the pairs of algorithms of different achievements.

To measure association between numeric variables we use Pearson's correlation coefficient [135], and for ordinal variables Spearman's rank correlation [161]. In both cases, we use t-test to determine the significance of correlation [78]. For nominal variables we use χ^2 test for independence [78] together with Cramér's V to express the strength of association in range $\langle 0, 1\rangle$ independently on sample size [39].

Chapter 9

Evaluation of Competent Algorithms for operators in isolation

In this chapter we analyze competent algorithms for the operators proposed in Chapter 7 and compare them with the existing canonical and semantic analogs. We consider initializations, selections, mutations and crossovers separately to reveal their features, unaffected to the possible extent by interactions with the other components of GP workflow. The chapter consists of four separate experiments, presenting analysis of performance, distribution of semantics, effectiveness, degree of geometry, size of programs and computational cost. Overall conclusions are provided at the end of this chapter.

9.1 Initialization

This section is intended to experimentally assess the CI operator in the context of other population initialization operators, RHH and SDI, and in effect answer question 1 posed in Section 8.1.

9.1.1 Fitness distribution

Empirical distributions of program fitness in populations initialized by RHH, SDI and CI, respectively, for each benchmark problem, are presented in Figure 9.1. The width of each bin is 1 and bin values are averaged over 30 runs of each operator initializing population of size 1000.

We observe that shape of distribution depends on domain. In symbolic regression, we observe Poisson-like distributions with $\lambda < 20$ for all considered problems. The distributions in Boolean domain remind normal distributions with standard deviation and kurtosis varying across problems. To statistically verify our observations, we conduct D'Agostino's K-squared tests for departure from normality [42] for each combination of operator and problem and present p-values in Table 9.1. Indeed, the tests confirm that none of the operators produces normally distributed fitness of programs in symbolic regression domain, however in Boolean domain RHH produces normal distribution for Par* (where Par* stands for all Par problems) and Cmp8 problems, SDI for Par7 and Mux11 and CI for Mux*. Deep insight into histograms reveals the reason for divergence between test results and observed distributions for other combinations of Boolean problems and operators: in Mux* and Maj* problems odd values of fitness are much less likely than even values, and in Cmp* problems even values are much more likely then the odd ones. Thus strictly speaking the distributions are not normal, however they look like normal distributions with 'holes'.

All considered operators produce programs in similar fitness ranges but of substantially different shapes. RHH tends to produce multi-modal, rugged distributions with few peaks. SDI produces

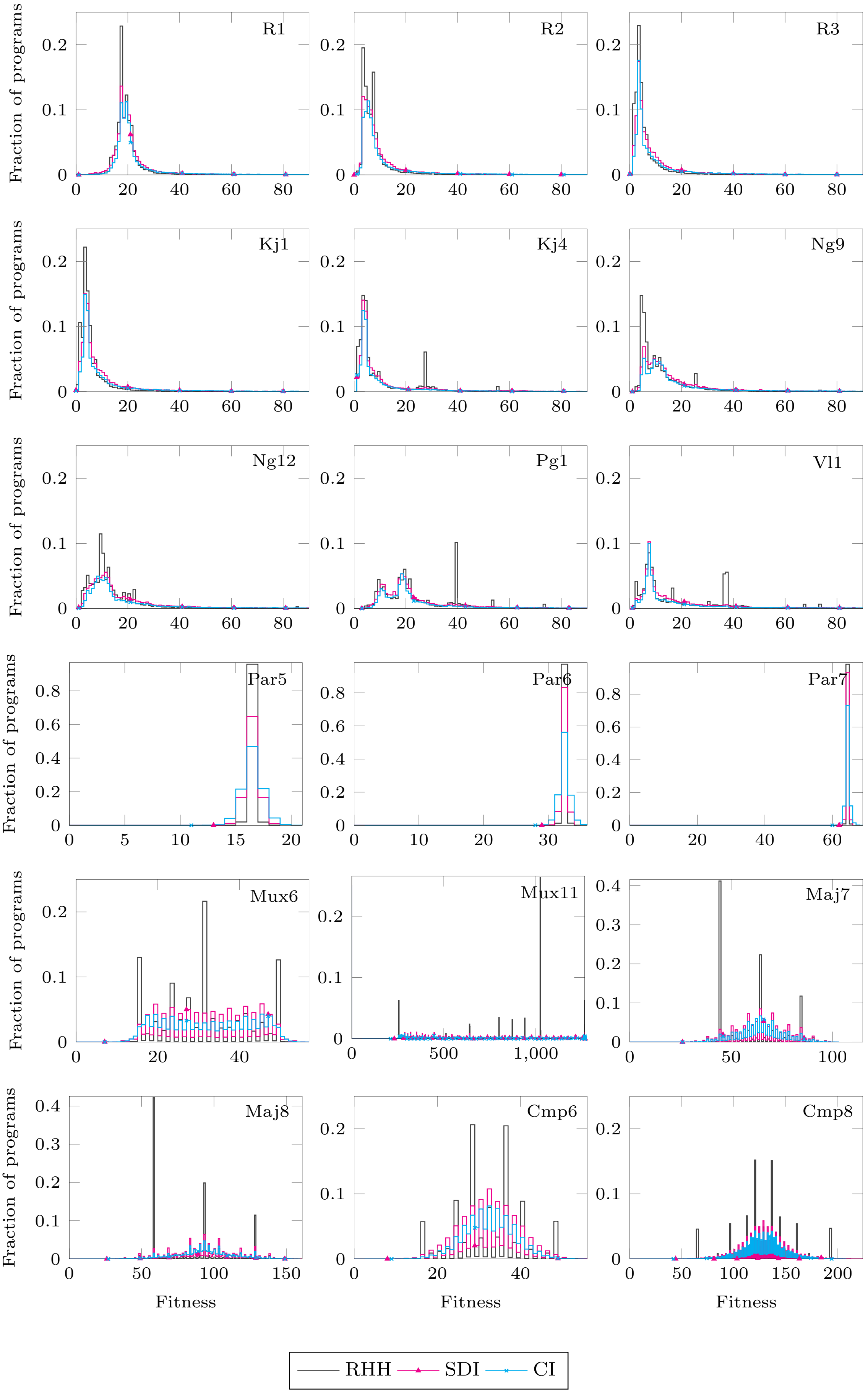


Figure 9.1: Empirical distribution of fitness in initial population initialized by RHH, SDI and CI, respectively.

Table 9.1: D'Agostino's K-squared tests for departure from normality [42] of distribution of fitness in populations initialized by RHH, SDI and CI, respectively, divided by problems. P-values ≥ 0.05 (in bold) reflect normal distribution.

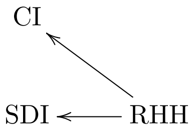
Problem	RHH	SDI	CI
R1	0.000	0.000	0.000
R2	0.000	0.000	0.000
R3	0.000	0.000	0.000
Kj1	0.000	0.000	0.000
Kj4	0.000	0.000	0.000
Ng9	0.000	0.000	0.000
Ng12	0.000	0.000	0.000
Pg1	0.000	0.000	0.000
Vl1	0.000	0.000	0.000
Par5	**0.606**	0.000	0.000
Par6	**0.606**	0.000	0.000
Par7	**0.606**	**0.606**	0.000
Mux6	0.000	0.000	**0.676**
Mux11	0.000	**0.996**	**0.483**
Maj7	0.000	0.000	0.000
Maj8	0.000	0.000	0.000
Cmp6	0.000	0.000	0.000
Cmp8	**0.063**	0.000	0.000

Table 9.2: Median and 95% confidence intervals of fitness of programs in initial population created by RHH, SDI and CI, respectively (the best in bold).

Problem		RHH			SDI			CI	
R1	18.23 ≤	**18.30**	≤ 18.36	19.26 ≤	19.32	≤ 19.37	20.26 ≤	20.34	≤ 20.42
R2	6.08 ≤	**6.12**	≤ 6.14	6.86 ≤	6.92	≤ 6.99	8.33 ≤	8.49	≤ 8.67
R3	3.65 ≤	**3.69**	≤ 3.72	5.54 ≤	5.65	≤ 5.74	7.65 ≤	7.85	≤ 8.05
Kj1	3.83 ≤	**3.87**	≤ 3.91	5.63 ≤	5.74	≤ 5.83	7.83 ≤	8.02	≤ 8.22
Kj4	5.64 ≤	**5.85**	≤ 6.07	8.91 ≤	9.17	≤ 9.41	17.12 ≤	18.44	≤ 19.98
Ng9	9.64 ≤	**9.78**	≤ 9.93	14.64 ≤	14.87	≤ 15.11	21.26 ≤	22.00	≤ 22.74
Ng12	11.06 ≤	**11.18**	≤ 11.30	14.85 ≤	15.21	≤ 15.55	22.45 ≤	23.12	≤ 23.86
Pg1	23.22 ≤	**23.54**	≤ 23.93	23.50 ≤	23.91	≤ 24.31	41.73 ≤	43.75	≤ 46.50
Vl1	17.55 ≤	**18.06**	≤ 18.82	19.77 ≤	20.38	≤ 20.89	51.71 ≤	56.14	≤ 60.90
Par5	16.00 ≤	**16.00**	≤ 16.00	16.00 ≤	**16.00**	≤ 16.00	16.00 ≤	**16.00**	≤ 16.00
Par6	32.00 ≤	**32.00**	≤ 32.00	32.00 ≤	**32.00**	≤ 32.00	32.00 ≤	**32.00**	≤ 32.00
Par7	64.00 ≤	**64.00**	≤ 64.00	64.00 ≤	**64.00**	≤ 64.00	64.00 ≤	**64.00**	≤ 64.00
Mux6	31.00 ≤	**31.00**	≤ 31.00	31.00 ≤	32.00	≤ 33.00	31.00 ≤	32.00	≤ 32.00
Mux11	1023.00 ≤	1023.00	≤ 1023.00	1023.00 ≤	1025.00	≤ 1039.00	1005.00 ≤	**1015.00**	≤ 1025.00
Maj7	58.00 ≤	**58.00**	≤ 58.00	64.00 ≤	64.00	≤ 64.00	64.00 ≤	64.00	≤ 64.00
Maj8	80.00 ≤	**82.00**	≤ 83.00	93.00 ≤	93.00	≤ 93.00	93.00 ≤	93.00	≤ 93.00
Cmp6	32.00 ≤	**32.00**	≤ 32.00	32.00 ≤	**32.00**	≤ 32.00	32.00 ≤	**32.00**	≤ 32.00
Cmp8	128.00 ≤	**128.00**	≤ 128.00	128.00 ≤	**128.00**	≤ 128.00	128.00 ≤	**128.00**	≤ 128.00
Rank:		1.33			2.14			2.53	

Table 9.3: Post-hoc analysis of Friedman's test conducted on Table 9.2: p-values of incorrectly judging an operator in a row as outranking an operator in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RHH	SDI	CI
RHH		**0.010**	**0.000**
SDI			0.333
CI			



smoother distributions, mostly unimodal. CI provides distributions similar in shape to those of SDI, however even smoother and with noticeably lower kurtosis.

The rugged characteristics of RHH's fitness distribution may indicate bias of the operator toward programs within a small set of semantics, i.e., oversampling of some areas of the search space and undersampling others. In turn effectiveness of SDI and CI causes the search space to be sampled more evenly.

Median and 95% confidence interval of fitness in initial populations provided by RHH, SDI and CI, respectively, are given in Table 9.2.

RHH provides the lowest (the best) median fitness of all compared operators, while CI the highest (the worst), and SDI fares in between. We explain this phenomenon by the fact that CI strives to build certain geometric structures that may require some programs to be far from the target, while other operators only discard syntactic (RHH) or semantic (SDI) duplicates, completely ignoring the distribution of the programs in semantic space. Note that fitness results are of a lower importance, since an initialization operator is supposed to prepare a beneficial starting conditions for the variation operators, instead of solving the problem in the first place.

We conduct Friedman's test on median fitness to assess differences between the operators. The test results in p-value of 4.74×10^{-5}, thus at significance level $\alpha = 0.05$ there is at least one pair of significantly different operators. Post-hoc analysis presented in Table 9.3 indicates that RHH produces significantly fitter programs than both SDI and CI, while the latter two are indiscernible.

9.1.2 Effectiveness and geometry

To verify effective properties of SDI and CI we count the average number of semantically unique programs in population of size 1000 initialized by each of the considered operators and present the average together with 95% confidence interval in Table 9.4.

All applications of SDI and CI are effective, hence both operators can be considered the most reliable and accurate on this criterion. RHH achieves $46\% - 98\%$ effective applications, depending on a problem. The differences in values observed for RHH come from varying dimensionality and size of semantic spaces across problems and/or different inputs used to calculate semantics. For symbolic regression the relationship between dimensionality of semantic space (20 for R*, Kj* problems and 100 for the rest) and number of effective applications seems to be weak and non-monotonous. In the Boolean domain, the number of effective applications clearly rises with the dimensionality of semantic space. We explain this phenomenon by the fact that semantic space for symbolic regression is continuous and infinite, while for Boolean domain it is finite and discrete, making it more likely to create semantically equal programs.

Friedman's test conducted on the number of unique programs results in conclusive p-value of 6.40×10^{-7}, thus we present post-hoc analysis in Table 9.5. The analysis confirms that both SDI and CI provide significantly higher ratio of effective applications than RHH.

Table 9.6 shows average and 95% confidence interval of the numbers of applications of RHH, SDI and CI until the first L_1-geometric application (cf. Definition 5.4) in the Boolean domain. Note that after the first geometric application, all subsequent applications are geometric by definition (because population's convex hull already consists the target). In other words we can say that an

Table 9.4: Average and 95% confidence interval of the numbers of semantically unique individuals in initial population of size 1000 created by RHH, SDI and CI, respectively (the highest in bold).

Problem	RHH	SDI	CI
R1	978.97 ±1.43	**1000.00** ±0.00	**1000.00** ±0.00
R2	978.97 ±1.43	**1000.00** ±0.00	**1000.00** ±0.00
R3	978.97 ±1.43	**1000.00** ±0.00	**1000.00** ±0.00
Kj1	978.97 ±1.43	**1000.00** ±0.00	**1000.00** ±0.00
Kj4	979.70 ±1.36	**1000.00** ±0.00	**1000.00** ±0.00
Ng9	977.50 ±1.52	**1000.00** ±0.00	**1000.00** ±0.00
Ng12	977.50 ±1.52	**1000.00** ±0.00	**1000.00** ±0.00
Pg1	978.20 ±1.38	**1000.00** ±0.00	**1000.00** ±0.00
Vl1	979.23 ±1.43	**1000.00** ±0.00	**1000.00** ±0.00
Par5	456.93 ±5.19	**1000.00** ±0.00	**1000.00** ±0.00
Par6	558.63 ±5.05	**1000.00** ±0.00	**1000.00** ±0.00
Par7	622.47 ±4.33	**1000.00** ±0.00	**1000.00** ±0.00
Mux6	558.63 ±5.05	**1000.00** ±0.00	**1000.00** ±0.00
Mux11	779.57 ±4.55	**1000.00** ±0.00	**1000.00** ±0.00
Maj7	622.47 ±4.33	**1000.00** ±0.00	**1000.00** ±0.00
Maj8	675.93 ±3.78	**1000.00** ±0.00	**1000.00** ±0.00
Cmp6	558.63 ±5.05	**1000.00** ±0.00	**1000.00** ±0.00
Cmp8	675.93 ±3.78	**1000.00** ±0.00	**1000.00** ±0.00
Rank:	3.00	1.50	1.50

Table 9.5: Post-hoc analysis of Friedman's test for Table 9.4: p-values of incorrectly judging an operator in row as producing higher number of effective programs than an operator in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RHH	SDI	CI
RHH			
SDI	**0.000**		
CI	**0.000**	1.000	

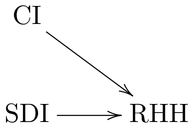


Table 9.6: Average and 95% confidence interval of the numbers of RHH, SDI and CI applications, respectively, until the first L_1-geometric application (the highest in bold). If a run does not result in any geometric application, we count it as 1000 (max number of applications).

Problem	RHH	SDI	CI
Par5	4.93 ±0.56	5.37 ±0.85	**4.47** ±0.44
Par6	5.67 ±0.77	5.63 ±0.75	**4.87** ±0.52
Par7	6.33 ±0.80	5.30 ±0.56	**5.07** ±0.57
Mux6	5.00 ±0.52	5.50 ±0.73	**4.87** ±0.53
Mux11	7.33 ±0.95	7.43 ±0.96	**6.40** ±0.63
Maj7	5.53 ±0.75	5.27 ±0.55	**5.03** ±0.57
Maj8	6.90 ±0.77	5.70 ±0.68	**5.33** ±0.54
Cmp6	4.87 ±0.53	4.97 ±0.70	**4.47** ±0.51
Cmp8	7.20 ±0.81	5.73 ±0.69	**5.33** ±0.53
Rank:	2.56	2.44	1.00

Table 9.7: Post-hoc analysis of Friedman's test conducted on Table 9.6: p-values of incorrectly judging an operator in a row as on average requiring less applications until the first L_1-geometric one than an operator in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RHH	SDI	CI
RHH			
SDI	0.970		
CI	**0.003**	**0.006**	

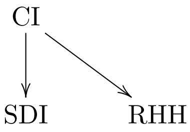


Table 9.8: Average and 95% confidence interval of numbers of RHH, SDI and CI applications, respectively, until the first L_2-geometric application (the highest in bold). If a run does not result in any geometric application, we count it as 1000 (max number of applications).

Problem	RHH		SDI		CI	
R1	432.00	±114.75	437.07	±108.40	**300.40**	±85.36
R2	220.55	±61.81	251.11	±71.86	**183.50**	±48.66
R3	180.69	±29.40	162.32	±26.97	**97.35**	±11.24
Kj1	217.00	±43.06	210.65	±41.89	**177.41**	±62.20
Kj4	**136.83**	±44.44	176.78	±54.41	144.90	±38.89
Ng9	**205.05**	±36.96	266.07	±58.44	225.41	±49.89
Ng12	**200.60**	±38.42	270.19	±59.85	224.29	±49.35
Pg1	198.12	±65.53	278.65	±63.33	**197.78**	±28.54
Vl1	164.09	±56.46	200.61	±49.66	**144.25**	±20.08
Rank:	1.89		2.78		1.33	

operator is on average n-geometric in meaning of Definition 5.4, where n is a number from Table 9.6. The averages clearly increase with dimensionality of semantic space for all combinations of problems and operators.

The smallest ns are obtained by CI for all Boolean problems, and there is no clear leader in the group of RHH and SDI. Moreover the confidence intervals for CI are the narrowest, thus the results of CI can be considered very stable and reliable. Friedman's test results in p-value 2.77×10^{-3} and post-hoc analysis from Table 9.7 leaves no doubt that CI requires fewer number of applications than SDI and RHH to reach the first geometric application.

Table 9.8 presents analogous results for symbolic regression and L_2 metric. In 6 out of 9 problems CI achieves the lowest average. In remaining three problems RHH achieves the lowest average, however confidence intervals of RHH and CI overlap, which suggests that the differences are not significant. In 7 out of 9 problems SDI requires the highest number of applications. In 6 out of 9 problems the confidence intervals for CI are the narrowest, thus CI's results are the most stable. Friedman's test is conclusive with p-value of 6.23×10^{-3}, and post-hoc analysis in Table 9.9 indicates that CI requires significantly fewer applications than SDI, however the evidence is too weak to differentiate CI or SDI with RHH.

Juxtaposition of Tables 9.6 and 9.8 leads to conclusion that building an L_1-convex hull in Boolean domain requires less applications than L_2-convex hull in symbolic regression domain. This

Table 9.9: Post-hoc analysis of Friedman's test conducted on Table 9.8: p-values of incorrectly judging an operator in a row as on average requiring less applications until the first L_2-geometric one than an operator in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RHH	SDI	CI
RHH		0.143	
SDI			
CI	0.466	**0.006**	

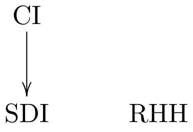
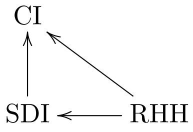


Table 9.10: Average and 95% confidence interval of numbers of nodes in programs produced by RHH, SDI and CI, respectively (the lowest in bold).

Problem	RHH	SDI	CI
R1	**8.89** ±0.09	13.36 ±0.07	81.66 ±0.86
R2	**8.89** ±0.09	13.36 ±0.07	80.96 ±0.84
R3	**8.89** ±0.09	13.36 ±0.07	81.78 ±0.86
Kj1	**8.89** ±0.09	13.36 ±0.07	82.00 ±0.86
Kj4	**8.89** ±0.09	13.23 ±0.07	78.73 ±0.82
Ng9	**8.76** ±0.08	12.68 ±0.07	64.76 ±0.66
Ng12	**8.76** ±0.08	12.68 ±0.07	64.67 ±0.66
Pg1	**8.76** ±0.08	12.52 ±0.06	59.12 ±0.62
Vl1	**8.76** ±0.08	12.52 ±0.06	62.03 ±0.65
Par5	**22.33** ±0.24	24.11 ±0.10	244.64 ±2.38
Par6	**22.32** ±0.24	23.37 ±0.10	226.81 ±2.23
Par7	**22.00** ±0.24	22.73 ±0.10	187.34 ±1.92
Mux6	**22.32** ±0.24	23.37 ±0.10	226.81 ±2.23
Mux11	21.56 ±0.24	**21.32** ±0.10	139.45 ±1.45
Maj7	**22.00** ±0.24	22.73 ±0.10	187.34 ±1.92
Maj8	**21.82** ±0.24	22.65 ±0.10	178.66 ±1.77
Cmp6	**22.32** ±0.24	23.37 ±0.10	226.81 ±2.23
Cmp8	**21.82** ±0.24	22.65 ±0.10	178.66 ±1.77
Rank:	1.06	1.94	3.00

Table 9.11: Post-hoc analysis of Friedman's test conducted on Table 9.10: p-values of incorrectly judging an operator in a row as producing smaller programs than an operator in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RHH	SDI	CI
RHH		**0.021**	**0.000**
SDI			**0.004**
CI			



may be explained by that Boolean semantic space is finite and consists of less points than the infinite regression one. This inclines us to hypothesize that in all operators L_1-convex hull of the population reaches entire Boolean semantic space with similar ease like it incorporates the target.

In an additional analysis we confirmed that all 30 runs of all operators in all Boolean problems had at least one L_1-geometric application. In symbolic regression, for each problem and operator there where runs that resulted in no geometric applications.

9.1.3 Program size

Table 9.10 presents average and 95% confidence interval of number of nodes in programs produced by RHH, SDI and CI, respectively. In all but one problem RHH produces the smallest programs. Only for Mux11, SDI produces smaller programs than RHH, however the difference is very low and the confidence intervals overlap. CI produces noticeably larger programs than both other operators.

Friedman's test is conclusive ($p = 1.16 \times 10^{-8}$) and post-hoc analysis in Table 9.11 confirms that RHH produces significantly smaller programs than SDI and CI, and SDI produces smaller programs than CI.

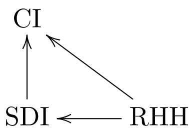
Varying numbers of nodes across problems clearly reflect the characteristics of particular operators. RHH produces exactly the same distribution of program sizes for all problems with the same set of instructions. In contrast, the distributions of program sizes produced by SDI depends

Table 9.12: Average and 95% confidence interval of time (seconds) required to initialize population of size 1000 by RHH, SDI and CI, respectively (the lowest in bold).

Problem	RHH	SDI	CI
R1	**0.05** ±0.00	0.32 ±0.12	14.04 ±6.18
R2	**0.05** ±0.00	0.49 ±0.15	14.19 ±6.14
R3	**0.05** ±0.00	0.35 ±0.13	13.53 ±6.14
Kj1	**0.04** ±0.00	0.29 ±0.10	13.79 ±6.20
Kj4	**0.04** ±0.00	0.41 ±0.16	12.21 ±4.94
Ng9	**0.05** ±0.00	0.63 ±0.40	304.97 ±94.24
Ng12	**0.05** ±0.00	1.08 ±0.60	309.63 ±94.93
Pg1	**0.04** ±0.00	0.84 ±0.56	73.43 ±31.93
Vl1	**0.05** ±0.00	1.04 ±0.63	92.61 ±39.80
Par5	**0.06** ±0.01	0.28 ±0.01	4.21 ±0.20
Par6	**0.05** ±0.00	0.26 ±0.01	4.21 ±0.22
Par7	**0.05** ±0.00	0.25 ±0.01	4.31 ±0.20
Mux6	**0.05** ±0.00	0.24 ±0.01	4.23 ±0.22
Mux11	**0.04** ±0.00	0.55 ±0.01	20.32 ±1.11
Maj7	**0.05** ±0.00	0.27 ±0.01	4.39 ±0.22
Maj8	**0.04** ±0.00	0.27 ±0.01	5.60 ±0.24
Cmp6	**0.05** ±0.00	0.24 ±0.01	4.32 ±0.23
Cmp8	**0.04** ±0.00	0.26 ±0.01	5.61 ±0.29
Rank:	1.00	2.00	3.00

Table 9.13: Post-hoc analysis of Friedman's test conducted on Table 9.12: p-values of incorrectly judging an operator in a row as faster than an operator in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RHH	SDI	CI
RHH		**0.008**	**0.000**
SDI			**0.008**
CI			



on the set of instructions and fitness cases, e.g., for R* and Kj1 problems that share the same range of inputs, the distribution is the same. Finally, CI is the only operator that depends also on the target, and in consequence its distribution of program sizes varies across problems.

9.1.4 Computational costs

Table 9.12 shows average and 95% confidence interval of time required to initialize population of size 1000 by RHH, SDI and CI, respectively. Time requirements for RHH are the lowest, SDI's ones are an order of magnitude higher, and CI's ones two orders of magnitude higher.

These observations are confirmed by Friedman's test that results in p-value of 1.20×10^{-8}, and its post-hoc analysis presented in Table 9.13.

Notice two outlying observations for CI on Ng9 and Ng12 problems. We explain these high times by the lowest range (and variance) of input variables among all considered problems (cf. Table 8.1). Low variance of inputs results in low variance of program output and makes it harder to create semantically diversified programs. This may in turn hinder the ability to expand convex hull of population, in which case CI repeats initialization routine multiple times until the convex hull expands (see line 4 of Algorithm 7.1).

To verify whether obtained times of *a particular operator* correlate with the numbers of nodes in produced programs (cf. Table 9.10) we calculate Spearman's rank correlation coefficient of these

Table 9.14: Spearman's rank correlation coefficient of number of nodes in produced programs and time consumed by an operator. Left: correlation of size and time over problems for each operator separately. Right: rankings of operators from last rows of Tables 9.10 and 9.12, and correlation of them. P-values of t test of significance of correlation, significant correlations ($\alpha = 0.05$) are marked in bold.

	Correlation	P-value	Size rank	Time rank
RHH	0.281	0.129	1.06	1.00
SDI	$-$**0.907**	0.000	1.94	2.00
CI	$-$**0.918**	0.000	3.00	3.00
		Correlation:	**0.999**	
		P-value:	0.000	

variables over all problems, for each operator separately and present them together with p-values of t test for significance of correlation in Table 9.14. We observe strong negative correlations for SDI and CI and t test is conclusive at significance level $\alpha = 0.05$ about the significance of these correlations. That is, SDI and CI produce big programs in less time than the small ones. We explain this by that both SDI and CI evaluate multiple candidates until they accept and produce semantically unique one. If by chance they attempt to produce a big program, it is less likely to be discarded, because the variety of programs' semantics increases with the program size.

Right part of Table 9.14 consists of *the rankings of all operators* on program size and consumed time took from Tables 9.10 and 9.12, and Spearman's rank correlation coefficient of these rankings. The coefficient is almost 1, hence we conclude that rankings are consistent. This may indicate that even if SDI and CI use bigger programs than RHH to achieve effectiveness and lower computational cost, they are unable to drop the cost below the RHH's one.

9.2 Selection

This section is devoted to assess properties of mate selection operators and answer question 2 from Section 8.1. We employ TS to select the first parent and one of two mate selection operators, STS and CTS, to select the second one. The control setup uses TS to select both parents; this is to verify whether STS and/or CTS favor effective and/or geometric selections

To minimize the impact of other factors and possibly approximate realistic working conditions, we calculate the interesting characteristics on a set of randomly generated populations, either by purely syntactic RHH or geometric effective CI.

9.2.1 Effectiveness and geometry

Table 9.15 presents empirical probability and 95% confidence interval of effective applications (cf. Definition 4.6) of TS, STS and CTS, respectively. Only CTS provides 100% of effective applications for all considered problems and in this is sense is the most reliable and accurate operator. STS is the runner-up and provides 92.9% $-$ 100% effective applications, depending on problem and distribution of semantics. The last operator is TS with 69.2% $-$ 99.7% effective applications.

Effective characteristic of CTS seems to be independent on the distribution of programs in the population: the 100% probability occurs for both RHH- and CI-initialized populations. The probabilities for the remaining operators are higher for CI's population than for the RHH's one. Our explanation is that all programs initialized by CI are semantically distinct, while RHH may produce semantic duplicates that obviously increases the probability of selecting a duplicate.

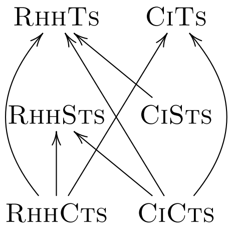
To statistically confirm our observations, we conduct Friedman's test on probability of effective selections. The test results in p-value of 4.29×10^{-12}, hence at significance level $\alpha = 0.05$ at least

Table 9.15: Empirical probability and 95% confidence interval of effective selections conducted by TS, STS and CTS (the highest in bold).

Problem	RhhTs	RhhSts	RhhCts	CiTs	CiSts	CiCts
R1	0.996 ±0.001	0.999 ±0.000	**1.000** ±0.000	0.996 ±0.001	0.999 ±0.000	**1.000** ±0.000
R2	0.995 ±0.001	0.999 ±0.000	**1.000** ±0.000	0.996 ±0.001	0.999 ±0.000	**1.000** ±0.000
R3	0.992 ±0.001	0.999 ±0.000	**1.000** ±0.000	0.996 ±0.001	1.000 ±0.000	**1.000** ±0.000
Kj1	0.992 ±0.001	0.999 ±0.000	**1.000** ±0.000	0.996 ±0.001	1.000 ±0.000	**1.000** ±0.000
Kj4	0.991 ±0.001	0.999 ±0.000	**1.000** ±0.000	0.997 ±0.001	1.000 ±0.000	**1.000** ±0.000
Ng9	0.995 ±0.001	0.999 ±0.000	**1.000** ±0.000	0.996 ±0.001	0.999 ±0.000	**1.000** ±0.000
Ng12	0.996 ±0.001	1.000 ±0.000	**1.000** ±0.000	0.996 ±0.001	1.000 ±0.000	**1.000** ±0.000
Pg1	0.996 ±0.001	0.999 ±0.000	**1.000** ±0.000	0.995 ±0.001	0.999 ±0.000	**1.000** ±0.000
Vl1	0.993 ±0.001	0.999 ±0.000	**1.000** ±0.000	0.996 ±0.001	1.000 ±0.000	**1.000** ±0.000
Par5	0.965 ±0.002	0.970 ±0.002	**1.000** ±0.000	0.997 ±0.001	0.999 ±0.000	**1.000** ±0.000
Par6	0.971 ±0.002	0.972 ±0.002	**1.000** ±0.000	0.996 ±0.001	1.000 ±0.000	**1.000** ±0.000
Par7	0.974 ±0.002	0.976 ±0.002	**1.000** ±0.000	0.996 ±0.001	0.999 ±0.000	**1.000** ±0.000
Mux6	0.692 ±0.005	0.929 ±0.003	**1.000** ±0.000	0.995 ±0.001	0.999 ±0.000	**1.000** ±0.000
Mux11	0.819 ±0.004	0.966 ±0.002	**1.000** ±0.000	0.996 ±0.001	1.000 ±0.000	**1.000** ±0.000
Maj7	0.991 ±0.001	0.998 ±0.000	**1.000** ±0.000	0.997 ±0.001	1.000 ±0.000	**1.000** ±0.000
Maj8	0.992 ±0.001	0.999 ±0.000	**1.000** ±0.000	0.996 ±0.001	1.000 ±0.000	**1.000** ±0.000
Cmp6	0.970 ±0.002	0.993 ±0.001	**1.000** ±0.000	0.996 ±0.001	1.000 ±0.000	**1.000** ±0.000
Cmp8	0.986 ±0.001	0.997 ±0.001	**1.000** ±0.000	0.995 ±0.001	0.999 ±0.000	**1.000** ±0.000
Rank:	5.889	4.278	1.500	4.778	3.056	1.500

Table 9.16: Post-hoc analysis of Friedman's test conducted on Table 9.15: p-values of incorrectly judging a setup in a row as selecting higher number of effective mate parents than a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RhhTs	RhhSts	RhhCts	CiTs	CiSts	CiCts
RhhTs						
RhhSts	0.092			0.965		
RhhCts	**0.000**	**0.000**		**0.000**	0.115	
CiTs	0.461					
CiSts	**0.000**	0.349		0.057		
CiCts	**0.000**	**0.000**	1.000	**0.000**	0.115	



one pair of setups differs. Post-hoc analysis is presented in Table 9.16. Overall, CTS is the most effective operator and there is no statistical difference between the outcomes for particular methods of population initialization.

Table 9.17 shows empirical probability and 95% confidence interval of L_1-geometric applications (cf. Definition 5.5) of TS, STS and CTS in Boolean domain problems. The values depend on population initialization: all operators achieve higher probabilities for the RHH-initialized populations than for the CI's ones. The operators suffer from *curse of dimensionality*: probability of geometric selection decreases with increasing dimensionality of semantic space for all problems. Our explanation is twofold. Firstly, the volume of a segment connecting[1] semantics of two parents may become zero in higher dimensionality, in which case the likelihood of producing a program on such a segment becomes very small. Secondly, the odds for existence of two programs in the population that form a segment that incorporates the target drops rapidly with growing number of dimensions.

The highest probability of L_1-geometric selections is achieved by CTS for all problems and for each population initialization method. Differences between the two other operators seem to be insignificant. We conduct Friedman's test that results in conclusive p-value of 2.67×10^{-6} and

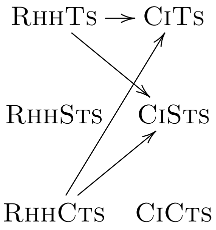
[1] A quantity of a space occupied by the segment.

Table 9.17: Empirical probability and 95% confidence interval of L_1-geometric selections conducted by TS, STS and CTS (the highest in bold).

Problem	RhhTs	RhhSts	RhhCts	CiTs	CiSts	CiCts
Par5	0.047 ±0.002	0.044 ±0.002	**0.214** ±0.005	0.004 ±0.001	0.006 ±0.001	0.012 ±0.001
Par6	0.037 ±0.002	0.032 ±0.002	**0.165** ±0.004	0.003 ±0.001	0.003 ±0.001	0.006 ±0.001
Par7	0.028 ±0.002	0.026 ±0.002	**0.137** ±0.004	0.001 ±0.000	0.001 ±0.000	0.002 ±0.000
Mux6	0.000 ±0.000	0.001 ±0.000	**0.009** ±0.001	0.000 ±0.000	0.000 ±0.000	0.002 ±0.000
Mux11	0.000 ±0.000	0.000 ±0.000	**0.000** ±0.000	0.000 ±0.000	0.000 ±0.000	0.000 ±0.000
Maj7	0.002 ±0.001	0.002 ±0.001	**0.035** ±0.002	0.001 ±0.000	0.001 ±0.000	0.004 ±0.001
Maj8	0.001 ±0.000	0.001 ±0.000	**0.004** ±0.001	0.000 ±0.000	0.000 ±0.000	0.001 ±0.000
Cmp6	0.019 ±0.002	0.019 ±0.002	**0.051** ±0.003	0.006 ±0.001	0.006 ±0.001	0.014 ±0.001
Cmp8	0.005 ±0.001	0.007 ±0.001	**0.021** ±0.002	0.000 ±0.000	0.000 ±0.000	0.002 ±0.001
Rank:	2.667	3.222	1.000	5.556	5.333	3.222

Table 9.18: Post-hoc analysis of Friedman's test conducted on Table 9.17: p-values of incorrectly judging a setup in a row as selecting more L_1-geometric mate parents than a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in the graph.

	RhhTs	RhhSts	RhhCts	CiTs	CiSts	CiCts
RhhTs		0.988		**0.012**	**0.028**	0.988
RhhSts				0.083	0.153	
RhhCts	0.401	0.114		**0.000**	**0.000**	0.114
CiTs						
CiSts				1.000		
CiCts		1.000		0.083	0.153	



present post-hoc analysis in Table 9.18. The evidence is too weak to confirm dominance of CTS over other operators at the significance level 0.05, however the obtained p-values only slightly exceed the significance level. Our observation that there are no significant differences between ST and STS within population initialization method is confirmed, and significant differences are only observed across initialization methods.

The corresponding probabilities for symbolic regression and L_2 metric are rather small for all operators and not suitable to draw meaningful conclusions. This is because L_2 segment has zero volume. To overcome this, we present the same statistics for L_1 metric in Table 9.19. The obtained values are higher here because L_1 segment often has non-zero volume.

Table 9.19: Empirical probability and 95% confidence interval of L_1-geometric selections conducted by TS, STS and CTS in regression problems (the highest in bold).

Problem	RhhTs	RhhSts	RhhCts	CiTs	CiSts	CiCts
R1	0.001 ±0.000	0.001 ±0.000	**0.001** ±0.000	0.000 ±0.000	0.000 ±0.000	0.000 ±0.000
R2	0.000 ±0.000	0.000 ±0.000	**0.001** ±0.000	0.000 ±0.000	0.000 ±0.000	0.000 ±0.000
R3	0.010 ±0.001	0.008 ±0.001	**0.024** ±0.002	0.004 ±0.001	0.003 ±0.001	0.006 ±0.001
Kj1	0.008 ±0.001	0.008 ±0.001	**0.018** ±0.002	0.002 ±0.001	0.001 ±0.000	0.003 ±0.001
Kj4	0.000 ±0.000	0.000 ±0.000	0.000 ±0.000	0.000 ±0.000	0.000 ±0.000	**0.001** ±0.000
Ng9	0.000 ±0.000	0.000 ±0.000	**0.001** ±0.000	0.000 ±0.000	0.000 ±0.000	0.001 ±0.000
Ng12	0.002 ±0.001	0.002 ±0.000	**0.004** ±0.001	0.001 ±0.000	0.001 ±0.000	0.001 ±0.000
Pg1	**0.001** ±0.000	0.001 ±0.000	0.001 ±0.000	0.000 ±0.000	0.000 ±0.000	0.000 ±0.000
Vl1	0.001 ±0.000	0.001 ±0.000	0.002 ±0.000	0.001 ±0.000	0.001 ±0.000	**0.002** ±0.000
Rank:	2.833	3.722	1.556	4.778	5.000	3.111

Table 9.20: Post-hoc analysis of Friedman's test conducted on Table 9.19: p-values of incorrectly judging a setup in a row as selecting more L_1-geometric mate parents than a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RhhTs	RhhSts	RhhCts	CiTs	CiSts	CiCts
RhhTs		0.915		0.234	0.136	1.000
RhhSts				0.838	0.695	
RhhCts	0.696	0.136		**0.004**	**0.001**	0.488
CiTs					1.000	
CiSts						
CiCts		0.983		0.406	0.264	

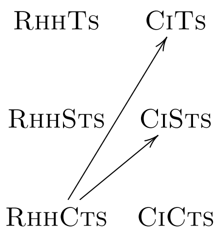


Table 9.21: Average and 95% confidence interval of time (milliseconds) consumed to select two parents from population of size 1000 by TS, STS and CTS, respectively (the lowest in bold).

Problem	RhhTs	RhhSts	RhhCts	CiTs	CiSts	CiCts
R1	**2.15** ±0.13	7.70 ±0.63	18.16 ±2.12	2.45 ±0.10	4.69 ±0.17	9.81 ±0.48
R2	**2.22** ±0.09	9.54 ±0.66	19.04 ±1.76	2.36 ±0.12	4.88 ±0.24	9.54 ±0.44
R3	**2.24** ±0.12	9.00 ±0.70	19.13 ±1.85	2.37 ±0.12	4.72 ±0.20	9.43 ±0.51
Kj1	**2.24** ±0.12	9.81 ±1.06	16.70 ±1.31	2.47 ±0.09	4.74 ±0.22	9.83 ±0.43
Kj4	2.18 ±0.09	8.76 ±0.80	18.38 ±2.04	**2.11** ±0.13	4.47 ±0.21	9.11 ±0.47
Ng9	**1.61** ±0.06	9.67 ±0.22	19.37 ±0.76	1.70 ±0.05	5.89 ±0.20	10.37 ±0.31
Ng12	**1.57** ±0.04	9.73 ±0.24	19.32 ±0.47	1.72 ±0.06	5.61 ±0.18	10.72 ±0.33
Pg1	**1.53** ±0.05	9.64 ±0.26	18.95 ±0.68	1.82 ±0.07	5.81 ±0.20	11.32 ±0.78
Vl1	**1.59** ±0.05	9.76 ±0.17	19.29 ±0.60	1.79 ±0.08	6.03 ±0.25	11.11 ±0.56
Par5	2.43 ±0.06	7.03 ±0.28	12.59 ±0.71	**1.76** ±0.09	3.59 ±0.21	7.88 ±0.50
Par6	2.15 ±0.11	5.67 ±0.33	10.42 ±0.44	**1.94** ±0.14	3.76 ±0.24	8.29 ±1.03
Par7	2.06 ±0.14	5.57 ±0.24	10.80 ±0.61	**1.96** ±0.15	3.87 ±0.28	8.08 ±0.58
Mux6	2.30 ±0.12	5.80 ±0.36	10.45 ±0.58	**1.96** ±0.18	3.58 ±0.25	7.54 ±0.55
Mux11	**1.72** ±0.10	5.01 ±0.31	9.72 ±0.65	1.80 ±0.10	3.96 ±0.21	8.49 ±0.45
Maj7	2.38 ±0.14	6.01 ±0.36	10.35 ±0.61	**1.87** ±0.15	3.62 ±0.24	7.68 ±0.53
Maj8	1.96 ±0.12	5.41 ±0.20	9.84 ±0.65	**1.91** ±0.14	3.86 ±0.32	7.67 ±0.54
Cmp6	2.07 ±0.13	5.22 ±0.28	10.03 ±0.55	**1.85** ±0.14	3.72 ±0.23	7.35 ±0.55
Cmp8	1.99 ±0.11	5.21 ±0.27	9.47 ±0.54	**1.82** ±0.11	3.47 ±0.20	7.56 ±0.42
Rank:	1.50	4.00	6.00	1.50	3.00	5.00

Conclusions are similar to these in the Boolean domain. The values obtained for the RHH-initialized populations are higher than the corresponding values for CI. CTS obtains the highest probability for 8 out of 9 problems for RHH's population and in 6 out of 9 problems for CI's one. Friedman's test conducted on Table 9.19 results in conclusive p-value of 1.27×10^{-3}, and post-hoc analysis in Table 9.20 for $\alpha = 0.05$ indicates significant differences only between setups using different initialization methods. CTS for RHH's population has significantly more L_1-geometric applications than TS and STS for CI's population.

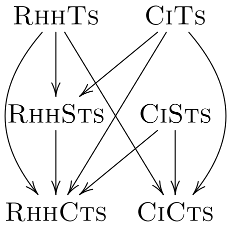
9.2.2 Computational Costs

Table 9.21 presents average and 95% confidence interval of CPU time consumed to select two parents: one using TS and one using TS, STS or CTS, respectively.

TS is the quickest operator, STS requires three to five times more time than TS, and CTS roughly two times more than STS. The times for TS seem to be unaffected by distribution of programs in population. The times for STS and CTS are about 30% − 50% lower for CI's population than for the RHH's one.

Table 9.22: Post-hoc analysis of Friedman's test conducted on Table 9.21: p-values of incorrectly judging a setup in a row as faster than a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RhhTs	RhhSts	RhhCts	CiTs	CiSts	CiCts
RhhTs		**0.001**	**0.000**		0.154	**0.000**
RhhSts			**0.017**			0.596
RhhCts						
CiTs	1.000	**0.001**	**0.000**		0.154	**0.000**
CiSts		0.596	**0.000**			**0.017**
CiCts			0.596			



Given that neither STS nor CTS feature computationally-heavy branches of code in their algorithms, both of them perform a fixed number of iterations μ, and population size and dimensionality of semantic space are constant, we did not see algorithmic reasons for this state of affairs and sought to technical aspects of hardware. We conduct an extra experiment that shows that the speedup is an effect of varying performance of floating point arithmetic. If an argument or outcome of a floating point operation is subnormal (denormal) value, infinity or NaN [72], the operation may consume even hundreds of times more CPU cycles than the same operation for normal values [95, 43]. Subnormal values may occur in fitness function, semantic distance and/or candidate selection criterion in CTS.

To statistically verify differences between operators, we conduct Friedman's test that results in conclusive p-value of 1.96×10^{-12}. Post-hoc analysis shown in Table 9.22 chiefly confirms our previous observations: TS is the quickest operator, STS is slower than TS, and CTS is slower than STS. There are no significant differences in time w.r.t. initialization method.

9.3 Mutation

This section is intended to answer research question 3 by running full evolutionary setups employed with respectively TM, SDM and CTM, and collecting statistics of training and test-set fitness, effectiveness, geometry, size of produced programs and consumed time. Moreover to verify whether starting conditions imposed by CI have impact on the evolutionary search we double each setup, i.e., run it once with population initialized by CI, once by RHH. Since mutations take only one parent, we do not involve mate selection operators and remain with TS. Namely we compare eight setups: RhhTsTm, RhhTsSdm, RhhTsSgm, RhhTsCm, CiTsTm, CiTsSdm, CiTsSgm, CiTsCm.

9.3.1 Fitness-based performance

In Figure 9.2 we present average and 95% confidence interval of the best of generation fitness and in Table 9.23 of the best of run fitness. CM achieves better fitness and converges much quicker than the other operators, and the fitness achieved by CM as of $30 - 50$ generation, depending on a problem, remains unbeatable by all other operators for the rest of run. One exception is relatively simple problem Maj8 that is solved also by SDM and SGM with almost the same success. Although other operators can be ordered from better to worse for most of the problems, differences of their fitness are relatively low, confidence intervals overlap and the orderings are not consistent across problems. Hence it is difficult to unambiguously determine a runner up.

In 13 out of 18 problems CM starting from RHH's population achieves better and in 3 out of 18 equal best of run fitness to CM starting from CI's population. TM with RHH achieves better fitness

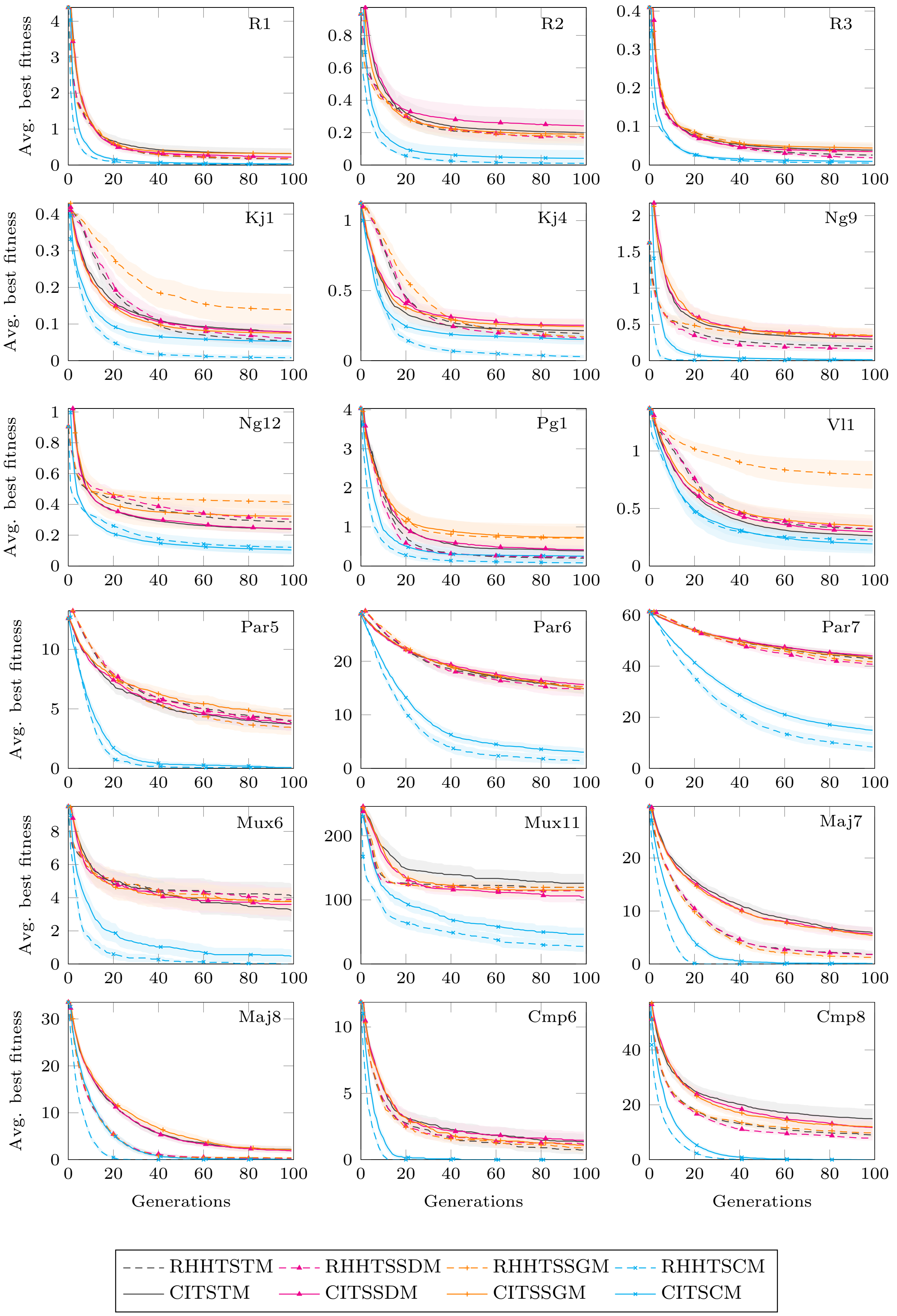


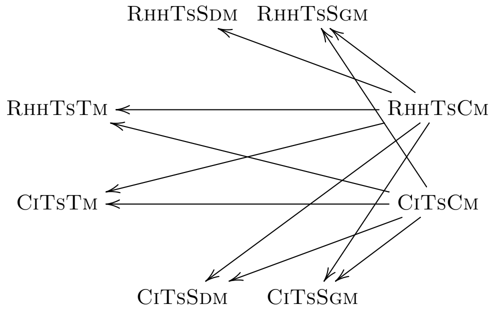
Figure 9.2: Average and 95% confidence interval of the best of generation fitness.

Table 9.23: Average and 95% confidence interval of the best of run fitness (the best in bold).

Prob.	RhhTsTm	RhhTsSdm	RhhTsSgm	RhhTsCm	CiTsTm	CiTsSdm	CiTsSgm	CiTsCm
R1	0.17 ±0.05	0.17 ±0.05	0.18 ±0.05	**0.02** ±0.00	0.32 ±0.15	0.22 ±0.07	0.32 ±0.08	0.03 ±0.01
R2	0.19 ±0.03	0.17 ±0.03	0.18 ±0.03	**0.01** ±0.00	0.20 ±0.08	0.24 ±0.10	0.19 ±0.06	0.04 ±0.05
R3	0.03 ±0.00	0.02 ±0.00	0.03 ±0.01	**0.00** ±0.00	0.04 ±0.02	0.03 ±0.01	0.04 ±0.01	0.01 ±0.00
Kj1	0.05 ±0.02	0.06 ±0.02	0.14 ±0.04	**0.01** ±0.01	0.08 ±0.02	0.08 ±0.02	0.08 ±0.01	0.05 ±0.02
Kj4	0.19 ±0.03	0.16 ±0.03	0.17 ±0.03	**0.03** ±0.01	0.21 ±0.05	0.25 ±0.05	0.24 ±0.05	0.15 ±0.04
Ng9	0.19 ±0.05	0.16 ±0.05	0.35 ±0.09	**0.00** ±0.00	0.30 ±0.10	0.34 ±0.09	0.35 ±0.10	0.01 ±0.01
Ng12	0.29 ±0.04	0.30 ±0.05	0.41 ±0.05	0.12 ±0.04	0.24 ±0.03	0.24 ±0.03	0.32 ±0.10	**0.10** ±0.03
Pg1	0.20 ±0.08	0.23 ±0.09	0.71 ±0.37	**0.08** ±0.06	0.39 ±0.11	0.41 ±0.12	0.73 ±0.28	0.25 ±0.19
Vl1	0.32 ±0.08	0.32 ±0.09	0.79 ±0.12	0.22 ±0.12	0.26 ±0.04	0.30 ±0.05	0.35 ±0.10	**0.19** ±0.07
Par5	3.83 ±0.65	4.00 ±0.64	3.43 ±0.64	**0.07** ±0.09	3.73 ±0.54	3.73 ±0.64	4.37 ±0.56	**0.07** ±0.09
Par6	15.10 ±1.30	14.67 ±1.34	15.30 ±1.34	**1.40** ±0.74	14.87 ±0.96	15.53 ±0.86	14.83 ±0.69	3.00 ±0.77
Par7	42.90 ±1.83	40.70 ±1.88	41.70 ±1.70	**8.30** ±1.88	43.33 ±1.57	43.93 ±1.27	43.20 ±1.44	14.73 ±1.66
Mux6	4.10 ±0.74	3.90 ±0.73	3.73 ±0.71	**0.00** ±0.00	3.20 ±0.71	3.60 ±0.75	3.77 ±0.83	0.47 ±0.40
Mux11	119.53 ±8.71	114.20 ±8.67	119.40 ±7.04	**27.47** ±8.35	125.93 ±14.63	104.07 ±10.21	115.27 ±10.45	46.40 ±10.35
Maj7	1.80 ±0.73	1.77 ±0.48	1.23 ±0.49	**0.00** ±0.00	5.83 ±0.99	5.60 ±1.30	5.37 ±0.76	0.10 ±0.19
Maj8	0.17 ±0.16	0.27 ±0.16	0.40 ±0.22	**0.00** ±0.00	2.20 ±0.68	1.87 ±0.62	2.13 ±0.61	0.07 ±0.09
Cmp6	0.73 ±0.32	1.13 ±0.37	0.87 ±0.33	**0.00** ±0.00	1.37 ±0.60	1.47 ±0.65	1.07 ±0.32	**0.00** ±0.00
Cmp8	8.93 ±0.80	7.80 ±0.86	9.67 ±1.31	**0.00** ±0.00	14.90 ±3.54	11.77 ±2.69	11.87 ±1.78	**0.00** ±0.00
Rank:	4.67	4.22	5.56	1.19	5.97	6.03	6.44	1.92

Table 9.24: Post-hoc analysis of Friedman's test conducted on Table 9.23: p-values of incorrectly judging a setup in a row as outranking a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RhhTsTm	RhhTsSdm	RhhTsSgm	RhhTsCm	CiTsTm	CiTsSdm	CiTsSgm	CiTsCm
RhhTsTm			0.959		0.750	0.707	0.363	
RhhTsSdm	0.999		0.729		0.385	0.344	0.115	
RhhTsSgm					1.000	0.999	0.959	
RhhTsCm	**0.001**	**0.005**	**0.000**		**0.000**	**0.000**	**0.000**	0.987
CiTsTm						1.000	0.999	
CiTsSdm							1.000	
CiTsSgm								
CiTsCm	**0.017**	0.088	**0.000**		**0.000**	**0.000**	**0.000**	



than with CI in 12 out of 18 problems, SDM in 13 out of 18, SGM in 12 out of 18, in remaining problems CI leads to not worse results.

In 14 out of 18 problems RhhTsCx is characterized by the narrowest confidence intervals, hence the lowest variance of the best of run fitness. In this sense RhhTsCx is the most stable and reliable setup.

To statistically verify whether CM is better than the remaining operators and initialization method influences the best of run fitness, we conduct Friedman's test. The test results in p-value of 2.49×10^{-9} and at significance level $\alpha = 0.05$ there is at least one significant difference between setups. Post-hoc analysis in Table 9.24 confirms our findings about outstanding performance of CM. The evidence is too weak to show that initializing using RHH leads to better results than using CI.

Table 9.25 shows empirical probability accompanied with 95% confidence interval of finding an optimal program p^*, i.e., $s(p^*) = t$ within 100 generations (known as *success rate* in GP).[2] For a half of problems none of the operators found the optimal program in any run, where 8 out of 9 unsolved problems are symbolic regression ones. On the other hand for 5 out of 18 problems there exists at least one operator that solves the problem in every run. The highest probability of solving each problem belongs to CM. In case CM starts from RHH-initialized population, it solves in every run 5 out of 18 problems, with total of 9 problems having non-zero probability of being solved. To compare, CiTsCm perfectly solves 2 out of 18 problems, and has non-zero probability for 8 problems. From other operators, each solves at least one problem at least once, however there is no clear runner up.

Friedman's test conducted on probability of success results in conclusive p-value of 1.10×10^{-5} and post-hoc analysis in Table 9.26 confirms our above findings.

9.3.2 Generalization abilities

Figure 9.3 presents median and 95% confidence interval of test-set fitness achieved by the best of generation program during the course of evolution and Table 9.27 shows the same data for the best of run program on training-set. We use median to overcome bias coming from outlying values that are common in our data due to, e.g., overfitting to the training-set.

For all problems except Pg1 CM achieves the best test-set fitness, where in 7 out of 8 problems by the setup that involves RHH and in one problem CI. Note that confidence intervals of both CM setups overlap, thus it is difficult to tell that RHH leads to significantly better performance of CM than CI. For Pg1 problem test-set fitness of both CM setups are in the tail of the ranking, however all operators achieve similar fitness here. It is difficult to differentiate remaining operators, since all of them achieve similar fitness for each problem and the ordering of operators varies across problems.

Note that in Kj1, Kj4, Pg1 and Vl1 problems, test-set fitness is deteriorating in later generations for most setups, hence we conclude that arithmetic expressions behind these problems are difficult to be rebuilt from data and most setups suffer from overfitting there. For the remaining problems we do not notice overfitting for any setup.

To compare setup, we conduct Friedman's test on Table 9.27. The test is conclusive with p-value of 2.93×10^{-3}, thus we carry out post-hoc analysis in Table 9.28. CM beginning from RHH's population achieves significantly better test-set fitness than SGM for both ways of initialization and SDM and TM for CI initialization.

Table 9.25: Empirical probability and 95% confidence interval of finding the optimal solution in 100 generations (the best in bold).

Problem	RhhTsTm	RhhTsSdm	RhhTsSgm	RhhTsCm	CiTsTm	CiTsSdm	CiTsSgm	CiTsCm
R1	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
R2	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
R3	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
Kj1	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
Kj4	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
Ng9	0.07 ±0.09	0.13 ±0.12	0.17 ±0.13	**0.90** ±0.11	0.00 ±0.00	0.03 ±0.06	0.03 ±0.06	0.43 ±0.18
Ng12	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
Pg1	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
Vl1	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
Par5	0.03 ±0.06	0.03 ±0.06	0.10 ±0.11	**0.93** ±0.09	0.00 ±0.00	0.03 ±0.06	0.03 ±0.06	**0.93** ±0.09
Par6	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	**0.37** ±0.17	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.03 ±0.06
Par7	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	**0.03** ±0.06	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
Mux6	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	**1.00** ±0.00	0.03 ±0.06	0.03 ±0.06	0.03 ±0.06	0.83 ±0.13
Mux11	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
Maj7	0.23 ±0.15	0.20 ±0.14	0.30 ±0.16	**1.00** ±0.00	0.00 ±0.00	0.03 ±0.06	0.00 ±0.00	0.97 ±0.06
Maj8	0.87 ±0.12	0.73 ±0.16	0.67 ±0.17	**1.00** ±0.00	0.17 ±0.13	0.17 ±0.13	0.17 ±0.13	0.93 ±0.09
Cmp6	0.50 ±0.18	0.33 ±0.17	0.40 ±0.18	**1.00** ±0.00	0.33 ±0.17	0.33 ±0.17	0.27 ±0.16	**1.00** ±0.00
Cmp8	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	**1.00** ±0.00	0.00 ±0.00	0.00 ±0.00	0.03 ±0.06	**1.00** ±0.00
Rank:	4.69	4.92	4.56	2.83	5.42	5.11	5.14	3.33

Table 9.26: Post-hoc analysis of Friedman's test conducted on Table 9.25: p-values of incorrectly judging a setup in a row as outranking a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RhhTsTm	RhhTsSdm	RhhTsSgm	RhhTsCm	CiTsTm	CiTsSdm	CiTsSgm	CiTsCm
RhhTsTm		1.000			0.863	0.993	0.990	
RhhTsSdm					0.980	1.000	1.000	
RhhTsSgm	1.000	0.997			0.717	0.963	0.952	
RhhTsCm	**0.008**	**0.002**	**0.021**		**0.000**	**0.000**	**0.000**	0.980
CiTsTm								
CiTsSdm					0.999		1.000	
CiTsSgm					0.999			
CiTsCm	0.150	**0.048**	0.268		**0.002**	**0.015**	**0.012**	

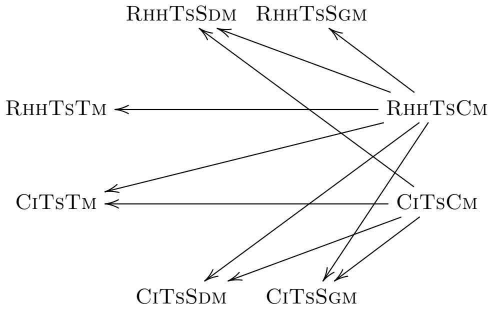


Table 9.27: Median and 95% confidence interval of test-set fitness of the best of run individual on training-set (the best in bold).

Problem	RhhTsTm	RhhTsSdm	RhhTsSgm	RhhTsCm	CiTsTm	CiTsSdm	CiTsSgm	CiTsCm
R1	0.11 ≤ 0.16 ≤ 0.30	0.13 ≤ 0.16 ≤ 0.39	0.12 ≤ 0.17 ≤ 0.23	0.01 ≤ **0.02** ≤ 0.09	0.11 ≤ 0.22 ≤ 0.89	0.10 ≤ 0.14 ≤ 0.42	0.18 ≤ 0.27 ≤ 0.41	0.02 ≤ 0.04 ≤ 0.18
R2	0.08 ≤ 0.11 ≤ 0.16	0.07 ≤ 0.08 ≤ 0.14	0.08 ≤ 0.10 ≤ 0.16	0.01 ≤ **0.02** ≤ 0.03	0.08 ≤ 0.16 ≤ 0.31	0.09 ≤ 0.12 ≤ 0.23	0.12 ≤ 0.16 ≤ 0.50	0.02 ≤ 0.04 ≤ 0.07
R3	0.02 ≤ 0.03 ≤ 0.06	0.01 ≤ 0.02 ≤ 0.03	0.02 ≤ 0.03 ≤ 0.06	0.01 ≤ 0.01 ≤ 0.02	0.02 ≤ 0.03 ≤ 0.06	0.03 ≤ 0.04 ≤ 0.08	0.02 ≤ 0.04 ≤ 0.06	0.00 ≤ **0.01** ≤ 0.02
Kj1	0.11 ≤ 0.17 ≤ 0.22	0.07 ≤ 0.19 ≤ 0.29	0.14 ≤ 0.20 ≤ 0.28	0.04 ≤ **0.08** ≤ 0.18	0.12 ≤ 0.18 ≤ 0.25	0.12 ≤ 0.19 ≤ 0.22	0.10 ≤ 0.11 ≤ 0.14	0.09 ≤ 0.14 ≤ 0.28
Kj4	0.62 ≤ 1.38 ≤ 2.49	1.21 ≤ 1.72 ≤ 2.99	0.95 ≤ 1.50 ≤ 2.48	0.50 ≤ **0.83** ≤ 1.81	2.01 ≤ 2.69 ≤ 4.30	1.75 ≤ 2.28 ≤ 3.46	2.00 ≤ 2.45 ≤ 3.91	1.05 ≤ 2.19 ≤ 2.67
Ng9	0.07 ≤ 0.12 ≤ 0.19	0.07 ≤ 0.11 ≤ 0.29	0.10 ≤ 0.15 ≤ 0.41	0.00 ≤ **0.00** ≤ 0.00	0.10 ≤ 0.15 ≤ 0.23	0.09 ≤ 0.16 ≤ 0.23	0.10 ≤ 0.15 ≤ 0.30	0.00 ≤ 0.00 ≤ 0.02
Ng12	0.10 ≤ 0.13 ≤ 0.21	0.10 ≤ 0.16 ≤ 0.26	0.20 ≤ 0.23 ≤ 0.30	0.03 ≤ **0.05** ≤ 0.10	0.09 ≤ 0.12 ≤ 0.22	0.10 ≤ 0.13 ≤ 0.17	0.13 ≤ 0.17 ≤ 0.20	0.05 ≤ 0.07 ≤ 0.11
Pg1	3.30 ≤ **3.89** ≤ 5.10	3.41 ≤ 3.90 ≤ 4.46	4.16 ≤ 4.54 ≤ 5.25	3.98 ≤ 4.34 ≤ 4.96	3.24 ≤ 4.16 ≤ 4.70	3.68 ≤ 4.19 ≤ 5.56	3.38 ≤ 4.07 ≤ 5.92	3.84 ≤ 4.72 ≤ 5.41
Vl1	0.41 ≤ 0.80 ≤ 1.45	0.45 ≤ 0.63 ≤ 1.69	1.89 ≤ 2.08 ≤ 2.14	0.37 ≤ **0.58** ≤ 1.49	0.99 ≤ 1.56 ≤ 1.93	0.94 ≤ 1.30 ≤ 2.12	0.59 ≤ 1.10 ≤ 2.55	0.78 ≤ 1.31 ≤ 1.68
Rank:	3.56	3.89	6.11	1.67	5.78	5.67	5.89	3.44

Table 9.28: Post-hoc analysis of Friedman's test conducted on Table 9.27: p-values of incorrectly judging a setup in a row as outranking a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RhhTsTm	RhhTsSdm	RhhTsSgm	RhhTsCm	CiTsTm	CiTsSdm	CiTsSgm	CiTsCm
RhhTsTm		1.000	0.343		0.534	0.600	0.467	
RhhTsSdm			0.534		0.728	0.786	0.666	
RhhTsSgm								
RhhTsCm	0.729	0.534	**0.003**		**0.009**	**0.013**	**0.006**	0.786
CiTsTm			1.000				1.000	
CiTsSdm			1.000		1.000		1.000	
CiTsSgm			1.000					
CiTsCm	1.000	1.000	0.289		0.468	0.534	0.404	

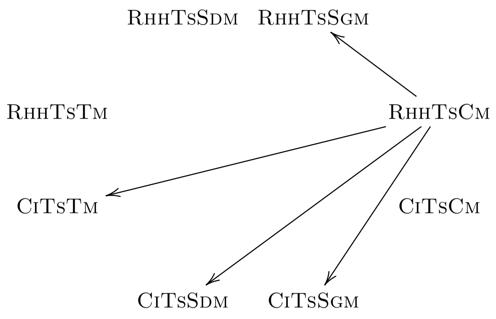
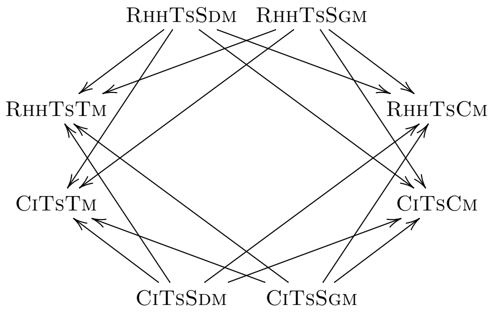


Table 9.29: Empirical probability and 95% confidence interval of effective mutations until 100 generation (the highest in bold).

Prob.	RhhTsTm	RhhTsSdm	RhhTsSgm	RhhTsCm	CiTsTm	CiTsSdm	CiTsSgm	CiTsCm
R1	0.883 ±0.002	1.000 ±0.000	**1.000** ±0.000	0.930 ±0.001	0.895 ±0.002	**1.000** ±0.000	**1.000** ±0.000	0.926 ±0.001
R2	0.882 ±0.002	1.000 ±0.000	**1.000** ±0.000	0.922 ±0.001	0.879 ±0.002	**1.000** ±0.000	**1.000** ±0.000	0.926 ±0.001
R3	0.872 ±0.002	1.000 ±0.000	**1.000** ±0.000	0.941 ±0.001	0.894 ±0.002	1.000 ±0.000	**1.000** ±0.000	0.941 ±0.001
Kj1	0.886 ±0.002	1.000 ±0.000	**1.000** ±0.000	0.921 ±0.001	0.899 ±0.001	1.000 ±0.000	**1.000** ±0.000	0.949 ±0.001
Kj4	0.858 ±0.002	1.000 ±0.000	**1.000** ±0.000	0.954 ±0.001	0.909 ±0.001	1.000 ±0.000	**1.000** ±0.000	0.977 ±0.001
Ng9	0.906 ±0.001	1.000 ±0.000	**1.000** ±0.000	0.956 ±0.003	0.903 ±0.001	**1.000** ±0.000	**1.000** ±0.000	0.929 ±0.002
Ng12	0.921 ±0.001	**1.000** ±0.000	**1.000** ±0.000	0.947 ±0.001	0.890 ±0.002	1.000 ±0.000	**1.000** ±0.000	0.943 ±0.001
Pg1	0.898 ±0.001	1.000 ±0.000	**1.000** ±0.000	0.908 ±0.001	0.903 ±0.001	1.000 ±0.000	**1.000** ±0.000	0.926 ±0.001
Vl1	0.903 ±0.001	1.000 ±0.000	**1.000** ±0.000	0.944 ±0.001	0.912 ±0.001	1.000 ±0.000	**1.000** ±0.000	0.969 ±0.001
Par5	0.453 ±0.002	**0.997** ±0.000	0.906 ±0.001	0.440 ±0.004	0.375 ±0.002	0.997 ±0.000	0.908 ±0.001	0.385 ±0.004
Par6	0.479 ±0.002	0.998 ±0.000	0.915 ±0.001	0.471 ±0.003	0.443 ±0.002	**0.998** ±0.000	0.923 ±0.001	0.438 ±0.002
Par7	0.535 ±0.002	**0.999** ±0.000	0.929 ±0.001	0.526 ±0.002	0.475 ±0.002	0.999 ±0.000	0.929 ±0.001	0.495 ±0.002
Mux6	0.242 ±0.002	0.962 ±0.001	0.899 ±0.001	0.371 ±0.005	0.248 ±0.002	**0.974** ±0.001	0.906 ±0.001	0.288 ±0.003
Mux11	0.222 ±0.002	0.958 ±0.001	0.922 ±0.001	0.316 ±0.002	0.256 ±0.002	**0.975** ±0.001	0.922 ±0.001	0.300 ±0.002
Maj7	0.538 ±0.003	**0.999** ±0.000	0.935 ±0.001	0.610 ±0.006	0.486 ±0.002	0.999 ±0.000	0.936 ±0.001	0.443 ±0.004
Maj8	0.615 ±0.003	**1.000** ±0.000	0.940 ±0.002	0.685 ±0.005	0.500 ±0.003	0.999 ±0.000	0.942 ±0.001	0.508 ±0.004
Cmp6	0.399 ±0.003	0.992 ±0.000	0.919 ±0.001	0.619 ±0.008	0.362 ±0.003	**0.993** ±0.000	0.920 ±0.001	0.376 ±0.006
Cmp8	0.487 ±0.002	**0.997** ±0.000	0.939 ±0.001	0.545 ±0.004	0.465 ±0.002	0.997 ±0.000	0.939 ±0.001	0.438 ±0.004
Rank:	6.889	2.444	2.750	5.500	7.389	2.333	2.472	6.222

Table 9.30: Post-hoc analysis of Friedman's test conducted on Table 9.29: p-values of incorrectly judging a setup in a row as conducting more effective applications than a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RhhTsTm	RhhTsSdm	RhhTsSgm	RhhTsCm	CiTsTm	CiTsSdm	CiTsSgm	CiTsCm
RhhTsTm					0.999			
RhhTsSdm	**0.000**		1.000	**0.004**	**0.000**		1.000	**0.000**
RhhTsSgm	**0.000**			**0.016**	**0.000**			**0.000**
RhhTsCm	0.679				0.277			0.987
CiTsTm								
CiTsSdm	**0.000**	1.000	1.000	**0.002**	**0.000**		1.000	**0.000**
CiTsSgm	**0.000**		1.000	**0.005**	**0.000**			**0.000**
CiTsCm	0.992				0.839			



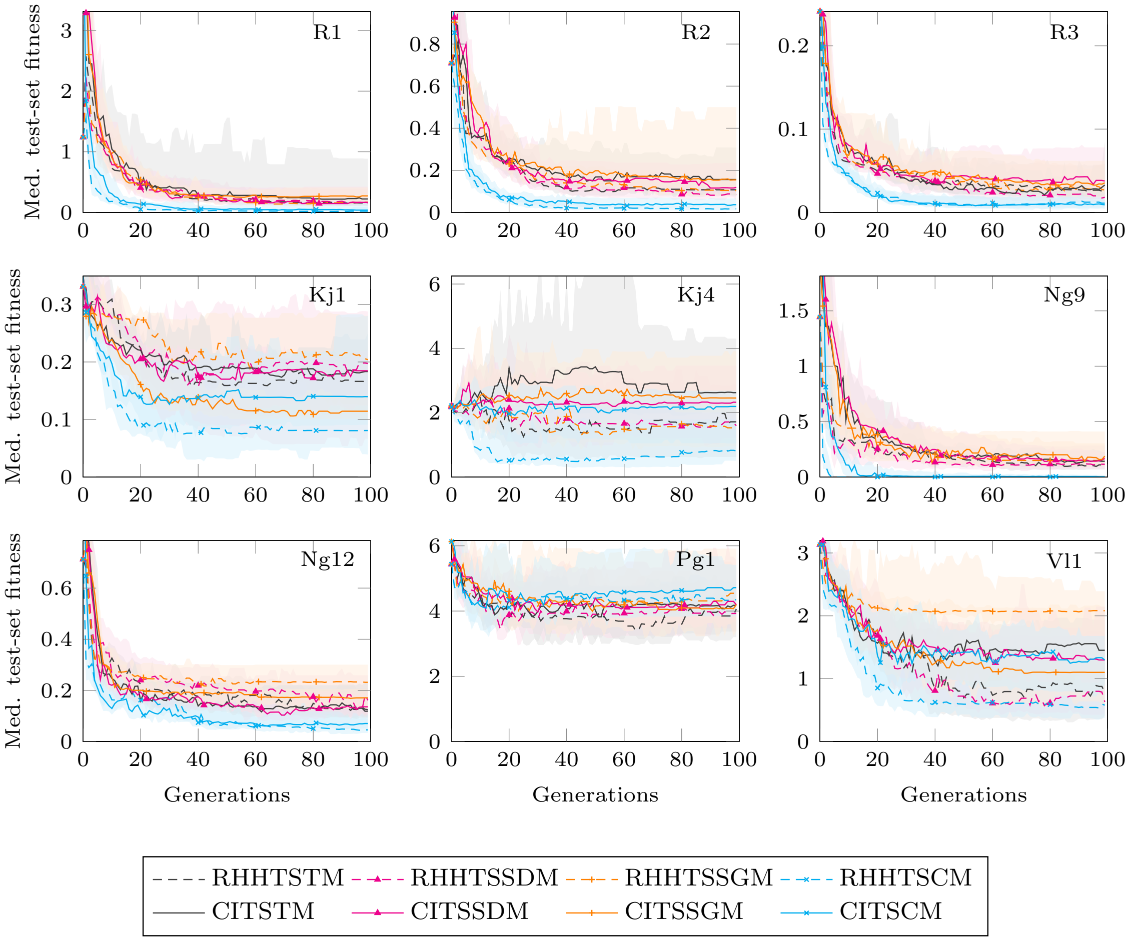


Figure 9.3: Median and 95% confidence interval of test-set fitness of the best of generation program on training-set.

9.3.3 Effectiveness and geometry

Figure 9.4 presents empirical probability and 95% confidence interval of mutations being effective (cf. Definition 4.7) over generations and Table 9.29 presents the same numbers for entire 100-generation runs.

The highest probability of almost 1.0 for all symbolic regression problems belongs SGM and for all Boolean problems to SDM. The second most effective operator in symbolic regression is SDM with values over 0.99 for the most part of runs and SGM in Boolean domain with values over 0.9. The least effective operator is TM in all regression problems and 2 out of 9 Boolean problems, for Maj7 CM is least effective. For the remaining Boolean problems both CM setups are in between TM setups for most part of the run. The imperfect effectiveness of CM may be a cause of a technical implementation of $\mathsf{Invert}(\cdot,\cdot,\cdot)$ function from Table 7.1 that returns at most two values, even if there are more inversions of parent semantics ($s(p)$ in Algorithm 7.4) available. Thus, forbidden semantics ($D_\varnothing$ in that algorithm) may not contain all values that prevent from producing neutral offspring.

Only SDM maintains roughly constant probability of effective mutations for entire run. For SGM and CM in the first few generations we see rapid fluctuations of the probability for some of the problems and the value stabilizes after approximately 20 generations. The fluctuations are also observed for an average number of unique semantics in population in Figure 9.5. They may

[2] We use similarity threshold, see Table 8.2.

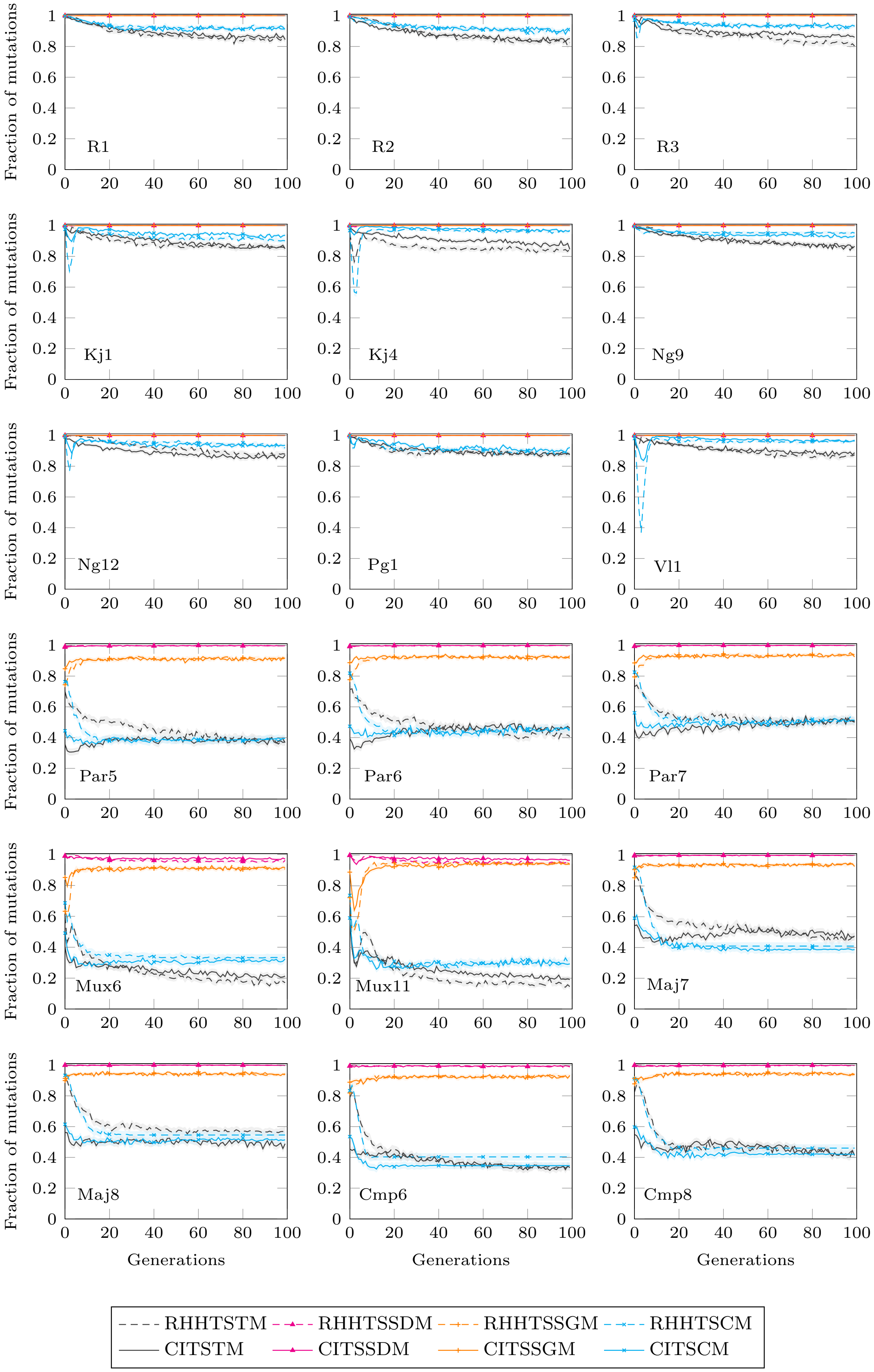


Figure 9.4: Empirical probability and 95% confidence interval of effective mutations per generation.

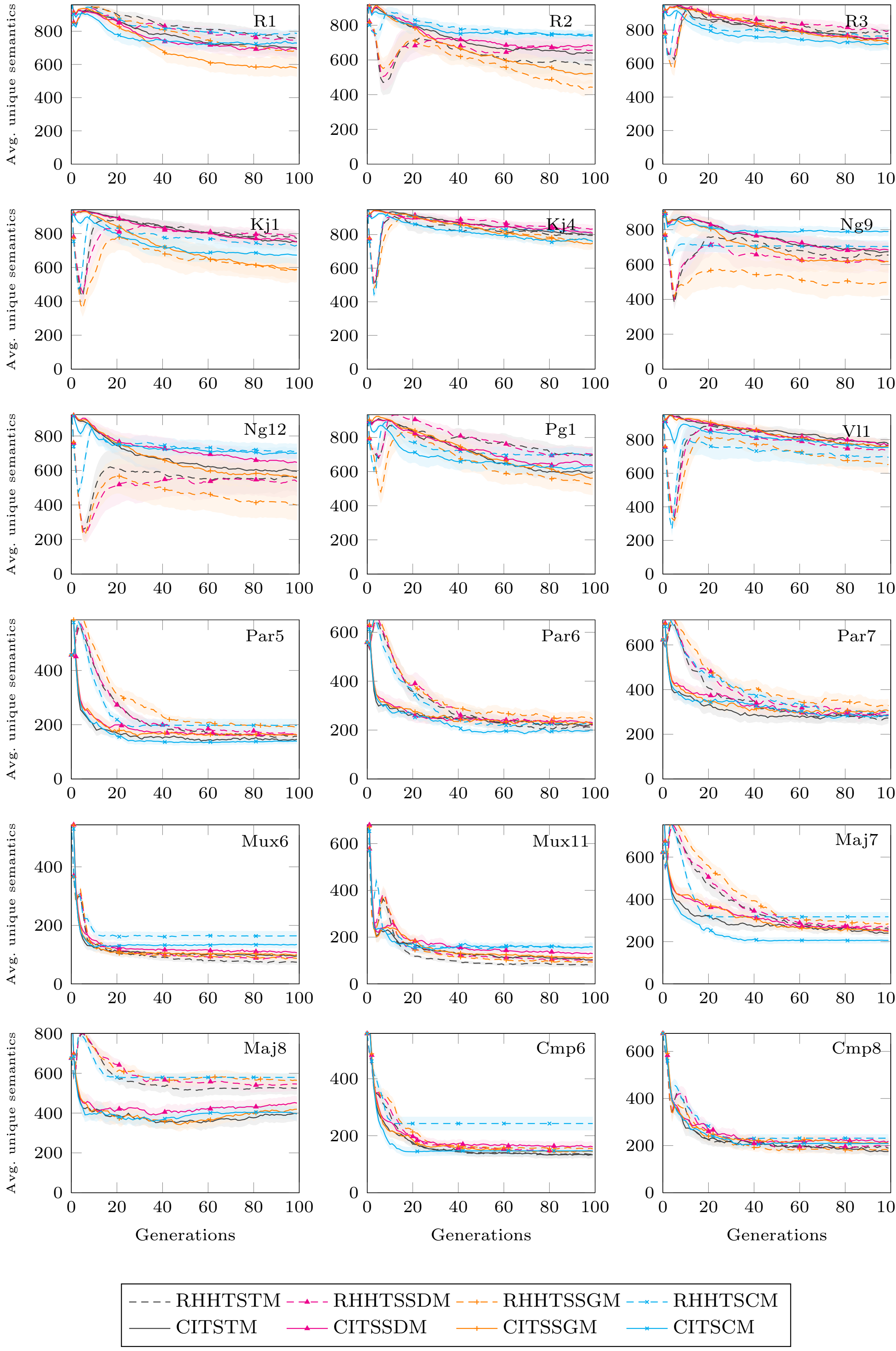


Figure 9.5: Average and 95% confidence interval of unique semantics in population of size 1000 over generations.

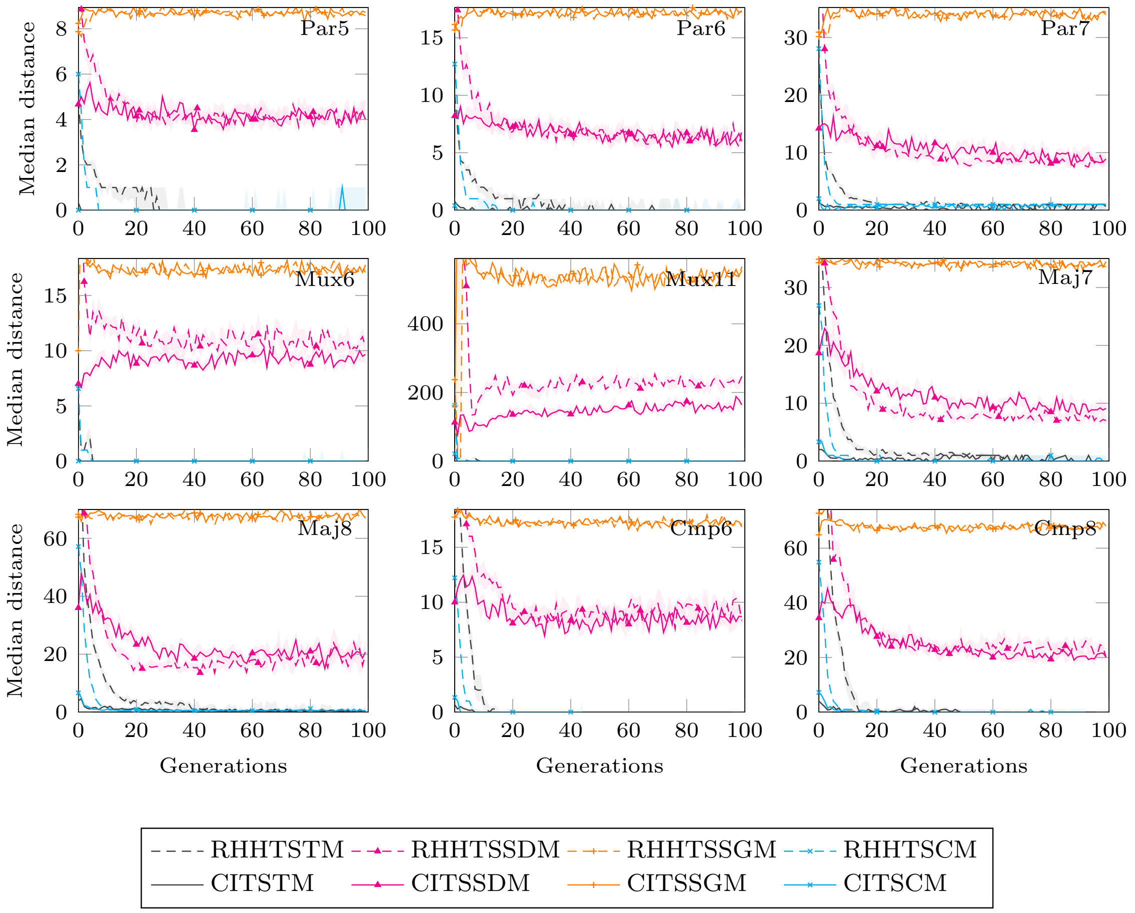


Figure 9.6: Median and 95% confidence interval of L_1 distance of offspring to parent over generations.

be an effect of dynamic changes in distribution of programs and semantics in population in early generations, when the operator transforms the distribution from the initial random one to a biased in an operator specific way. TM tends to slightly decrease probability of effective mutations over time. Decreasing number of distinct semantics over generations for all operators in Figure 9.5 clearly indicate convergence to a (local) optimum. However SGM and SDM are the only operators that maintain high probability of effective mutations even for converging populations.

Although SGM is characterized by over 90% of effective mutations, it also leads to the least diverse populations for 6 out of 9 symbolic regression problems. This phenomenon is not observed for Boolean domain. Given that SGM in symbolic regression only moves program semantics by a vector (cf. Section 6.2.1), we hypothesize that drop of diversity may be caused by suppressing mutations that move program semantics back to the location where it was a few generations earlier. Thus effectiveness of SGM from the perspective of multiple subsequent applications to the same program may be questionable.

We carry out Friedman's test to statistically verify differences between operators on the total probability of effective mutations. The test results in conclusive p-value of 4.05×10^{-9}, thus we conduct post-hoc analysis in Table 9.30. Our observations that SDM and SGM are the most effective operators are confirmed: both SDM and both SGM setups outperform all TM and CM setups.

Figure 9.6 and Table 9.31 present median and 95% confidence interval of L_1 semantic distance of offspring to its parent over generations and in the entire run, respectively, for Boolean problems. Figure 9.7 and Table 9.8 consist of the same values for L_2 distance and symbolic regression.

Table 9.31: Median and 95% confidence interval of L_1 distance of offspring to parent until 100 generation (the lowest in bold).

Problem	RhhTsTm	RhhTsSdm	RhhTsSgm	RhhTsCm	CiTsTm	CiTsSdm	CiTsSgm	CiTsCm
Par5	$0.00 \le$ **0.00** ≤ 0.00	$4.40 \le 4.44 \le 4.50$	$8.70 \le 8.71 \le 8.71$	$0.00 \le$ **0.00** ≤ 0.00	$0.00 \le$ **0.00** ≤ 0.00	$4.17 \le 4.20 \le 4.25$	$8.70 \le 8.70 \le 8.71$	$0.00 \le$ **0.00** ≤ 0.00
Par6	$0.40 \le 0.50 \le 0.50$	$7.11 \le 7.17 \le 7.25$	$17.12 \le 17.13 \le 17.14$	$0.00 \le$ **0.00** ≤ 0.00	$0.00 \le$ **0.00** ≤ 0.00	$6.71 \le 6.78 \le 6.83$	$17.16 \le 17.16 \le 17.17$	$0.00 \le$ **0.00** ≤ 0.00
Par7	$1.33 \le 1.40 \le 1.50$	$10.00 \le 10.00 \le 10.22$	$33.97 \le 33.98 \le 34.00$	$1.00 \le 1.00 \le 1.00$	$0.44 \le$ **0.50** ≤ 0.50	$10.60 \le 10.75 \le 10.86$	$34.00 \le 34.00 \le 34.02$	$0.67 \le 0.80 \le 1.00$
Mux6	$0.00 \le$ **0.00** ≤ 0.00	$11.18 \le 11.25 \le 11.33$	$17.35 \le 17.36 \le 17.38$	$0.00 \le$ **0.00** ≤ 0.00	$0.00 \le$ **0.00** ≤ 0.00	$9.07 \le 9.14 \le 9.20$	$17.32 \le 17.33 \le 17.34$	$0.00 \le$ **0.00** ≤ 0.00
Mux11	$0.00 \le$ **0.00** ≤ 0.00	$228.92 \le 230.40 \le 232.53$	$536.00 \le 536.29 \le 536.75$	$0.00 \le$ **0.00** ≤ 0.00	$0.00 \le$ **0.00** ≤ 0.00	$144.00 \le 145.00 \le 146.00$	$537.10 \le 537.70 \le 538.04$	$0.00 \le$ **0.00** ≤ 0.00
Maj7	$1.33 \le 1.33 \le 1.40$	$9.50 \le 9.60 \le 9.70$	$34.00 \le 34.00 \le 34.02$	$1.00 \le 1.00 \le 1.00$	$0.40 \le 0.50 \le 0.50$	$11.38 \le 11.50 \le 11.62$	$34.10 \le 34.11 \le 34.13$	$0.00 \le$ **0.00** ≤ 0.00
Maj8	$4.50 \le 4.67 \le 5.00$	$21.00 \le 21.29 \le 21.60$	$67.43 \le 67.46 \le 67.48$	$3.00 \le 3.00 \le 3.50$	$0.60 \le$ **0.67** ≤ 0.67	$22.50 \le 22.71 \le 23.00$	$67.64 \le 67.66 \le 67.68$	$0.67 \le$ **0.67** ≤ 0.67
Cmp6	$0.00 \le$ **0.00** ≤ 0.00	$10.20 \le 10.29 \le 10.36$	$17.27 \le 17.28 \le 17.29$	$1.00 \le 1.00 \le 1.00$	$0.00 \le$ **0.00** ≤ 0.00	$8.64 \le 8.73 \le 8.80$	$17.26 \le 17.27 \le 17.28$	$0.00 \le$ **0.00** ≤ 0.00
Cmp8	$0.33 \le 0.40 \le 0.50$	$29.00 \le 29.33 \le 29.60$	$67.79 \le 67.81 \le 67.84$	$1.00 \le 1.00 \le 1.00$	$0.00 \le$ **0.00** ≤ 0.00	$24.30 \le 24.50 \le 24.75$	$67.71 \le 67.72 \le 67.75$	$0.00 \le$ **0.00** ≤ 0.00
Rank:	3.17	5.67	7.44	2.94	1.94	5.33	7.56	1.94

Table 9.32: Post-hoc analysis of Friedman's test conducted on Table 9.31: p-values of incorrectly judging a setup in a row as producing offspring in lower distance to parent than a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RhhTsTm	RhhTsSdm	RhhTsSgm	RhhTsCm	CiTsTm	CiTsSdm	CiTsSgm	CiTsCm
RhhTsTm		0.336	**0.004**			0.531	**0.002**	
RhhTsSdm			0.761				0.700	
RhhTsSgm							1.000	
RhhTsCm	1.000	0.230	**0.002**			0.398	**0.001**	
CiTsTm	0.960	**0.021**	**0.000**	0.987		0.052	**0.000**	
CiTsSdm		1.000	0.566				0.497	
CiTsSgm								
CiTsCm	0.960	**0.021**	**0.000**	0.987	1.000	0.052	**0.000**	

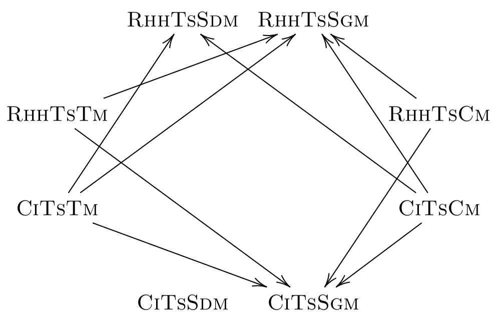
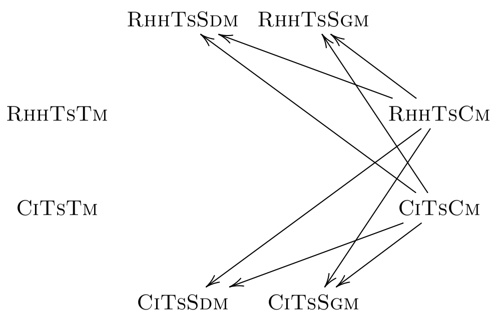


Table 9.33: Median and 95% confidence interval of L_2 distance of offspring to parent until 100 generation (the lowest in bold).

Problem	RhhTsTm	RhhTsSdm	RhhTsSgm	RhhTsCm	CiTsTm	CiTsSdm	CiTsSgm	CiTsCm
R1	$1.62 \le$ 1.65 ≤ 1.67	$3.53 \le$ 3.57 ≤ 3.62	$1.13 \le$ 1.13 ≤ 1.13	$0.10 \le$ **0.10** ≤ 0.10	$2.50 \le$ 2.53 ≤ 2.55	$4.59 \le$ 4.66 ≤ 4.71	$1.13 \le$ 1.13 ≤ 1.13	$0.25 \le$ 0.25 ≤ 0.26
R2	$0.50 \le$ 0.50 ≤ 0.51	$1.08 \le$ 1.08 ≤ 1.09	$1.13 \le$ 1.13 ≤ 1.13	$0.04 \le$ **0.04** ≤ 0.04	$0.62 \le$ 0.63 ≤ 0.64	$1.14 \le$ 1.15 ≤ 1.15	$1.13 \le$ 1.13 ≤ 1.13	$0.07 \le$ 0.07 ≤ 0.07
R3	$0.17 \le$ 0.17 ≤ 0.18	$0.60 \le$ 0.60 ≤ 0.61	$1.12 \le$ 1.12 ≤ 1.12	$0.04 \le$ **0.04** ≤ 0.04	$0.30 \le$ 0.30 ≤ 0.31	$0.62 \le$ 0.63 ≤ 0.64	$1.12 \le$ 1.12 ≤ 1.12	$0.06 \le$ 0.07 ≤ 0.07
Kj1	$0.25 \le$ 0.26 ≤ 0.26	$0.61 \le$ 0.62 ≤ 0.62	$1.13 \le$ 1.13 ≤ 1.13	$0.04 \le$ **0.04** ≤ 0.04	$0.28 \le$ 0.28 ≤ 0.29	$0.66 \le$ 0.66 ≤ 0.67	$1.12 \le$ 1.12 ≤ 1.12	$0.11 \le$ 0.12 ≤ 0.12
Kj4	$0.60 \le$ 0.61 ≤ 0.61	$1.03 \le$ 1.04 ≤ 1.05	$1.13 \le$ 1.13 ≤ 1.13	$0.17 \le$ **0.17** ≤ 0.18	$0.80 \le$ 0.81 ≤ 0.81	$1.12 \le$ 1.13 ≤ 1.14	$1.12 \le$ 1.12 ≤ 1.12	$0.51 \le$ 0.52 ≤ 0.52
Ng9	$0.94 \le$ 0.96 ≤ 0.97	$2.11 \le$ 2.16 ≤ 2.20	$2.53 \le$ 2.53 ≤ 2.53	$0.19 \le$ 0.20 ≤ 0.21	$1.15 \le$ 1.16 ≤ 1.17	$2.07 \le$ 2.10 ≤ 2.13	$2.51 \le$ 2.51 ≤ 2.52	$0.10 \le$ **0.10** ≤ 0.10
Ng12	$3.13 \le$ 3.18 ≤ 3.22	$5.38 \le$ 5.41 ≤ 5.45	$2.51 \le$ 2.51 ≤ 2.51	$0.15 \le$ 0.15 ≤ 0.15	$0.85 \le$ 0.87 ≤ 0.88	$1.81 \le$ 1.83 ≤ 1.85	$2.51 \le$ 2.51 ≤ 2.51	$0.13 \le$ **0.14** ≤ 0.14
Pg1	$1.66 \le$ 1.68 ≤ 1.69	$2.67 \le$ 2.71 ≤ 2.76	$2.53 \le$ 2.53 ≤ 2.53	$0.19 \le$ **0.20** ≤ 0.20	$1.79 \le$ 1.81 ≤ 1.83	$3.21 \le$ 3.24 ≤ 3.28	$2.52 \le$ 2.52 ≤ 2.52	$0.47 \le$ 0.48 ≤ 0.48
Vl1	$0.77 \le$ 0.78 ≤ 0.79	$2.14 \le$ 2.19 ≤ 2.23	$2.52 \le$ 2.52 ≤ 2.52	$0.29 \le$ **0.29** ≤ 0.29	$0.67 \le$ 0.68 ≤ 0.69	$1.34 \le$ 1.35 ≤ 1.37	$2.52 \le$ 2.52 ≤ 2.52	$0.46 \le$ 0.47 ≤ 0.47
Rank:	3.78	6.00	7.00	1.22	4.00	6.33	5.89	1.78

Table 9.34: Post-hoc analysis of Friedman's test conducted on Table 9.33: p-values of incorrectly judging a setup in a row as producing offspring in lower distance to parent than a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RhhTsTm	RhhTsSdm	RhhTsSgm	RhhTsCm	CiTsTm	CiTsSdm	CiTsSgm	CiTsCm
RhhTsTm		0.534	0.097		1.000	0.344	0.601	
RhhTsSdm			0.989			1.000		
RhhTsSgm								
RhhTsCm	0.344	**0.001**	**0.000**		0.238	**0.000**	**0.001**	1.000
CiTsTm		0.667	0.157			0.468	0.728	
CiTsSdm			0.999					
CiTsSgm		1.000	0.980			1.000		
CiTsCm	0.666	**0.006**	**0.000**		0.534	**0.002**	**0.009**	



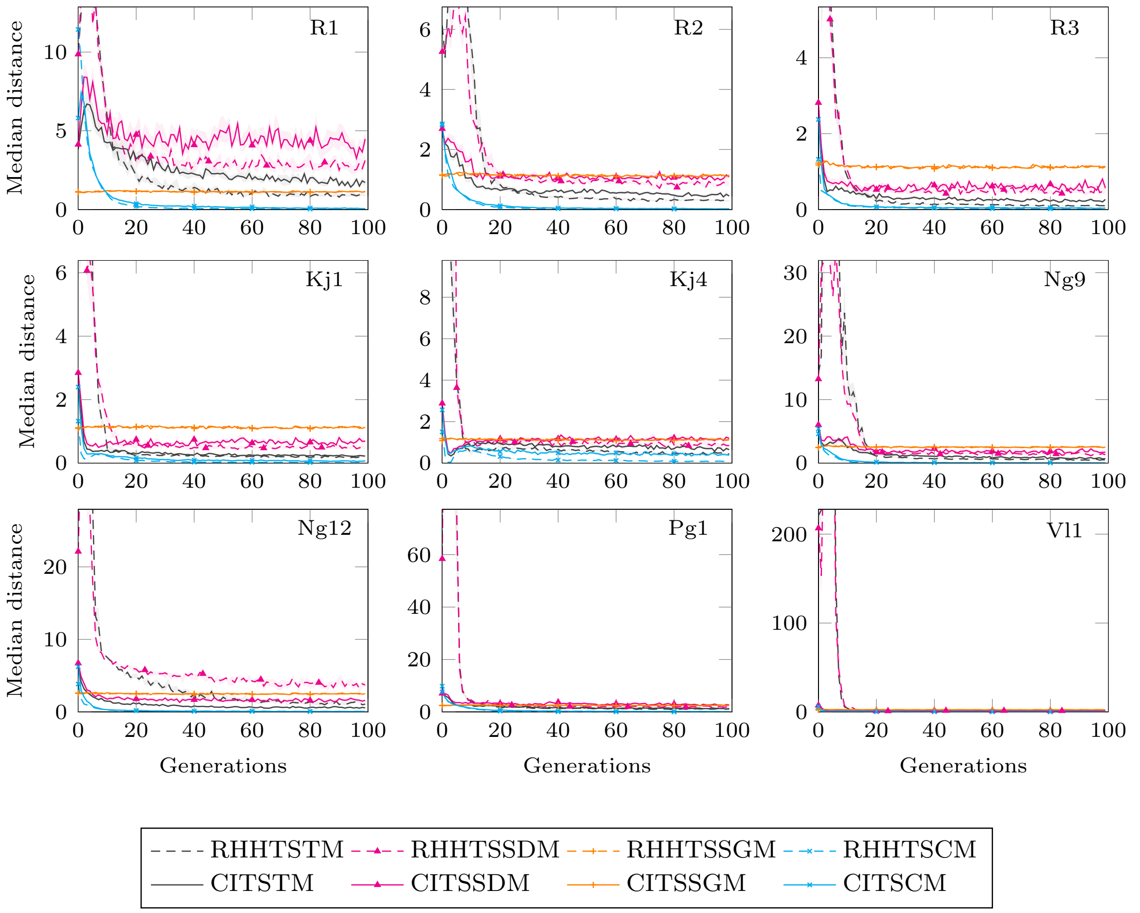


Figure 9.7: Median and 95% confidence interval of L_2 distance of offspring to parent over generations.

It is difficult to judge whether higher or lower distances are better from the perspective of the overall progress of evolution, and the optimal distance of an offspring to a parent is dependent on a distance of the parent to the target. Moreover we do not explicitly set radius of mutation for any operator. Thus we assume that an operator that keeps distance of the offspring to the parent below a certain threshold better matches principles of geometric mutation (cf. Definition 5.6).

SGM is the only operator that maintains roughly constant distance from the offspring to the parent during evolutionary run. In Boolean domain median distance is clearly dependent on number of fitness cases and approximately amounts to $2^{|F|-2}$, while in symbolic regression it is about 1.1 and 2.5 for 20 and 100 fitness cases, respectively. This indicates that SGM is blind to a state of population and does not adapt its search strategy between exploration and exploitation. In contrast median distance between an offspring and a parent for remaining operators decreases over time and the drop is steeper in early generations. In other words the remaining operators seem to gradually (and implicitly) adjust their strategies from exploration to exploitation over time.

The lowest statistics are achieved by CiTsCm for all Boolean problems except Par7 and CiTsTm for all Boolean problems except Maj7. In contrast in 7 out of 9 symbolic regression problems the lowest values belong to RhhTsCm and in two problems for CiTsCm. The highest values in both domains belong to either SGM or SDM, depending on a problem. By juxtaposing this observation with high effectiveness of SDM and SDM, it becomes clear that both operators, although effective, are unbiased and their search strategies are virtually random.

To statistically compare differences between operators we conduct two Friedman's tests, separately for each domain. The tests result in conclusive p-values of 4.48×10^{-5} and 1.83×10^{-5}, respectively for Boolean and symbolic regression domains. Corresponding post-hoc analyses are presented in Tables 9.7 and 9.9. In Boolean domain indeed both SGM setups produce offspring in much higher distance to parent than all other operators except SDM. In turn TM and CM with CI carry out shorter steps than RhhTsSdm. In symbolic regression both SGM and both SDM setups produce more distant offspring to parent than both CM setups.

9.3.4 Program size

Average and 95% confidence interval of number of nodes in the best of generation program is shown in Figure 9.8 and in the best of run program in Table 9.35.

First, we observe that the size increases linearly in number of generations for both CM setups in symbolic regression, and logarithmically in the Boolean domain, finally resulting in bigger programs than of the remaining operators. Second, although RHH initializes smaller programs than CI, in 6 out of 9 symbolic regression problems CM accompanied with RHH finally produces bigger programs than with CI at the end of run. In contrast the same occurs only for 2 out of 9 Boolean problems.

The sizes of programs produced by TM, SDM and SGM for most of the problems are roughly the same and can be divided into two groups depending on the way of initialization: setups that employ RHH typically produce smaller programs than the corresponding setups with CI, however the gap between corresponding setups decreases over time. The smallest programs are produced by RhhTsSgm in 12 out of 18 problems.

To statistically verify differences between setups we conduct Friedman's test that results in conclusive p-value of 2.41×10^{-12}. Post-hoc analysis is presented in Table 9.36 and confirms that all setups except CiTsSdm produce smaller programs than both CM setups. The analysis also reveals that RhhTsSgm produces smaller programs than CiTsTm and CiTsSdm.

9.3.5 Computational costs

Figure 9.9 presents average and 95% confidence interval of the total CPU time consumed by each setup over generations and Table 9.37 presents the same numbers at the end of run.

First, initialization times have major impact on total execution times. In symbolic regression, the setups that use CI require more time to initialize than the corresponding setups with RHH require for entire run. An exception is CM in problems with 20 fitness cases, where RhhTsCm reaches initialization time of CiTsCm after approximately 80 generations. In Boolean domain CI time is comparable with times consumed by about $20-40$ generations of corresponding setups with RHH. For all problems, at any stage of evolution, each setup with RHH requires no more time than the corresponding setup with CI.

Within a group of RHH initialization clearly the lowest time is consumed by SGM for 15 out of 18 problems, three times CM is quicker. We attribute this phenomenon to the fact that all runs of CM quickly find optimums for these three problems, hence the evolution terminates earlier there. For CI initialization, again SGM is the quickest in 11 out of 18 problems, in six problems TM and once CM are the quickest. Note that differences in time between SGM, TM and SDM are rather low, however between CM and other operators are moderate.

To statistically assess these differences we carry out Friedman's test on total times. The test results in conclusive p-value of 7.55×10^{-15}, thus we conduct post-hoc analysis in Table 9.38. Each setup with RHH is statistically faster than at least one setup with CI. RhhTsSgm is the fastest setup that outperforms 5 out of 7 other setups. Runner up is RhhTsTm that outperforms three other setups. The setup outperformed the most number of times (five) is CiTsCm, hence it can be considered the slowest one.

To verify whether computational costs of *a particular setup* correlate with a size of produced programs, we calculate Spearman's rank correlation coefficient of these two variables over problems for each setup separately and present the values together with p-values of t test for significance of correlation in left part of Table 9.39. Only for three setups, RhhTsTm, RhhTsSdm and

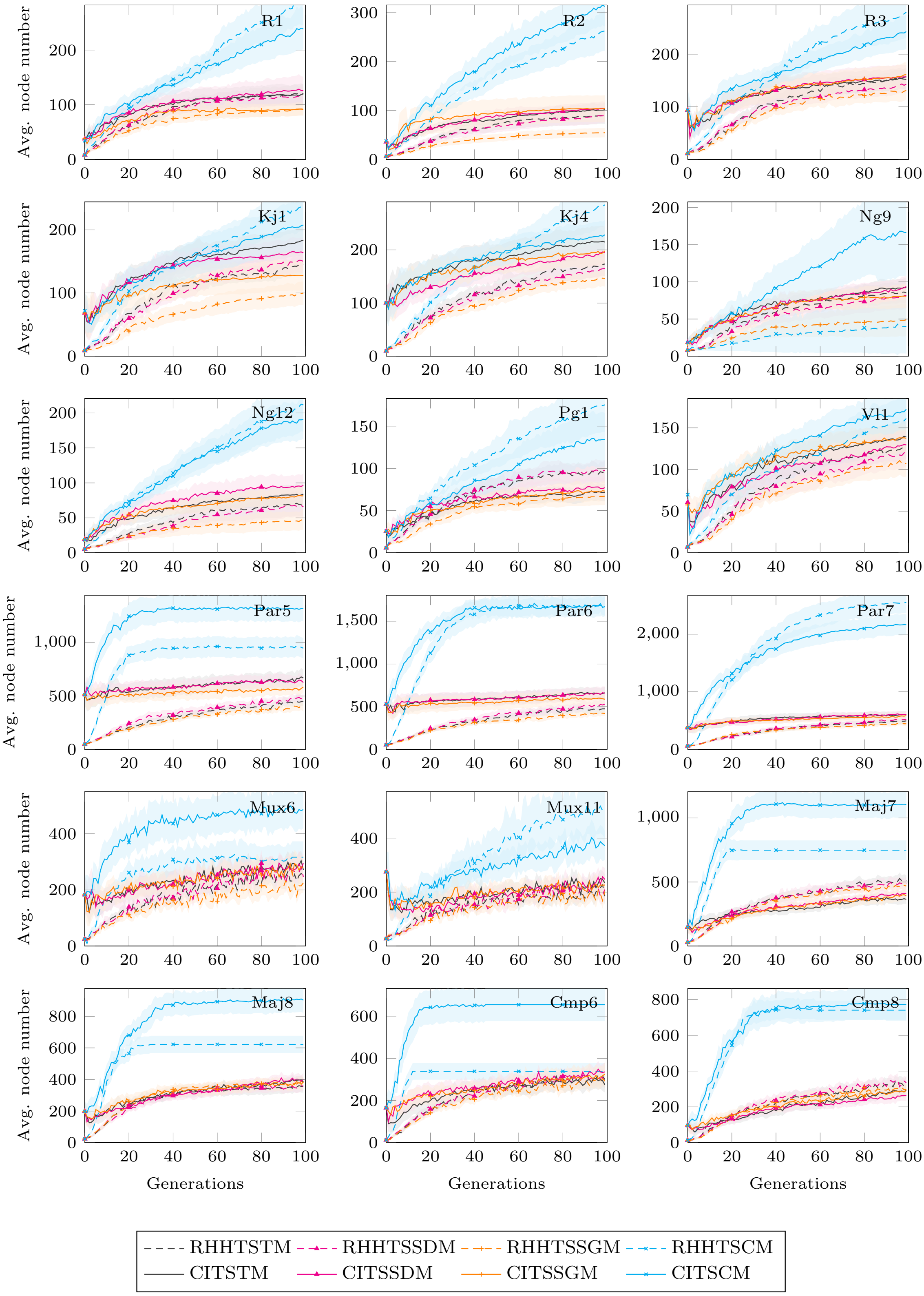


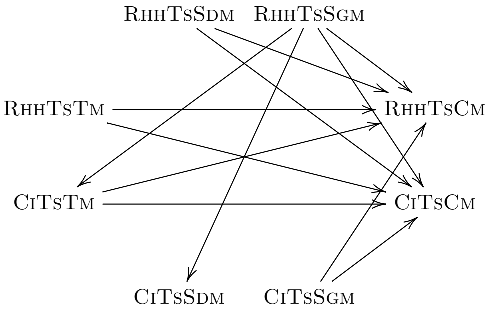
Figure 9.8: Average and 95% confidence interval of number of nodes in the best of generation program.

Table 9.35: Average and 95% confidence interval of number of nodes in the best of run program (the lowest in bold).

Prob.	RhhTsTm	RhhTsSdm	RhhTsSgm	RhhTsCm	CiTsTm	CiTsSdm	CiTsSgm	CiTsCm
R1	118.90 ±16.66	116.87 ±15.06	92.33 ±11.26	294.43 ±45.25	119.77 ±18.79	127.63 ±26.94	**91.53** ±11.91	239.10 ±35.25
R2	89.30 ±16.48	92.73 ±16.19	**54.50** ±9.49	261.63 ±40.09	103.90 ±14.43	105.53 ±13.90	103.37 ±27.22	313.90 ±41.04
R3	156.57 ±16.76	142.67 ±16.09	**130.53** ±17.43	278.57 ±47.39	152.23 ±19.31	158.53 ±22.41	160.90 ±23.83	246.40 ±33.75
Kj1	143.13 ±14.19	151.50 ±17.43	**98.67** ±18.28	236.63 ±38.14	183.93 ±23.05	163.23 ±23.69	128.40 ±23.79	208.10 ±40.65
Kj4	171.57 ±20.92	161.53 ±12.91	**148.13** ±16.40	284.73 ±39.15	215.00 ±29.31	196.93 ±20.18	197.90 ±25.43	228.57 ±27.36
Ng9	86.00 ±13.40	81.00 ±17.05	48.17 ±21.20	**43.37** ±39.63	92.40 ±12.55	95.33 ±14.65	80.03 ±15.34	169.23 ±59.87
Ng12	70.67 ±17.52	67.63 ±17.03	**47.17** ±14.32	212.10 ±38.54	85.33 ±15.40	95.17 ±16.95	82.13 ±13.60	192.53 ±27.24
Pg1	93.93 ±12.81	98.77 ±12.53	**66.90** ±10.43	179.07 ±33.52	71.13 ±10.19	76.73 ±12.56	73.87 ±11.69	135.87 ±32.34
Vl1	126.27 ±15.10	121.67 ±18.62	**112.23** ±19.55	160.20 ±27.84	137.57 ±18.43	128.43 ±16.78	138.63 ±17.43	172.10 ±21.88
Par5	435.00 ±40.62	461.73 ±37.03	**382.33** ±36.11	962.40 ±96.16	646.27 ±73.23	648.73 ±59.22	566.57 ±46.82	1317.73 ±120.58
Par6	471.00 ±42.08	527.33 ±58.17	**425.73** ±37.99	1701.27 ±111.13	657.13 ±69.11	644.27 ±76.62	603.77 ±81.35	1675.93 ±106.48
Par7	488.60 ±46.56	518.40 ±36.86	**448.17** ±37.21	2562.20 ±185.87	605.07 ±63.38	605.73 ±55.41	588.30 ±60.59	2162.20 ±178.97
Mux6	256.13 ±45.87	259.20 ±40.62	**206.70** ±28.06	316.60 ±56.34	284.47 ±40.62	294.33 ±42.45	275.03 ±42.16	488.20 ±66.06
Mux11	**179.27** ±33.51	206.87 ±35.87	191.53 ±36.31	518.00 ±107.85	236.73 ±37.42	239.60 ±30.73	202.43 ±33.37	378.00 ±63.56
Maj7	521.73 ±45.20	490.20 ±43.57	479.03 ±32.93	748.73 ±76.47	**373.33** ±37.64	405.47 ±26.39	404.60 ±48.18	1101.73 ±101.66
Maj8	363.93 ±42.14	**358.13** ±32.71	371.03 ±32.87	622.40 ±54.79	390.20 ±26.59	413.47 ±38.86	380.43 ±36.31	910.80 ±76.69
Cmp6	314.60 ±36.93	298.87 ±32.14	**288.70** ±32.01	338.13 ±39.60	293.07 ±41.72	335.93 ±39.89	311.87 ±52.67	654.27 ±77.71
Cmp8	339.73 ±37.55	340.93 ±36.85	296.90 ±31.99	740.33 ±53.44	288.87 ±35.06	**274.27** ±33.92	280.33 ±31.29	771.67 ±90.80
Rank:	3.28	3.28	1.61	7.17	4.44	5.11	3.61	7.50

Table 9.36: Post-hoc analysis of Friedman's test conducted on Table 9.35: p-values of incorrectly judging a setup in a row as producing smaller programs than a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RhhTsTm	RhhTsSdm	RhhTsSgm	RhhTsCm	CiTsTm	CiTsSdm	CiTsSgm	CiTsCm
RhhTsTm				**0.000**	0.844	0.325	1.000	**0.000**
RhhTsSdm	1.000			**0.000**	0.844	0.325	1.000	**0.000**
RhhTsSgm	0.454	0.454		**0.000**	**0.012**	**0.000**	0.218	**0.000**
RhhTsCm								1.000
CiTsTm				**0.020**		0.992		**0.005**
CiTsSdm				0.188				0.068
CiTsSgm				**0.000**	0.972	0.595		**0.000**
CiTsCm								



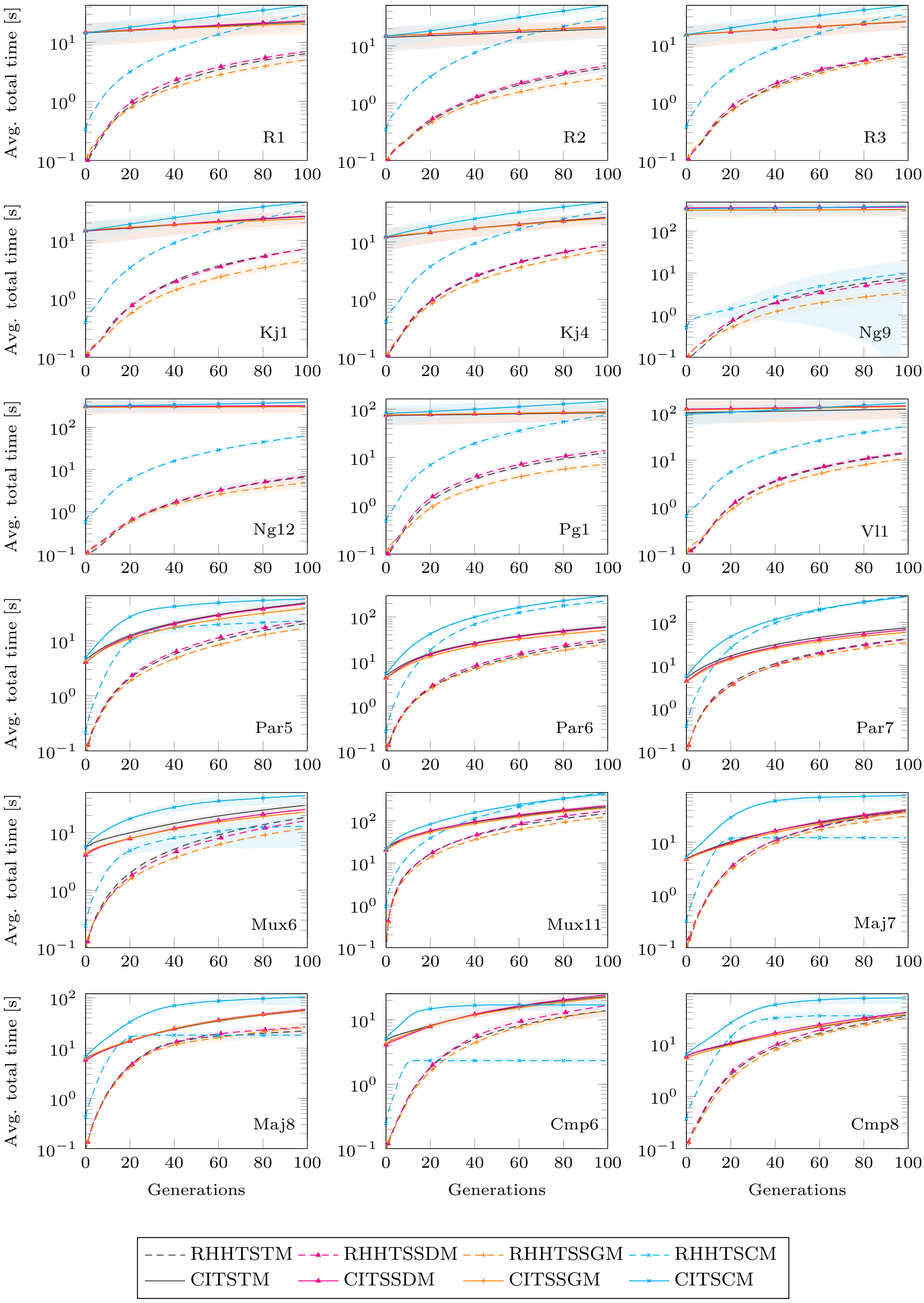


Figure 9.9: Average and 95% confidence interval of total execution time w.r.t. number of generations.

Table 9.37: Average and 95% confidence interval of total execution time spent to finish run (the lowest in bold).

Prob.	RhhTsTm	RhhTsSdm	RhhTsSgm	RhhTsCm	CiTsTm	CiTsSdm	CiTsSgm	CiTsCm
R1	6.52 ±0.76	7.07 ±0.82	**5.07** ±0.57	30.40 ±2.78	22.29 ±6.25	23.39 ±6.40	20.85 ±6.69	42.60 ±7.64
R2	4.15 ±0.68	4.56 ±0.70	**2.75** ±0.39	30.45 ±2.82	19.80 ±6.27	21.51 ±6.18	21.11 ±6.53	50.92 ±7.75
R3	6.92 ±0.74	7.18 ±0.73	**6.28** ±0.68	33.53 ±3.92	24.75 ±6.57	25.15 ±6.45	25.16 ±6.40	47.51 ±7.75
Kj1	7.21 ±0.65	7.23 ±0.70	**4.54** ±0.79	32.85 ±3.53	25.72 ±6.22	26.14 ±6.39	23.68 ±6.89	45.24 ±6.23
Kj4	8.95 ±0.81	9.04 ±0.77	**7.35** ±0.82	34.76 ±3.83	26.93 ±5.69	26.20 ±5.22	25.66 ±5.73	49.24 ±6.99
Ng9	7.92 ±1.36	6.75 ±1.30	**3.46** ±1.16	9.96 ±10.04	325.83 ±98.50	371.34 ±115.82	325.06 ±101.13	394.26 ±111.60
Ng12	6.74 ±1.61	7.07 ±1.48	**4.84** ±1.20	62.42 ±7.43	318.94 ±93.35	325.84 ±96.73	307.53 ±94.68	393.74 ±95.28
Pg1	12.56 ±1.33	14.04 ±1.51	**7.37** ±1.07	75.35 ±11.31	84.66 ±28.05	88.14 ±28.65	87.35 ±28.52	145.16 ±36.25
Vl1	14.20 ±1.62	14.99 ±2.04	**11.00** ±1.44	52.56 ±5.96	123.14 ±45.57	143.64 ±60.79	139.22 ±60.43	165.14 ±42.54
Par5	20.78 ±1.30	23.22 ±1.73	**17.12** ±1.23	23.16 ±8.73	48.81 ±6.57	47.21 ±4.88	38.81 ±4.28	56.77 ±13.25
Par6	28.62 ±2.55	32.01 ±2.60	**24.51** ±1.95	227.34 ±30.34	61.56 ±6.92	59.09 ±7.68	50.59 ±6.55	300.63 ±20.99
Par7	41.80 ±3.73	40.92 ±2.74	**34.04** ±2.88	421.76 ±37.16	74.88 ±8.56	67.35 ±6.51	59.11 ±6.19	398.49 ±35.84
Mux6	18.64 ±1.56	16.19 ±1.47	**12.08** ±0.97	12.82 ±7.34	30.25 ±4.01	25.60 ±3.14	23.08 ±2.70	44.29 ±12.45
Mux11	150.25 ±14.00	171.88 ±16.25	**124.15** ±9.23	469.56 ±86.15	215.21 ±24.95	230.23 ±29.99	203.78 ±29.97	434.58 ±50.90
Maj7	37.82 ±3.93	38.65 ±3.72	31.49 ±3.43	**12.15** ±1.42	39.78 ±3.52	41.94 ±3.01	36.99 ±3.23	76.64 ±15.64
Maj8	21.99 ±5.01	26.70 ±6.40	26.19 ±7.81	**18.15** ±2.46	56.38 ±4.45	59.17 ±6.34	57.12 ±5.77	104.68 ±34.70
Cmp6	13.94 ±2.66	16.67 ±2.77	13.83 ±2.06	**2.33** ±0.23	23.16 ±3.66	24.47 ±4.03	22.35 ±3.22	17.03 ±3.21
Cmp8	34.10 ±3.86	40.62 ±3.65	**31.20** ±3.51	34.41 ±7.42	36.54 ±3.83	39.78 ±3.65	36.95 ±3.69	75.14 ±17.06
Rank:	2.39	3.39	1.22	4.72	5.50	6.22	4.83	7.72

Table 9.38: Post-hoc analysis of Friedman's test conducted on Table 9.37: p-values of incorrectly judging a setup in a row as faster than a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RhhTsTm	RhhTsSdm	RhhTsSgm	RhhTsCm	CiTsTm	CiTsSdm	CiTsSgm	CiTsCm
RhhTsTm		0.925		0.081	**0.003**	**0.000**	0.056	**0.000**
RhhTsSdm				0.730	0.161	**0.012**	0.641	**0.000**
RhhTsSgm	0.844	0.137		**0.001**	**0.000**	**0.000**	**0.000**	**0.000**
RhhTsCm					0.981	0.595	1.000	**0.006**
CiTsTm						0.987		0.116
CiTsSdm								0.595
CiTsSgm					0.992	0.687		**0.010**
CiTsCm								

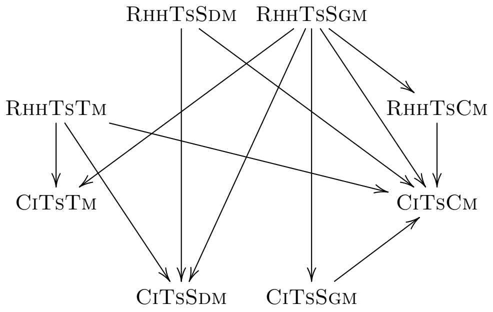


Table 9.39: Spearman's rank correlation coefficient of number of nodes in produced programs and computational time consumed by a setup. Left: correlation over problems for each setup separately. Right: rankings of setups from last rows of Tables 9.35 and 9.37, and correlation of them. P-values of t test of significance of correlation, significant correlations ($\alpha = 0.05$) are marked in bold.

	Correlation	P-value	Size rank	Time rank
RhhTsTm	**0.810**	0.000	3.28	2.39
RhhTsSdm	**0.847**	0.000	3.28	3.39
RhhTsSgm	**0.853**	0.000	1.61	1.22
RhhTsCm	0.065	0.399	7.17	4.72
CiTsTm	−0.082	0.374	4.44	5.50
CiTsSdm	−0.106	0.337	5.11	6.22
CiTsSgm	−0.088	0.365	3.61	4.83
CiTsCm	0.020	0.469	7.50	7.72
		Correlation:	**0.819**	
		P-value:	0.006	

Table 9.40: Exemplary analysis for time budget of 15 seconds for Kj4 problem. The highest generation and the lowest statistics are in bold.

	Generation	Fitness	Test-set fitness	Program size
RhhTsTm	**100**	0.19 ±0.03	0.62 ≤ 1.38 ≤ 2.49	171.57 ±20.92
RhhTsSdm	**100**	0.19 ±0.04	1.02 ≤ 1.78 ≤ 2.94	167.70 ±18.11
RhhTsSgm	**100**	0.17 ±0.03	0.95 ≤ 1.50 ≤ 2.48	148.13 ±16.40
RhhTsCm	58	**0.05** ±0.02	0.31 ≤ **0.57** ≤ 1.60	203.03 ±28.31
CiTsTm	23	0.31 ±0.05	2.18 ≤ 2.88 ≤ 5.82	163.13 ±29.26
CiTsSdm	22	0.39 ±0.05	2.05 ≤ 2.35 ≤ 3.22	**133.13** ±19.81
CiTsSgm	22	0.37 ±0.05	2.19 ≤ 2.45 ≤ 3.30	155.23 ±24.38
CiTsCm	9	0.43 ±0.05	2.06 ≤ 2.23 ≤ 3.01	138.53 ±28.46

RhhTsSgm, we observe strong positive correlation (> 81%). Therefore by enforcing limits on program size for these setups the computational costs would decrease. For the remaining operators there are clearly other factors than program size that influence computational costs.

Right part of Table 9.39 consists of rankings of *all setups* on the final number of nodes and total computational time took from Tables 9.35 and 9.37. The correlation between rankings is strong, hence in general a setup that leads to bigger programs would also be slower than then a setup that leads to small programs.

In practical applications we often impose a time budget that when exceeded causes the algorithm to terminate prematurely. Given a high enough budget, all of the considered setups can terminate normally, however for very strict budget we may obtain substantially different final results than presented in the previous sections. To simulate this case we encourage the reader to mark in a plot in Figure 9.9 the assumed time budget using horizontal line, for each trial read the number of generation where it crosses the line[3] and compare interested statistics, e.g., training-set and test-set fitness or program size achieved up to these generations.

For instance we assume time budget of 15 seconds for Kj4 problem. Average generation and corresponding statistics are presented in Table 9.40. The analysis shows that although RhhTsTm, RhhTsSdm, RhhTsSgm are ex aequo the fastest setups, the best training-set and test-set fitness are achieved by RhhTsCm in the same time and in only 58 generations. In addition the smallest programs are produced by CiTsSdm, however it reaches only 22 generations, hence it may not converged yet and its results are far from the final.

[3] Use 1 or 100 if the trial is above or below the line, respectively.

This example shows that although CM is the slowest operator, CM can achieve better results than faster operators within the same time budget in less number of generations.

9.4 Crossover

This section is aimed to answer research question 4 on characteristics of crossover operators. Similarly to the question regarding mutation, we execute separate runs with CX, TX, SDX and SGX to compare them in terms of training-set and test-set fitness, degree of effectiveness and geometry, size of produced programs and computational costs. We double each setup to measure impact of use of CI and CTS as replacements for canonical RHH and TS, respectively. To sum up there are eight setups: RhhTsTx, RhhTsSdx, RhhTsSgx, RhhTsCx, CiCtsTx, CiCtsSdx, CiCtsSgx, CiCtsCx.

9.4.1 Fitness-based performance

We present average and 95% confidence interval of the best of generation fitness achieved by the considered setups in Figure 9.10 and the best of run fitness in Table 9.41.

SGX clearly suffers from premature convergence in symbolic regression, in few early generations its fitness rapidly falls down, then it sticks to a value far from the optimum for the rest of run. In contrast in Boolean domain SGX converges to the optimum in less number of generations than any other operator. One exception is Mux11 problem that remains unsolved by all setups and for which TX and SDX achieve a bit better fitness than SGX.

CX gradually improves the fitness, where the curve of improvement is steeper in early generations than in the later ones. For symbolic regression problems CX achieves the best fitness of all operators for almost entire runs, while in Boolean problems it is outperformed by SGX and becomes a runner up in 8 out of 9 problems. In Mux11 CX takes position in the tail of the ranking.

The company of CI and CTS seems to not have an impact on performance of CX in regression domain, since both trials of CX overlap in 6 out of 9 problems and once CiCtsCx is better than RhhTsCx. In Boolean domain CiCtsCx achieves better fitness than RhhTsCx in 6 out of 9 problems, once their results are exact the same, and two times RhhTsCx is slightly better. The pair of CI and CTS also helps SGX in 4 out of 9 symbolic regression problems and 3 out of 9 Boolean problems. For remaining regression problems RhhTsSgx is better, while for Boolean ones there is no noticeably difference between both SGX setups. Thus it is difficult to draw clear conclusions on impact of CI and CTS on SGX. The duet of CI and CTS helps also blind to geometry TX and SDX, respectively in 4 and 5 out of 9 symbolic regression and in 5 out of 9 Boolean problems, however evidence is too weak to draw meaningful conclusions.

Performance of CX is quite stable over problems and domains. In 7 out of 18 problems CiCtsCx and in 6 problems RhhTsCx are characterized by the narrowest confidence intervals (thus variance too) of the final average fitness. Hence CX can be considered very stable and reliable in terms of the expected results.

To statistically assess differences between the operators we conduct Friedman's test on the average of the best of run fitness. The resulting p-value of 1.26×10^{-9} is conclusive at the significance level $\alpha = 0.05$, hence we carry out post-hoc analysis in Table 9.42. The analysis reveals outranking of CX over TX irrelevantly to initialization and selection methods, and of CiCtsCx over both SGX setups.

In Table 9.43 we present empirical probability, i.e., success rate, and 95% confidence interval of solving the benchmark problems optimally, i.e., by finding program $p^* : s(p^*) = t$.[4]

Eight out of nine symbolic regression problems remain unsolved for each run of each setup. In contrary only Mux11 is unsolved by all operators in Boolean domain. Both SGX setups solve

[4] We use similarity threshold, see Table 8.2.

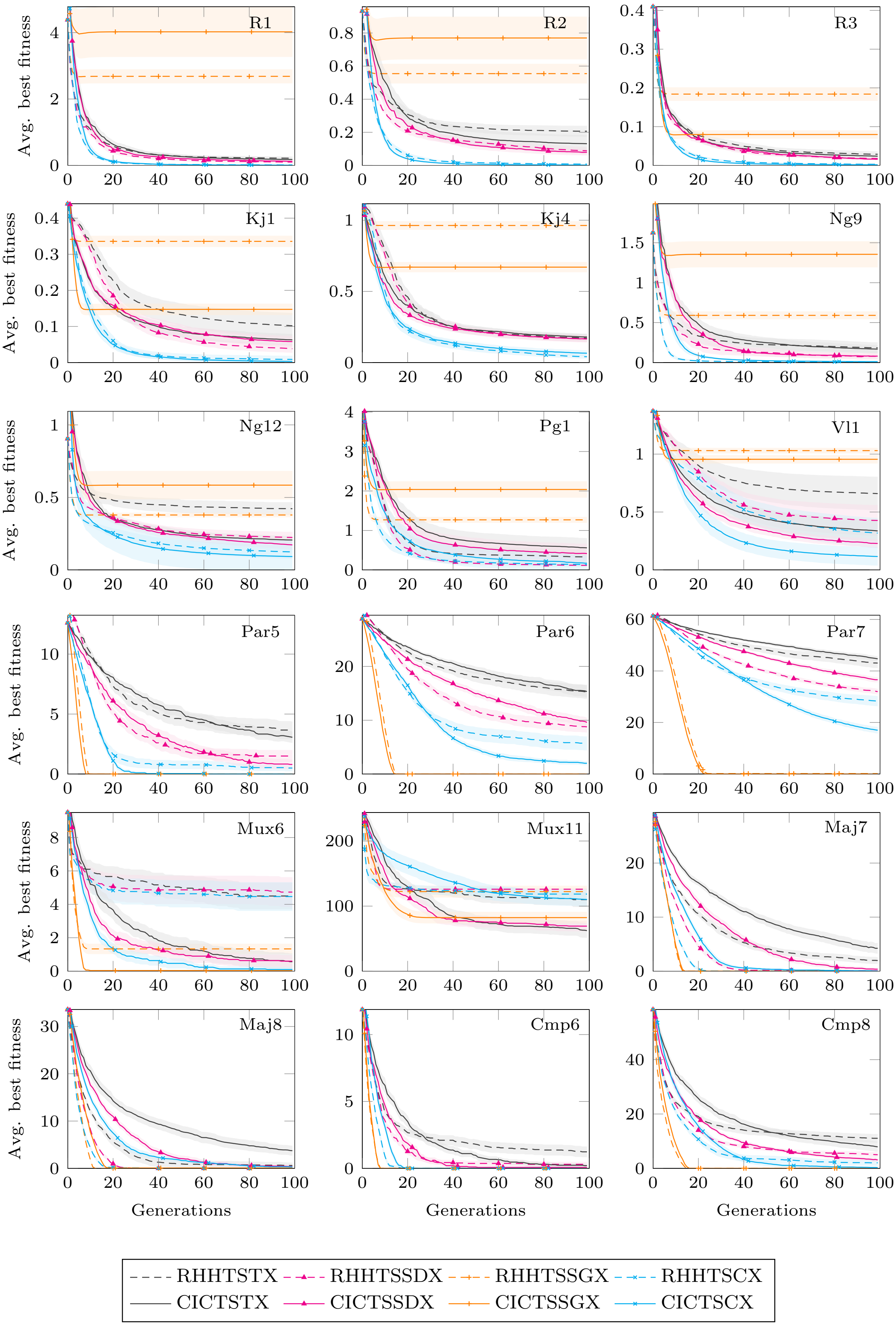


Figure 9.10: Average and 95% confidence interval of the best of generation fitness.

Table 9.41: Average and 95% confidence interval of the best of run fitness (the best in bold).

Prob.	RhhTsTx	RhhTsSdx	RhhTsSgx	RhhTsCx	CiCtsTx	CiCtsSdx	CiCtsSgx	CiCtsCx
R1	0.22 ±0.05	0.10 ±0.04	2.68 ±0.21	0.01 ±0.01	0.18 ±0.05	0.12 ±0.04	4.02 ±0.76	**0.01** ±0.00
R2	0.21 ±0.03	0.09 ±0.02	0.55 ±0.06	0.01 ±0.00	0.13 ±0.05	0.08 ±0.02	0.77 ±0.13	**0.00** ±0.00
R3	0.03 ±0.01	0.02 ±0.00	0.18 ±0.02	0.00 ±0.00	0.02 ±0.01	0.02 ±0.00	0.08 ±0.01	**0.00** ±0.00
Kj1	0.10 ±0.04	0.04 ±0.01	0.34 ±0.02	0.01 ±0.01	0.06 ±0.02	0.06 ±0.02	0.15 ±0.02	**0.00** ±0.00
Kj4	0.17 ±0.03	0.17 ±0.03	0.96 ±0.03	**0.04** ±0.01	0.18 ±0.02	0.16 ±0.02	0.67 ±0.04	0.06 ±0.02
Ng9	0.18 ±0.07	0.08 ±0.04	0.59 ±0.05	**0.01** ±0.01	0.17 ±0.05	0.08 ±0.03	1.36 ±0.16	0.01 ±0.00
Ng12	0.42 ±0.04	0.22 ±0.04	0.38 ±0.02	0.12 ±0.05	0.21 ±0.03	0.17 ±0.03	0.58 ±0.10	**0.09** ±0.12
Pg1	0.33 ±0.19	**0.12** ±0.04	1.27 ±0.07	0.14 ±0.05	0.56 ±0.24	0.41 ±0.25	2.04 ±0.21	0.17 ±0.20
Vl1	0.66 ±0.15	0.43 ±0.13	1.03 ±0.02	0.32 ±0.13	0.34 ±0.10	0.23 ±0.04	0.96 ±0.04	**0.12** ±0.08
Par5	3.60 ±0.75	1.50 ±0.50	**0.00** ±0.00	0.47 ±0.27	3.07 ±0.47	0.77 ±0.41	**0.00** ±0.00	**0.00** ±0.00
Par6	15.23 ±1.33	8.73 ±1.05	**0.00** ±0.00	5.67 ±1.30	15.27 ±1.11	9.53 ±0.87	**0.00** ±0.00	1.97 ±0.34
Par7	43.00 ±1.95	31.80 ±1.09	0.17 ±0.16	28.23 ±1.99	44.63 ±0.98	36.40 ±1.02	**0.00** ±0.00	16.90 ±1.26
Mux6	4.50 ±0.80	4.73 ±0.90	1.33 ±0.31	4.47 ±0.86	0.57 ±0.49	0.60 ±0.36	**0.03** ±0.06	0.10 ±0.14
Mux11	109.93 ±11.35	125.93 ±2.06	122.27 ±9.09	118.47 ±7.20	**62.33** ±11.30	69.20 ±8.07	82.27 ±8.93	109.87 ±11.10
Maj7	1.93 ±0.61	0.07 ±0.09	**0.00** ±0.00	**0.00** ±0.00	4.23 ±1.00	0.37 ±0.27	**0.00** ±0.00	0.03 ±0.06
Maj8	0.60 ±0.63	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	3.73 ±1.07	0.33 ±0.31	0.07 ±0.09	0.30 ±0.21
Cmp6	1.23 ±0.40	0.30 ±0.19	**0.00** ±0.00	**0.00** ±0.00	0.13 ±0.12	0.07 ±0.09	**0.00** ±0.00	**0.00** ±0.00
Cmp8	11.03 ±1.60	5.00 ±0.91	**0.00** ±0.00	2.00 ±1.06	7.87 ±1.85	3.10 ±0.74	**0.00** ±0.00	0.27 ±0.33
Rank:	6.44	4.67	5.08	2.86	5.61	4.22	4.81	2.31

Table 9.42: Post-hoc analysis of Friedman's test conducted on Table 9.41: p-values of incorrectly judging a setup in a row as outranking a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RhhTsTx	RhhTsSdx	RhhTsSgx	RhhTsCx	CiCtsTx	CiCtsSdx	CiCtsSgx	CiCtsCx
RhhTsTx								
RhhTsSdx	0.355		1.000		0.941		1.000	
RhhTsSgx	0.700				0.998			
RhhTsCx	**0.000**	0.334	0.110		**0.016**	0.700	0.241	
CiCtsTx	0.970							
CiCtsSdx	0.110	0.999	0.964		0.678		0.996	
CiCtsSgx	0.466		1.000		0.975			
CiCtsCx	**0.000**	0.070	**0.014**	0.997	**0.001**	0.258	**0.042**	

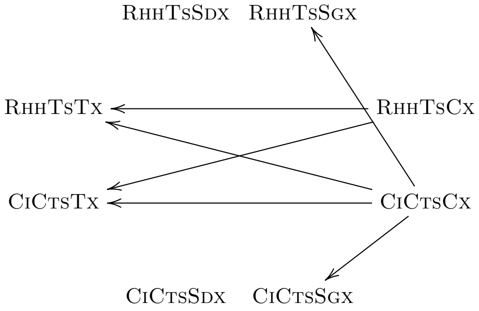
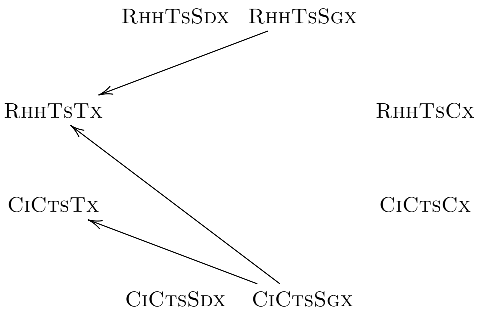


Table 9.43: Empirical probability and 95% confidence interval of finding the optimal program (the highest in bold).

Problem	RhhTsTx	RhhTsSdx	RhhTsSgx	RhhTsCx	CiCtsTx	CiCtsSdx	CiCtsSgx	CiCtsCx
R1	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
R2	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
R3	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
Kj1	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
Kj4	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
Ng9	0.23 ±0.15	0.13 ±0.12	0.00 ±0.00	**0.50** ±0.18	0.13 ±0.12	0.13 ±0.12	0.00 ±0.00	0.10 ±0.11
Ng12	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
Pg1	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
Vl1	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
Par5	0.03 ±0.06	0.33 ±0.17	**1.00** ±0.00	0.67 ±0.17	0.00 ±0.00	0.53 ±0.18	**1.00** ±0.00	**1.00** ±0.00
Par6	0.00 ±0.00	0.00 ±0.00	**1.00** ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	**1.00** ±0.00	0.00 ±0.00
Par7	0.00 ±0.00	0.00 ±0.00	0.87 ±0.12	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	**1.00** ±0.00	0.00 ±0.00
Mux6	0.03 ±0.06	0.07 ±0.09	0.10 ±0.11	0.03 ±0.06	0.73 ±0.16	0.67 ±0.17	**0.97** ±0.06	0.93 ±0.09
Mux11	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
Maj7	0.27 ±0.16	0.93 ±0.09	**1.00** ±0.00	**1.00** ±0.00	0.07 ±0.09	0.77 ±0.15	**1.00** ±0.00	0.97 ±0.06
Maj8	0.80 ±0.14	**1.00** ±0.00	**1.00** ±0.00	**1.00** ±0.00	0.07 ±0.09	0.83 ±0.13	0.93 ±0.09	0.77 ±0.15
Cmp6	0.30 ±0.16	0.73 ±0.16	**1.00** ±0.00	**1.00** ±0.00	0.87 ±0.12	0.93 ±0.09	**1.00** ±0.00	**1.00** ±0.00
Cmp8	0.00 ±0.00	0.00 ±0.00	**1.00** ±0.00	0.47 ±0.18	0.00 ±0.00	0.13 ±0.12	**1.00** ±0.00	0.87 ±0.12
Rank:	5.33	4.92	3.69	4.14	5.31	4.75	3.53	4.33

Table 9.44: Post-hoc analysis of Friedman's test conducted on Table 9.43: p-values of incorrectly judging a setup in a row as outranking a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RhhTsTx	RhhTsSdx	RhhTsSgx	RhhTsCx	CiCtsTx	CiCtsSdx	CiCtsSgx	CiCtsCx
RhhTsTx								
RhhTsSdx	0.994				0.996			
RhhTsSgx	**0.047**	0.305		0.992	0.054	0.503		0.935
RhhTsCx	0.334	0.834			0.366	0.948		1.000
CiCtsTx	1.000							
CiCtsSdx	0.960	1.000			0.969			
CiCtsSgx	**0.017**	0.159	1.000	0.948	**0.021**	0.305		0.807
CiCtsCx	0.575	0.960			0.611	0.994		



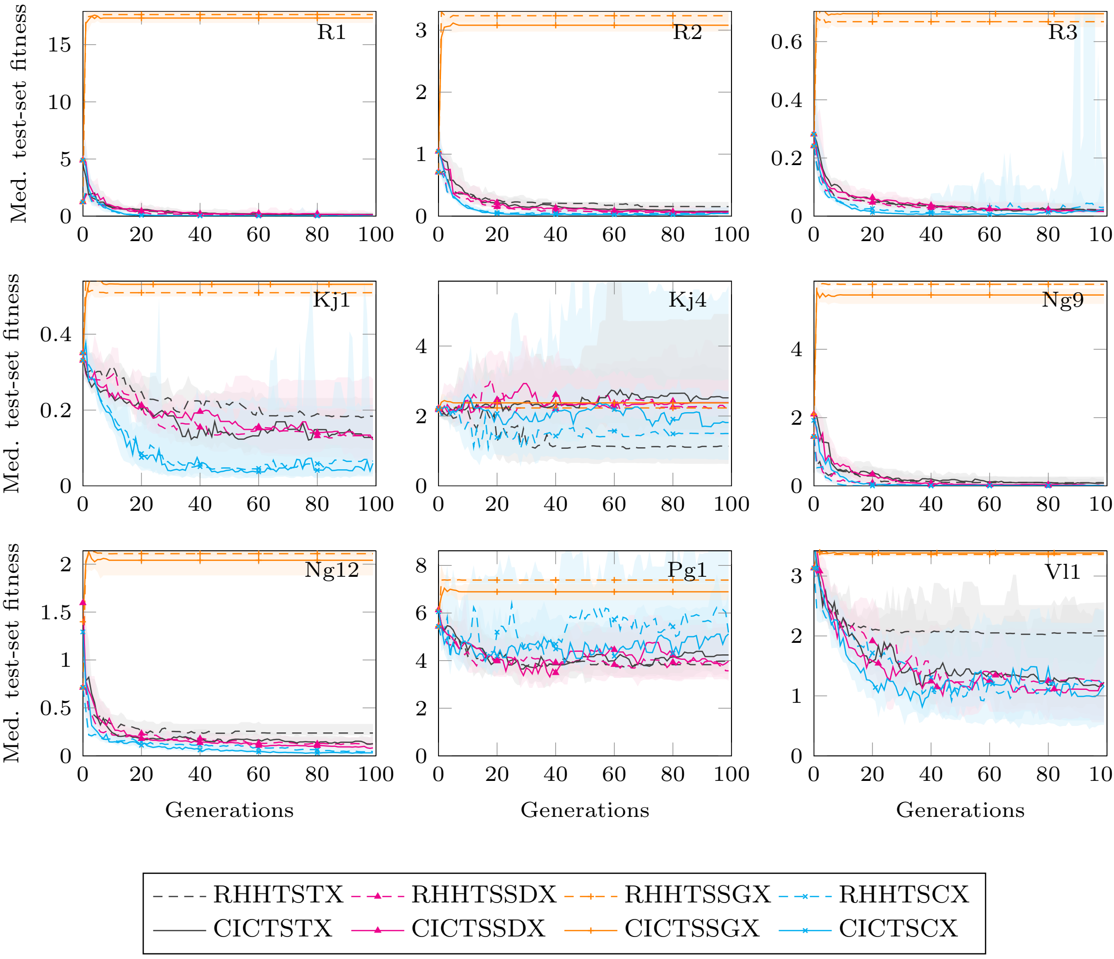


Figure 9.11: Median and 95% confidence interval of test-set fitness of the best of generation program on training-set.

in all runs 6 out of 9 Boolean problems. In remaining ones, except Mux11, they feature non-zero probability of success. The runner up is CX, whose both setups solve at least once 6 out of 9 Boolean problems and one regression problem. Canonical setup RhhTsTx seems to be the worst one, since it achieves the lowest probability for 14 out of 18 problems.

We carry out Friedman's test to verify whether there are significant differences in probability of solving problems between setups. The test results in conclusive p-value of 1.71×10^{-2}, thus we conduct post-hoc analysis presented in Table 9.44. The analysis shows outranking of both SGX setups over RhhTsTx and of CiCtsSgx over CiCtsTx. The main reason for that must be prominent performance of SGX on Boolean problems.

9.4.2 Generalization abilities

Figure 9.11 shows median and 95% confidence interval of test-set fitness of the best of generation program on training-set and Table 9.45 presents the same values for the best of run program. We switched to median to work around outlying observations.

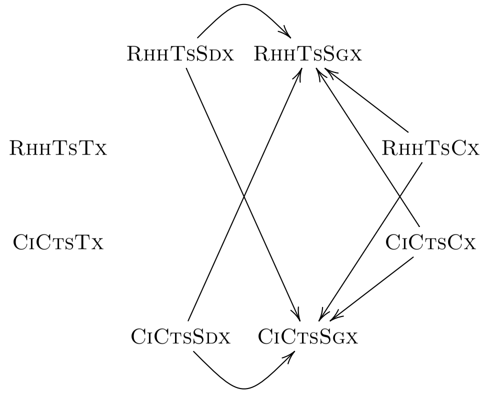
In all nine problems the worst test-set fitness is achieved by SGX. SGX is deteriorating its test-set fitness in the first ten generations while it improves the training-set one, then SGX converges on both sets. This clearly indicates major overfitting combined with premature convergence.

Table 9.45: Median and 95% confidence interval of test-set fitness of the best of run program on training-set (the best in bold).

Problem	RhhTsTx	RhhTsSdx	RhhTsSgx	RhhTsCx	CiCtsTx	CiCtsSdx	CiCtsSgx	CiCtsCx
R1	0.10 ≤ 0.15 ≤ 0.25	0.04 ≤ 0.06 ≤ 0.12	17.23 ≤ 17.64 ≤ 17.92	0.01 ≤ 0.04 ≤ 0.25	0.09 ≤ 0.15 ≤ 0.34	0.06 ≤ 0.12 ≤ 0.39	16.90 ≤ 17.33 ≤ 17.87	0.01 ≤ **0.02** ≤ 0.09
R2	0.10 ≤ 0.15 ≤ 0.23	0.03 ≤ 0.05 ≤ 0.09	3.14 ≤ 3.23 ≤ 3.30	0.04 ≤ 0.05 ≤ 0.14	0.05 ≤ 0.08 ≤ 0.14	0.05 ≤ 0.07 ≤ 0.19	2.98 ≤ 3.08 ≤ 3.26	0.01 ≤ **0.04** ≤ 0.15
R3	0.02 ≤ 0.02 ≤ 0.05	0.01 ≤ 0.02 ≤ 0.04	0.65 ≤ 0.67 ≤ 0.70	0.02 ≤ 0.04 ≤ 11.04	0.01 ≤ 0.02 ≤ 0.04	0.01 ≤ **0.02** ≤ 0.03	0.68 ≤ 0.70 ≤ 0.70	0.01 ≤ 0.02 ≤ 0.08
Kj1	0.12 ≤ 0.19 ≤ 0.23	0.07 ≤ 0.13 ≤ 0.28	0.50 ≤ 0.51 ≤ 0.54	0.03 ≤ 0.07 ≤ 0.21	0.09 ≤ 0.13 ≤ 0.20	0.06 ≤ 0.12 ≤ 0.21	0.53 ≤ 0.53 ≤ 0.54	0.03 ≤ **0.06** ≤ 0.20
Kj4	0.63 ≤ **1.14** ≤ 3.39	1.43 ≤ 2.23 ≤ 3.89	2.19 ≤ 2.23 ≤ 2.29	0.77 ≤ 1.62 ≤ 6.29	1.65 ≤ 2.52 ≤ 6.71	1.99 ≤ 2.30 ≤ 3.04	2.35 ≤ 2.38 ≤ 2.45	1.48 ≤ 1.81 ≤ 2.71
Ng9	0.01 ≤ 0.09 ≤ 0.27	0.00 ≤ 0.01 ≤ 0.06	5.85 ≤ 5.90 ≤ 6.00	0.00 ≤ **0.00** ≤ 0.01	0.03 ≤ 0.08 ≤ 0.20	0.01 ≤ 0.02 ≤ 0.11	5.33 ≤ 5.59 ≤ 5.77	0.01 ≤ 0.02 ≤ 0.03
Ng12	0.21 ≤ 0.24 ≤ 0.33	0.10 ≤ 0.13 ≤ 0.20	2.09 ≤ 2.11 ≤ 2.13	0.03 ≤ 0.04 ≤ 0.11	0.10 ≤ 0.13 ≤ 0.24	0.06 ≤ 0.08 ≤ 0.13	1.88 ≤ 2.04 ≤ 2.14	0.02 ≤ **0.03** ≤ 0.04
Pg1	3.44 ≤ 3.97 ≤ 5.36	3.21 ≤ **3.57** ≤ 4.34	7.27 ≤ 7.39 ≤ 7.44	4.00 ≤ 5.14 ≤ 8.59	3.75 ≤ 4.39 ≤ 5.41	3.24 ≤ 3.98 ≤ 5.36	6.54 ≤ 6.89 ≤ 7.21	3.65 ≤ 5.14 ≤ 6.78
Vl1	1.58 ≤ 2.08 ≤ 2.44	0.67 ≤ 1.28 ≤ 2.12	3.33 ≤ 3.36 ≤ 3.42	0.59 ≤ 1.22 ≤ 2.04	0.79 ≤ 1.23 ≤ 2.56	0.58 ≤ 1.21 ≤ 1.63	3.35 ≤ 3.38 ≤ 3.42	0.47 ≤ **0.98** ≤ 2.44
Rank:	4.67	3.67	7.22	2.89	4.78	3.33	7.33	2.11

Table 9.46: Post-hoc analysis of Friedman's test conducted on Table 9.45: p-values of incorrectly judging a setup in a row as outranking a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RhhTsTx	RhhTsSdx	RhhTsSgx	RhhTsCx	CiCtsTx	CiCtsSdx	CiCtsSgx	CiCtsCx
RhhTsTx			0.344		1.000		0.288	
RhhTsSdx	0.989		**0.043**		0.980		**0.032**	
RhhTsSgx							1.000	
RhhTsCx	0.786	0.998	**0.004**		0.729	1.000	**0.003**	
CiCtsTx			0.403				0.344	
CiCtsSdx	0.944	1.000	**0.018**		0.916		**0.012**	
CiCtsSgx								
CiCtsCx	0.344	0.881	**0.000**	0.998	0.288	0.965	**0.000**	



The situation is quite different for other operators that typically improve their test-set fitness over generations. In 6 out of 9 problems, the best test-set fitness is obtained by CX. We also do not notice any signs of overfitting of neither CX, nor TX or SDX. For the remaining problems, namely Kj4, Pg1 and Vl1, CX is being improving test-set fitness in early generations, then the fitness becomes oscillating with a slight bias towards deterioration. In all nine problems CiCtsSdx is better than CiCtsTx and in seven problems RhhTsSdx is better than RhhTsTx.

It is not clear how use of CI and CTS influences test-set fitness or overfitting, however we observe a slight bias toward better fitness in CiCts* setups, e.g., in 6 out of 9 problems for CX and TX, 5 out of 9 for SGX and 4 out of 9 for SDX.

To statistically assess our observations, we carry out Friedman's test on Table 9.45. The test is conclusive with p-value of 2.04×10^{-4} . Post-hoc analysis presented in Table 9.46 leaves no doubts that both CX and both SDX setups outrank both SGX setups.

9.4.3 Effectiveness and geometry

Figure 9.12 and Table 9.47 present empirical probability and 95% confidence interval of effective crossover applications (cf. Definition 4.7) over generations and in the entire 100-generation run, respectively.

The obtained probabilities are noticeably higher for symbolic regression than for the Boolean domain. This a natural consequence of virtually infinite and continuous semantic space in real domain, in contrast to discrete and finite space in Boolean domain.

In all regression problems and in 6 out of 9 Boolean problems the highest values that amount to 100% and $> 82.8\%$, respectively, are reported for CiCtsSgx. Second most effective setup is RhhTsSgx that achieves $> 99.8\%$ and $> 59.8\%$, respectively for the same problems. In two Mux problems we observe deviation from the norm, where SGX achieves much lower probabilities than in other problems. This phenomenon is especially visible for RhhTsSgx for which the probabilities are $< 5.5\%$. The plot shows that effectiveness of SGX there decreases from the initial of approximately 50% to almost zero in less than 20 generations.

In turn both SDX setups achieve slightly lower probabilities than SGX and higher than other operators.

In symbolic regression TX and CX feature nearly equal total probability of effective crossovers, however the value for both CX setups typically decreases in early generations, while for RhhTsTx it increases and for CiCtsTx is roughly constant. For both operators higher values are observed for CiCts* variant, probably due to effectiveness of CI and CTS. In Boolean domain both CX setups start from different probabilities of effective crossovers, then in first 20 generations the probabilities are equalizing. For TX there is no such a clear trend, however in 6 out of 9 Boolean problems RhhTsTx leads to higher probabilities than CiCtsTx.

The imperfect effectiveness of CX may be a cause of a technical implementation of $\mathsf{Invert}(\cdot,\cdot,\cdot)$ function from Table 7.1 that returns at most two values, even if there are more inversions of parents' semantics ($s(p_1)$ and $s(p_2)$ in Algorithm 7.5) available. Thus, forbidden semantics ($D^1_\varnothing$ and $D^2_\varnothing$ in that algorithm) may not contain all values that prevent from producing neutral offspring.

To statistically verify ordering of the operators we conduct Friedman's test on Table 9.47. P-value of the test is 4.29×10^{-9}, thus there is difference in at least one pair of setups and post-hoc analysis, presented in Table 9.48 is conducted. The analysis mainly confirms our observations: both SGX and SDX setups provide higher probabilities of effective applications than both TX and CX setups. There is no other significant difference.

We also verify how effectiveness of particular operators influences semantic diversity of entire population. Figure 9.13 presents average and 95% confidence interval of number of semantically unique programs in population of size 1000.

Similarly like in Figure 9.12, the values are higher for symbolic regression than for Boolean domain, what we also attribute to the infinite and continuous structure of semantic space. For

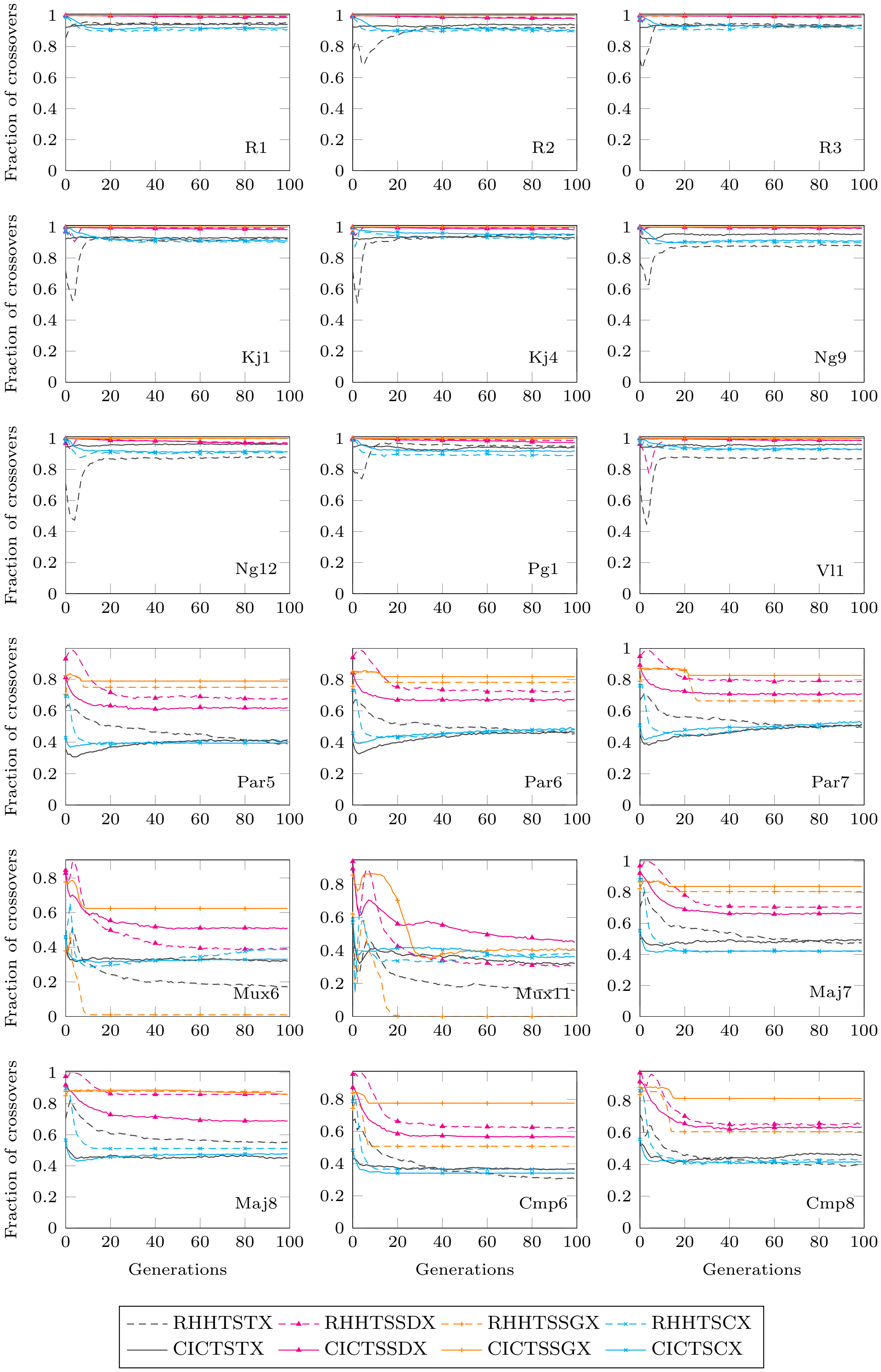


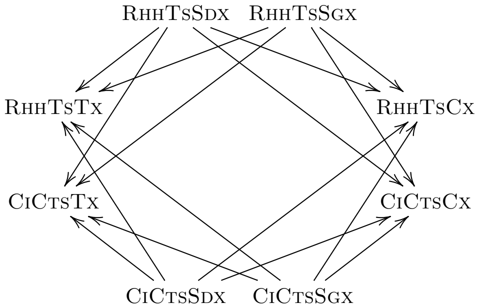
Figure 9.12: Empirical probability and 95% confidence interval of effective crossovers per generation.

Table 9.47: Empirical probability and 95% confidence interval of effective crossovers until 100 generation (the highest in bold).

Prob.	RhhTsTx	RhhTsSdx	RhhTsSgx	RhhTsCx	CiCtsTx	CiCtsSdx	CiCtsSgx	CiCtsCx
R1	0.948 ±0.000	0.997 ±0.000	0.999 ±0.000	0.912 ±0.000	0.942 ±0.000	0.993 ±0.000	**1.000** ±0.000	0.924 ±0.000
R2	0.887 ±0.000	0.990 ±0.000	0.999 ±0.000	0.904 ±0.000	0.936 ±0.000	0.988 ±0.000	**1.000** ±0.000	0.916 ±0.000
R3	0.929 ±0.000	0.997 ±0.000	0.998 ±0.000	0.920 ±0.000	0.933 ±0.000	0.994 ±0.000	**1.000** ±0.000	0.937 ±0.000
Kj1	0.892 ±0.000	0.992 ±0.000	0.998 ±0.000	0.913 ±0.000	0.931 ±0.000	0.990 ±0.000	**1.000** ±0.000	0.924 ±0.000
Kj4	0.917 ±0.000	0.995 ±0.000	0.998 ±0.000	0.935 ±0.000	0.934 ±0.000	0.991 ±0.000	**1.000** ±0.000	0.960 ±0.000
Ng9	0.883 ±0.000	0.993 ±0.000	1.000 ±0.000	0.909 ±0.000	0.950 ±0.000	0.995 ±0.000	**1.000** ±0.000	0.912 ±0.000
Ng12	0.844 ±0.000	0.982 ±0.000	0.999 ±0.000	0.908 ±0.000	0.960 ±0.000	0.981 ±0.000	**1.000** ±0.000	0.920 ±0.000
Pg1	0.940 ±0.000	0.993 ±0.000	1.000 ±0.000	0.897 ±0.000	0.940 ±0.000	0.986 ±0.000	**1.000** ±0.000	0.924 ±0.000
Vl1	0.849 ±0.000	0.983 ±0.000	0.999 ±0.000	0.932 ±0.000	0.955 ±0.000	0.992 ±0.000	**1.000** ±0.000	0.940 ±0.000
Par5	0.470 ±0.001	0.723 ±0.001	0.790 ±0.002	0.420 ±0.001	0.387 ±0.001	0.629 ±0.001	**0.819** ±0.002	0.389 ±0.001
Par6	0.509 ±0.001	0.761 ±0.000	0.832 ±0.001	0.466 ±0.001	0.430 ±0.001	0.679 ±0.001	**0.847** ±0.001	0.458 ±0.001
Par7	0.544 ±0.001	0.818 ±0.000	0.598 ±0.001	0.490 ±0.001	0.468 ±0.001	0.721 ±0.001	**0.868** ±0.001	0.495 ±0.001
Mux6	0.223 ±0.001	0.457 ±0.001	0.025 ±0.000	0.346 ±0.001	0.328 ±0.001	**0.538** ±0.001	0.521 ±0.002	0.323 ±0.001
Mux11	0.223 ±0.001	0.399 ±0.001	0.055 ±0.000	0.368 ±0.001	0.357 ±0.001	**0.540** ±0.001	0.493 ±0.001	0.391 ±0.001
Maj7	0.544 ±0.001	0.811 ±0.001	0.854 ±0.001	0.535 ±0.001	0.482 ±0.001	0.696 ±0.001	**0.865** ±0.001	0.426 ±0.001
Maj8	0.598 ±0.001	**0.933** ±0.001	0.878 ±0.001	0.608 ±0.001	0.458 ±0.001	0.729 ±0.001	0.854 ±0.001	0.463 ±0.001
Cmp6	0.381 ±0.001	0.694 ±0.001	0.698 ±0.002	0.512 ±0.002	0.371 ±0.001	0.622 ±0.001	**0.828** ±0.002	0.367 ±0.001
Cmp8	0.446 ±0.001	0.695 ±0.001	0.820 ±0.001	0.447 ±0.001	0.447 ±0.001	0.657 ±0.001	**0.876** ±0.001	0.422 ±0.001
Rank:	6.556	2.944	2.778	6.333	6.278	3.500	1.222	6.389

Table 9.48: Post-hoc analysis of Friedman's test conducted on Table 9.47: p-values of incorrectly judging a setup in a row as carrying out more effective applications than a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RhhTsTx	RhhTsSdx	RhhTsSgx	RhhTsCx	CiCtsTx	CiCtsSdx	CiCtsSgx	CiCtsCx
RhhTsTx								
RhhTsSdx	**0.000**			**0.001**	**0.001**	0.998		**0.001**
RhhTsSgx	**0.000**	1.000		**0.000**	**0.000**	0.988		**0.000**
RhhTsCx	1.000							1.000
CiCtsTx	1.000			1.000				1.000
CiCtsSdx	**0.005**			**0.012**	**0.015**			**0.009**
CiCtsSgx	**0.000**	0.409	0.547	**0.000**	**0.000**	0.098		**0.000**
CiCtsCx	1.000							



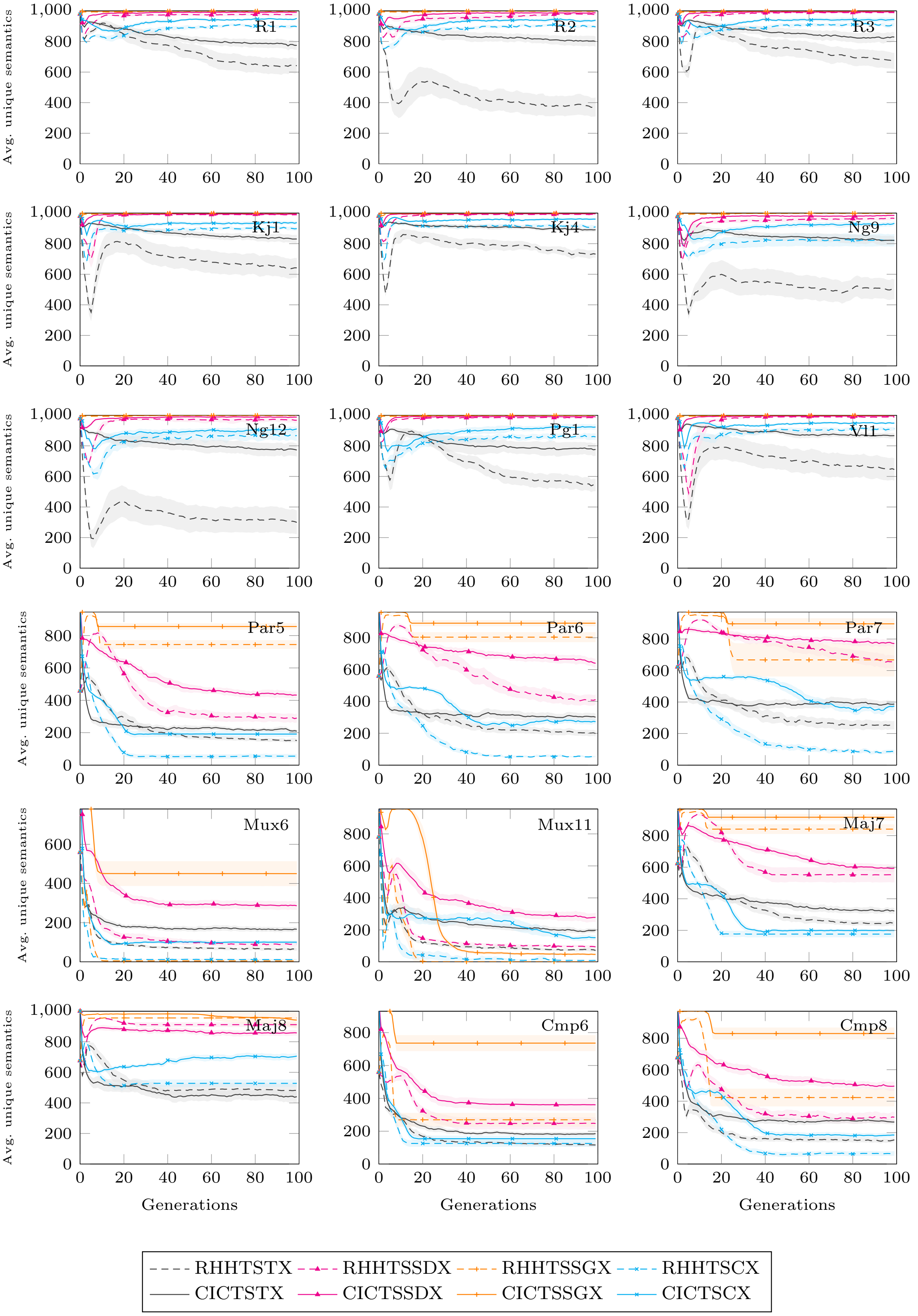


Figure 9.13: Average and 95% confidence interval of semantically unique programs in population of size 1000 over generations.

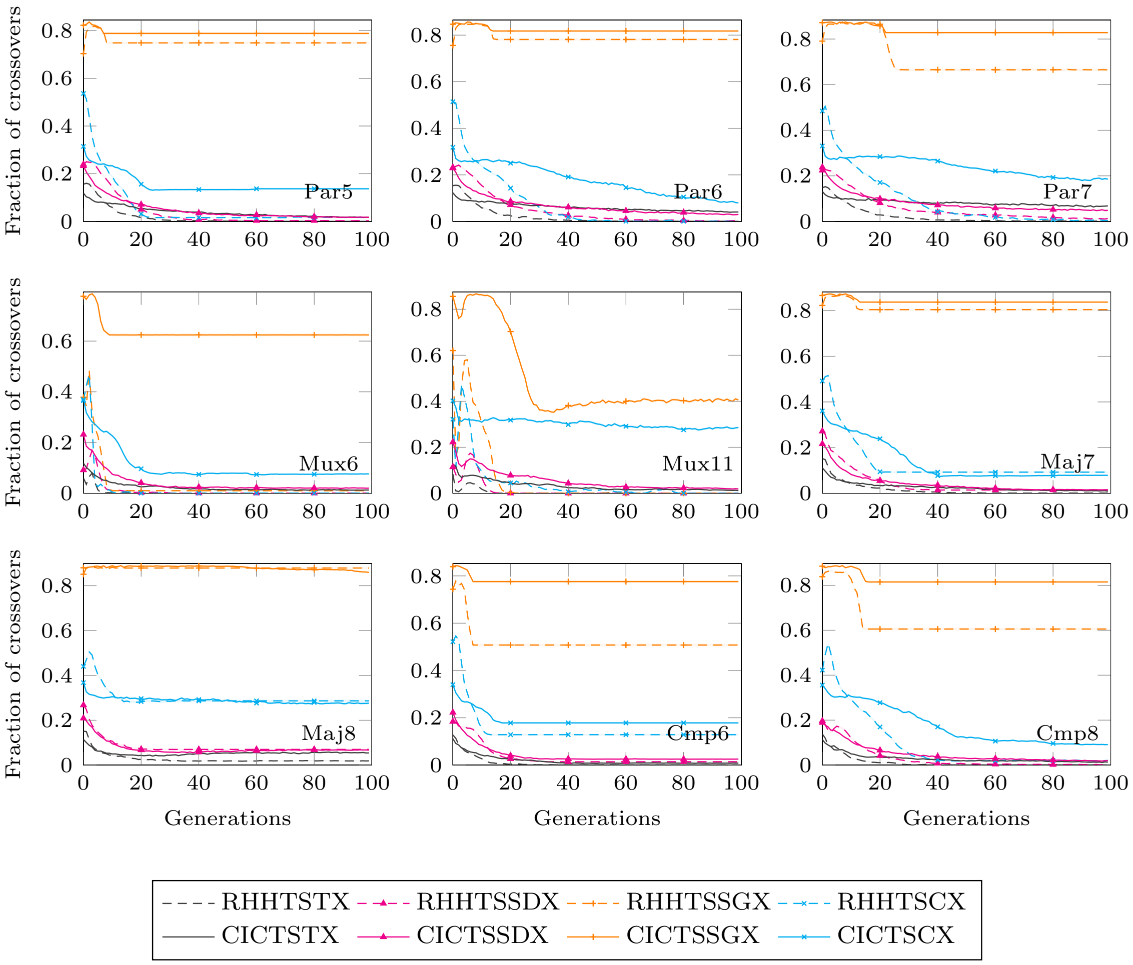


Figure 9.14: Empirical probability and 95% confidence interval of L_1 geometric effective crossovers per generation.

all regression problems all setups that employ CI and CTS achieve higher diversity than the corresponding setups with RHH and TS. Both SGX and SDX setups maintain nearly 1000 unique programs in the population for the entire runs and third place is taken by CX. Diversities for RhhTsTx are the lowest for entire runs and noticeably lower than for CiCtsTx.

In turn in Boolean domain SGX in first few generations slightly increases the number of unique programs to almost 1000, then a rapid fall of this value begins, which ends when either all runs finds the optimum or the number falls down to 1. Because of superior performance of SGX in Boolean domain, we observe actual drop to 1 only in difficult to SGX problems, i.e., Mux*. Due to inability to select semantically distinct parents in low diverse populations, SGX is unable to produce simultaneously geometric and effective offspring, which explains the drop of probability of effective applications observed in Figure 9.12 for Mux* problems.

In turn diversity in populations maintained by TX, SDX and CX slowly falls down in early generations then stays at roughly constant levels. For all of them noticeably higher values are seen for CiCts* variants than for RhhTs* ones.

Figure 9.14 and Table 9.49 present empirical probability and 95% confidence interval of L_1 geometric applications (cf. Definition 5.7) of crossover in Boolean problems, respectively over generations and total in 100-generational run. Figure 9.15 and Table 9.51 present the same values for L_2 metric and symbolic regression problems. Since neutral applications are geometric by definition, we exclude them from statistics to not obfuscate the results.

Table 9.49: Empirical probability and 95% confidence interval of L_1 geometric crossovers until 100 generation (the highest in bold).

Prob.	RhhTsTx	RhhTsSdx	RhhTsSgx	RhhTsCx	CiCtsTx	CiCtsSdx	CiCtsSgx	CiCtsCx
Par5	0.018 ±0.000	0.043 ±0.000	0.790 ±0.002	0.079 ±0.000	0.040 ±0.000	0.054 ±0.000	**0.819** ±0.002	0.201 ±0.001
Par6	0.020 ±0.000	0.047 ±0.000	0.832 ±0.001	0.071 ±0.000	0.062 ±0.000	0.066 ±0.000	**0.847** ±0.001	0.176 ±0.000
Par7	0.023 ±0.000	0.057 ±0.000	0.598 ±0.001	0.090 ±0.000	0.082 ±0.000	0.080 ±0.000	**0.868** ±0.001	0.240 ±0.000
Mux6	0.004 ±0.000	0.010 ±0.000	0.025 ±0.000	0.018 ±0.000	0.030 ±0.000	0.048 ±0.000	**0.521** ±0.002	0.136 ±0.001
Mux11	0.004 ±0.000	0.013 ±0.000	0.055 ±0.000	0.042 ±0.000	0.033 ±0.000	0.054 ±0.000	**0.493** ±0.001	0.302 ±0.001
Maj7	0.017 ±0.000	0.077 ±0.000	0.854 ±0.001	0.267 ±0.001	0.028 ±0.000	0.048 ±0.000	**0.865** ±0.001	0.174 ±0.001
Maj8	0.035 ±0.000	0.122 ±0.001	**0.878** ±0.001	0.359 ±0.001	0.054 ±0.000	0.075 ±0.000	0.854 ±0.001	0.290 ±0.001
Cmp6	0.011 ±0.000	0.045 ±0.000	0.698 ±0.002	0.315 ±0.002	0.024 ±0.000	0.066 ±0.000	**0.828** ±0.002	0.249 ±0.001
Cmp8	0.013 ±0.000	0.031 ±0.000	0.820 ±0.001	0.104 ±0.000	0.029 ±0.000	0.049 ±0.000	**0.876** ±0.001	0.208 ±0.001
Rank:	8.000	6.222	2.333	4.000	6.222	5.000	1.111	3.111

Table 9.50: Post-hoc analysis of Friedman's test conducted on Table 9.49: p-values of incorrectly judging a setup in a row as carrying out more L_1 geometric applications than a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RhhTsTx	RhhTsSdx	RhhTsSgx	RhhTsCx	CiCtsTx	CiCtsSdx	CiCtsSgx	CiCtsCx
RhhTsTx								
RhhTsSdx	0.786							
RhhTsSgx	**0.000**	**0.017**		0.837	**0.017**	0.289		0.998
RhhTsCx	**0.012**	0.534			0.534	0.989		
CiCtsTx	0.786	1.000						
CiCtsSdx	0.156	0.965			0.965			
CiCtsSgx	**0.000**	**0.000**	0.965	0.195	**0.000**	**0.017**		0.666
CiCtsCx	**0.001**	0.125		0.995	0.124	0.728		

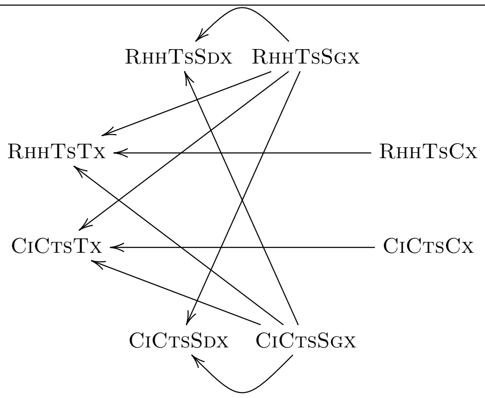
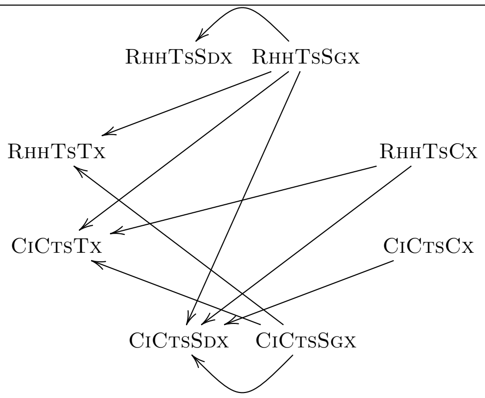


Table 9.51: Empirical probability and 95% confidence interval of L_2 geometric crossovers until 100 generation (the highest in bold).

Prob.	RhhTsTx	RhhTsSdx	RhhTsSgx	RhhTsCx	CiCtsTx	CiCtsSdx	CiCtsSgx	CiCtsCx
R1	0.000 ±0.000	0.000 ±0.000	**0.967** ±0.000	0.001 ±0.000	0.000 ±0.000	0.000 ±0.000	0.960 ±0.000	0.000 ±0.000
R2	0.000 ±0.000	0.000 ±0.000	**0.990** ±0.000	0.003 ±0.000	0.000 ±0.000	0.000 ±0.000	0.982 ±0.000	0.000 ±0.000
R3	0.001 ±0.000	0.001 ±0.000	**0.994** ±0.000	0.007 ±0.000	0.000 ±0.000	0.000 ±0.000	0.989 ±0.000	0.003 ±0.000
Kj1	0.002 ±0.000	0.004 ±0.000	**0.994** ±0.000	0.019 ±0.000	0.000 ±0.000	0.000 ±0.000	0.990 ±0.000	0.005 ±0.000
Kj4	0.001 ±0.000	0.003 ±0.000	**0.993** ±0.000	0.016 ±0.000	0.000 ±0.000	0.000 ±0.000	0.987 ±0.000	0.002 ±0.000
Ng9	0.000 ±0.000	0.000 ±0.000	**0.974** ±0.000	0.003 ±0.000	0.000 ±0.000	0.000 ±0.000	0.963 ±0.000	0.000 ±0.000
Ng12	0.000 ±0.000	0.001 ±0.000	**0.979** ±0.000	0.006 ±0.000	0.000 ±0.000	0.000 ±0.000	0.968 ±0.000	0.002 ±0.000
Pg1	0.000 ±0.000	0.000 ±0.000	**0.961** ±0.000	0.009 ±0.000	0.000 ±0.000	0.000 ±0.000	0.953 ±0.000	0.004 ±0.000
Vl1	0.001 ±0.000	0.003 ±0.000	**0.981** ±0.000	0.021 ±0.000	0.000 ±0.000	0.000 ±0.000	0.968 ±0.000	0.010 ±0.000
Rank:	5.556	5.000	1.000	3.000	7.000	8.000	2.000	4.444

Table 9.52: Post-hoc analysis of Friedman's test conducted on Table 9.51: p-values of incorrectly judging a setup in a row as carrying out more L_2 geometric applications than a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RhhTsTx	RhhTsSdx	RhhTsSgx	RhhTsCx	CiCtsTx	CiCtsSdx	CiCtsSgx	CiCtsCx
RhhTsTx					0.916	0.404		
RhhTsSdx	1.000				0.666	0.156		
RhhTsSgx	**0.002**	**0.012**		0.666	**0.000**	**0.000**	0.989	0.057
RhhTsCx	0.344	0.667			**0.013**	**0.000**		0.916
CiCtsTx						0.989		
CiCtsSdx								
CiCtsSgx	**0.043**	0.156		0.989	**0.000**	**0.000**		0.404
CiCtsCx	0.980	1.000			0.343	**0.043**		



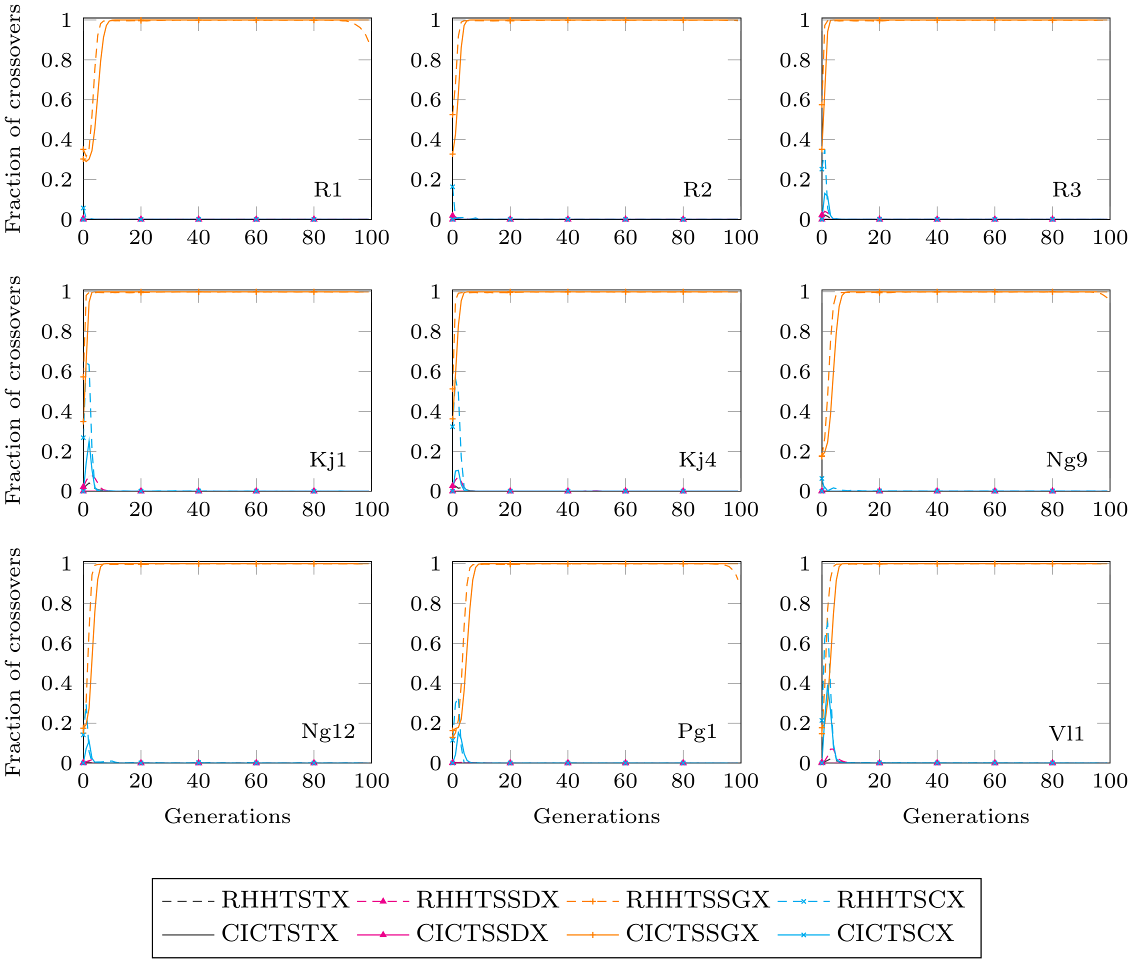


Figure 9.15: Empirical probability and 95% confidence interval of L_2 geometric effective crossovers per generation.

In all Boolean problems CiCtsSgx maintains the highest probability of L_1 geometric applications and RhhTsSgx is the second most geometric for all problems except Mux*, for which the probability rapidly falls down to zero in first few generations. We attribute this to the drop of diversity in population observed earlier in Figure 9.13, thus inability to provide effective geometric applications. Second most geometric operator is CX, for which CiCtsCx setup is characterized by noticeably higher probability of geometric applications than RhhTsCx in 6 out of 9 problems. For the remaining operators the probability of geometric applications is less than 12.2% and is slightly higher for SDX than SDM. This should not be surprising since both operators are blind to geometry of search space and SDX maintains higher probability of effective applications than TX.

In all symbolic regression problems both SGX setups maintain nearly the same probability of L_2 geometric effective applications of over 95.3% applications. In turn the probability for all other operators is much lower and at the level of $10 - 20\%$ in first 10 generations, then drops down to almost zero. Total values in Table 9.51 indicate that RhhTsCx is the third most geometric setup with average probability of geometric effective applications of approximately 1%. CiCtsTx and CiCtsSdx are least geometric setups with the probabilities of almost zero. For all operators the probabilities in problems with 100 fitness cases are slightly lower, than with 20 fitness cases, which may be an effect of complex construction of segment in high dimensional spaces.

The probabilities for SGX are not 100% for all problems due to floating point arithmetic errors or neutral applications, which we excluded from statistics and which constitute the complement probability.

To statistically support our observations, we carry out Friedman's tests on final probabilities separately for Boolean and real geometry. The tests result in conclusive p-values of 3.31×10^{-8} and 1.82×10^{-8}, respectively, thus we conduct separate post-hoc analyzes in Tables 9.50 and 9.52, respectively. The analysis for Boolean domain shows that both SGX setups are significantly more geometric than both TX and both SDX setups. In addition CX is considered more geometric than TX within each pair of initialization and selection. In real domain both SGX setups provide significantly more geometric applications than both TX setups and CiCtsSdx, and RhhTsSgx provides more geometric applications than RhhTsSdx. Additionally both CX setups are more geometric than CiCtsSdx.

9.4.4 Program size

Figure 9.16 presents average and 95% confidence interval of number of nodes in the best of generation program and Table 9.53 presents the same values for the best of run program.

This experiment confirms our remark from Section 6.2.1 on size of programs produced by SGX that is exponential in number of generations and reach millions of nodes in less than 20 generations for all problems. The observed growth halts only when all SGX runs find the optimum, i.e., all runs terminate. Hence SGX is the most bloating of all considered operators.

Second biggest programs are produced by CX, however their final sizes are typically of few hundreds nodes and only for six problems the size grows over one thousand nodes. In turn SDX and TX produce smaller programs than CX in all problems except Mux*, where RhhTsCx results in the smallest programs. Programs produced by TX are smaller than the programs produced by SDX accompanied with the same initialization and selection in 15 out of 18 problems. What is more use of CI and CTS for all operators leads to programs not smaller than programs produced with support of RHH and TS.

To statistically compare final program sizes, we employ Friedman's test that results in conclusive p-value of 1.42×10^{-14}. We conduct post-hoc analysis presented in Table 9.54. Both TX and both SDX setups produce significantly smaller programs than both SGX setups. RhhTsCx produces significantly smaller programs than both SGX setups. RhhTsTx produces smaller programs than both CX setups and CiCtsTx produces smaller programs than CiCtsCx.

9.4.5 Computational costs

Figure 9.17 and Table 9.55 present average and 95% confidence interval of total CPU time consumed by the considered operators over generations and until 100 generation, respectively.

First, we observe noticeably impact of the choice of initialization method on the total computation time: from all setups, these that involve CI require more total time than the corresponding setups that employ RHH. Second, CX in symbolic regression is the most time-demanding operator and SDX is the second. The fastest operator is TX and SGX is slightly slower than TX. The situation is quite different in Boolean domain, where in 6 out of 9 problems CiCtsSdx is the slowest setup and RhhTsSdx is only slightly faster. In the remaining problems CiCtsCx is the slowest one. In turn the fastest setup is RhhTsSgx in all but Mux* problems, where RhhTsTx is the fastest. We explain this phenomenon by the fact that Mux* problems are the only Boolean ones that are challenging for SGX to solve, hence it requires more time to find the optimum.

We conduct Friedman's test to assess significance of differences between obtained total computation times. The test outputs conclusive p-value of 1.47×10^{-12}, thus we carry out post-hoc analysis in Table 9.56. Indeed the times consumed by both CX setups are greater than both SGX setups and RhhTsTx. Additionally RhhTsTx is faster than both SDX setups and CiCtsTx,

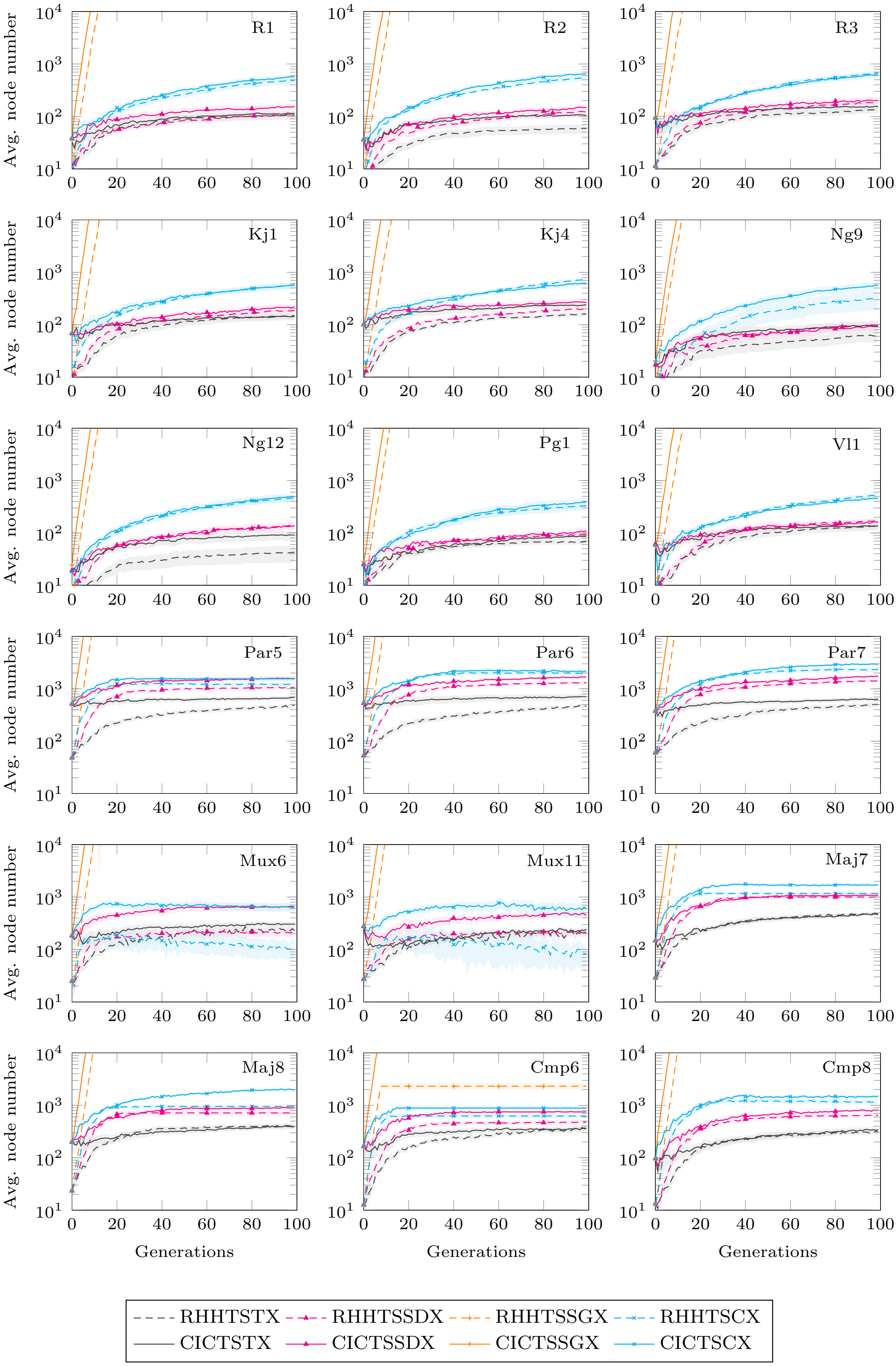


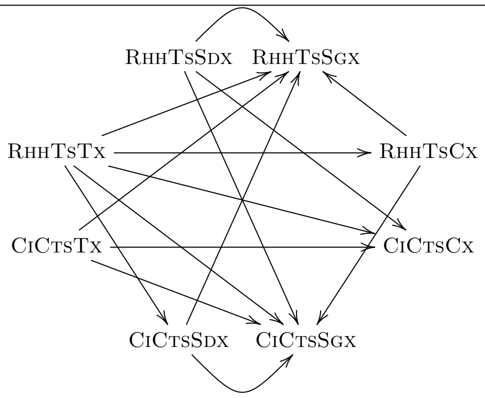
Figure 9.16: Average and 95% confidence interval of numbers of nodes in the best of generation program.

Table 9.53: Average and 95% confidence interval of number of nodes in the best of run program (the lowest in bold). For values greater than 10^4 we left only an order of magnitude.

Prob.	RhhTsTx	RhhTsSdx	RhhTsSgx	RhhTsCx	CiCtsTx	CiCtsSdx	CiCtsSgx	CiCtsCx
R1	**106.40** ±18.44	106.73 ±13.75	10^{17} $\pm 10^{16}$	490.13 ±74.07	114.40 ±13.97	154.00 ±24.55	10^{17} $\pm 10^{16}$	569.33 ±68.11
R2	**57.33** ±11.49	122.20 ±16.03	10^{17} $\pm 10^{16}$	552.53 ±60.59	106.10 ±12.27	144.87 ±21.49	10^{17} $\pm 10^{16}$	654.67 ±82.84
R3	**131.07** ±19.11	190.03 ±20.83	10^{17} $\pm 10^{16}$	692.77 ±76.50	153.20 ±23.92	206.77 ±33.29	10^{17} $\pm 10^{16}$	664.00 ±77.81
Kj1	148.00 ±24.42	185.83 ±21.32	10^{17} $\pm 10^{16}$	572.60 ±72.84	**144.60** ±17.99	222.73 ±33.41	10^{17} $\pm 10^{16}$	582.57 ±88.00
Kj4	**158.90** ±15.82	199.43 ±21.73	10^{17} $\pm 10^{16}$	735.33 ±79.46	236.80 ±30.05	270.30 ±50.54	10^{17} $\pm 10^{16}$	622.27 ±64.47
Ng9	**63.37** ±17.09	101.97 ±20.18	10^{17} $\pm 10^{16}$	312.53 ±123.12	93.07 ±19.07	100.63 ±18.11	10^{17} $\pm 10^{16}$	569.57 ±108.27
Ng12	**42.23** ±15.31	139.37 ±23.20	10^{17} $\pm 10^{16}$	473.13 ±72.78	91.10 ±18.66	136.00 ±20.35	10^{17} $\pm 10^{16}$	503.70 ±85.67
Pg1	**69.17** ±7.77	93.10 ±9.74	10^{17} $\pm 10^{16}$	328.07 ±59.97	86.43 ±10.97	109.27 ±17.85	10^{17} $\pm 10^{17}$	392.13 ±88.47
Vl1	**134.03** ±23.07	170.37 ±20.89	10^{17} $\pm 10^{16}$	527.53 ±66.34	135.50 ±14.95	156.60 ±26.41	10^{17} $\pm 10^{16}$	463.03 ±45.41
Par5	**471.13** ±45.90	1049.73 ±90.35	10^{4} ±1048.14	1210.93 ±111.24	681.40 ±69.58	1576.00 ±181.99	10^{5} ±7071.28	1551.87 ±143.66
Par6	**479.73** ±42.97	1311.93 ±108.54	10^{6} $\pm 10^{5}$	1959.07 ±189.07	717.13 ±90.83	1685.53 ±159.96	10^{7} $\pm 10^{6}$	2045.47 ±163.67
Par7	**516.07** ±44.49	1418.80 ±125.01	10^{16} $\pm 10^{16}$	2304.00 ±242.46	629.20 ±49.63	1764.47 ±161.41	10^{9} $\pm 10^{8}$	2965.20 ±185.99
Mux6	250.87 ±33.93	221.07 ±43.17	10^{17} $\pm 10^{16}$	**97.00** ±35.03	319.07 ±38.26	631.40 ±116.76	10^{16} $\pm 10^{16}$	635.00 ±118.69
Mux11	190.20 ±32.60	224.53 ±38.53	10^{17} $\pm 10^{16}$	**73.07** ±31.48	226.07 ±23.75	459.13 ±65.74	10^{17} $\pm 10^{16}$	588.60 ±181.63
Maj7	**469.27** ±40.04	987.13 ±82.95	10^{5} $\pm 10^{4}$	1172.80 ±86.83	476.80 ±37.23	1085.93 ±99.46	10^{6} $\pm 10^{5}$	1706.40 ±142.99
Maj8	409.07 ±46.61	715.93 ±88.36	10^{5} ±6436.94	952.60 ±82.00	**393.33** ±41.13	899.60 ±105.93	10^{16} $\pm 10^{16}$	2043.80 ±159.88
Cmp6	**332.13** ±44.97	475.33 ±50.34	2317.63 ±340.26	626.00 ±58.71	368.13 ±50.34	747.20 ±80.18	10^{4} ±5394.53	886.07 ±98.91
Cmp8	**317.53** ±36.20	650.53 ±63.66	10^{5} $\pm 10^{5}$	1169.33 ±122.68	346.47 ±43.91	798.13 ±76.09	10^{7} $\pm 10^{6}$	1476.73 ±134.48
Rank:	1.28	3.00	7.28	4.61	2.22	4.11	7.72	5.78

Table 9.54: Post-hoc analysis of Friedman's test conducted on Table 9.53: p-values of incorrectly judging a setup in a row as producing smaller programs than a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RhhTsTx	RhhTsSdx	RhhTsSgx	RhhTsCx	CiCtsTx	CiCtsSdx	CiCtsSgx	CiCtsCx
RhhTsTx		0.409	**0.000**	**0.001**	0.944	**0.012**	**0.000**	**0.000**
RhhTsSdx			**0.000**	0.500		0.875	**0.000**	**0.016**
RhhTsSgx							0.999	
RhhTsCx			**0.024**				**0.003**	0.844
CiCtsTx		0.981	**0.000**	0.067		0.286	**0.000**	**0.000**
CiCtsSdx			**0.003**	0.999			**0.000**	0.454
CiCtsSgx								
CiCtsCx			0.594				0.251	



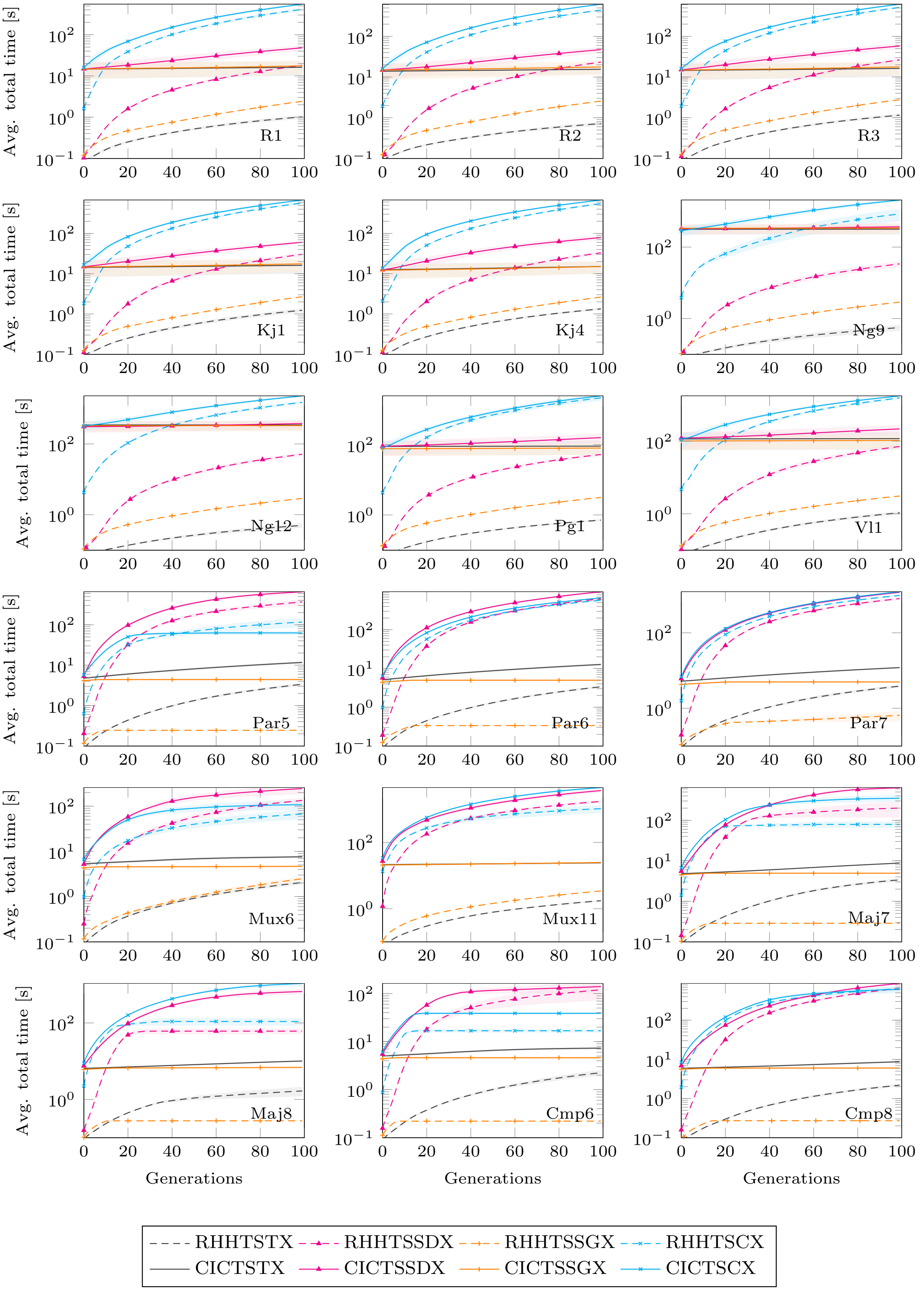


Figure 9.17: Average and 95% confidence interval of total execution time w.r.t. number of generations.

Table 9.55: Average and 95% confidence interval of total execution time spent to finish run (the lowest in bold).

Prob.	RhhTsTx	RhhTsSdx	RhhTsSgx	RhhTsCx	CiCtsTx	CiCtsSdx	CiCtsSgx	CiCtsCx
R1	**1.02** ±0.11	17.98 ±1.89	2.42 ±0.07	411.56 ±40.60	16.25 ±6.15	48.00 ±7.44	17.59 ±6.35	549.09 ±40.57
R2	**0.72** ±0.07	23.36 ±3.06	2.57 ±0.09	442.11 ±28.48	15.53 ±3.84	47.25 ±7.73	17.72 ±6.31	617.22 ±35.49
R3	**1.16** ±0.10	26.43 ±2.94	2.76 ±0.09	514.26 ±41.22	16.29 ±6.13	57.83 ±9.57	17.76 ±6.33	616.08 ±36.44
Kj1	**1.24** ±0.15	30.37 ±3.63	2.68 ±0.07	569.83 ±45.47	16.18 ±6.05	60.58 ±6.62	17.62 ±6.32	675.85 ±51.20
Kj4	**1.34** ±0.09	32.92 ±3.62	2.62 ±0.09	548.34 ±40.79	15.07 ±5.03	78.78 ±10.93	15.02 ±4.93	671.80 ±35.91
Ng9	**0.56** ±0.11	33.94 ±6.27	2.88 ±0.07	851.11 ±323.91	318.34 ±93.47	370.06 ±99.45	340.84 ±102.38	2074.41 ±279.46
Ng12	**0.50** ±0.07	51.03 ±6.24	2.88 ±0.05	1496.03 ±160.57	344.59 ±103.00	376.97 ±98.58	330.34 ±102.60	2274.75 ±181.16
Pg1	**0.71** ±0.05	52.29 ±6.29	3.15 ±0.12	2133.81 ±382.98	90.62 ±39.57	158.06 ±42.93	79.93 ±29.42	2426.83 ±356.24
Vl1	**1.06** ±0.12	71.02 ±10.38	3.08 ±0.10	1546.61 ±157.72	117.10 ±58.28	217.26 ±59.55	104.66 ±43.47	1756.09 ±95.06
Par5	3.36 ±0.27	367.31 ±59.37	**0.24** ±0.01	116.62 ±30.87	11.68 ±0.90	656.40 ±90.67	4.40 ±0.26	62.89 ±7.09
Par6	3.37 ±0.24	630.13 ±57.08	**0.34** ±0.01	578.36 ±42.12	12.70 ±0.96	956.57 ±73.57	4.97 ±0.31	661.82 ±30.88
Par7	3.85 ±0.36	807.16 ±79.93	**0.65** ±0.21	977.58 ±70.13	11.93 ±0.66	1186.17 ±103.28	4.97 ±0.28	1224.72 ±62.87
Mux6	**2.05** ±0.21	134.10 ±20.74	2.49 ±0.28	67.39 ±15.83	7.59 ±0.56	245.35 ±60.86	4.66 ±0.34	107.99 ±26.61
Mux11	**1.74** ±0.10	1728.24 ±218.04	3.43 ±0.10	1046.10 ±282.24	24.19 ±1.45	3678.56 ±510.84	24.77 ±1.57	4522.62 ±406.16
Maj7	3.38 ±0.30	200.77 ±84.52	**0.29** ±0.01	79.38 ±14.06	8.78 ±0.30	637.29 ±92.74	4.94 ±0.27	349.82 ±70.29
Maj8	1.66 ±0.47	61.06 ±10.37	**0.28** ±0.01	109.36 ±24.25	10.10 ±0.56	663.88 ±115.51	6.88 ±0.54	1079.50 ±171.16
Cmp6	2.25 ±0.30	120.32 ±47.84	**0.22** ±0.01	16.85 ±1.49	7.32 ±0.49	140.06 ±42.01	4.62 ±0.24	38.86 ±2.32
Cmp8	2.19 ±0.17	651.44 ±55.43	**0.27** ±0.01	640.75 ±105.10	8.69 ±0.56	869.91 ±67.22	6.00 ±0.44	614.00 ±124.29
Rank:	1.39	5.17	1.61	6.22	3.89	6.83	3.56	7.33

Table 9.56: Post-hoc analysis of Friedman's test conducted on Table 9.55: p-values of incorrectly judging a setup in a row as faster than a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	RhhTsTx	RhhTsSdx	RhhTsSgx	RhhTsCx	CiCtsTx	CiCtsSdx	CiCtsSgx	CiCtsCx
RhhTsTx		**0.000**	1.000	**0.000**	**0.045**	**0.000**	0.137	**0.000**
RhhTsSdx				0.902		0.454		0.137
RhhTsSgx		**0.000**		**0.000**	0.097	**0.000**	0.250	**0.000**
RhhTsCx						0.995		0.875
CiCtsTx		0.771		0.081		**0.008**		**0.001**
CiCtsSdx								0.999
CiCtsSgx		0.500		**0.024**	1.000	**0.002**		**0.000**
CiCtsCx								

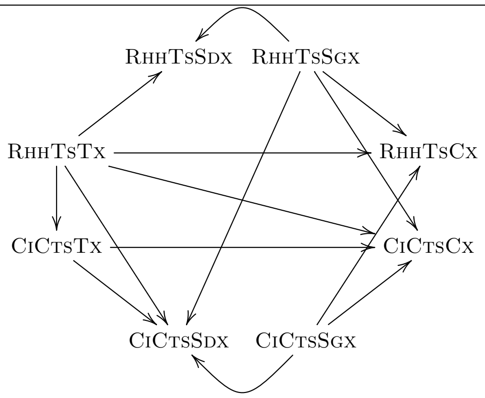


Table 9.57: Spearman's rank correlation coefficient of number of nodes in produced programs and computational time consumed by a setup. Left: correlation over problems for each setup separately. Right: rankings of setups from last rows of Tables 9.53 and 9.55, and correlation of them. P-values of t test of significance of correlation, significant correlations ($\alpha = 0.05$) are marked in bold.

	Correlation	P-value	Size rank	Time rank
RhhTsTx	**0.965**	0.000	1.28	1.39
RhhTsSdx	**0.748**	0.000	3.00	5.17
RhhTsSgx	**0.872**	0.000	7.28	1.61
RhhTsCx	−0.255	0.154	4.61	6.22
CiCtsTx	−**0.765**	0.000	2.22	3.89
CiCtsSdx	**0.620**	0.003	4.11	6.83
CiCtsSgx	**0.938**	0.000	7.72	3.56
CiCtsCx	−**0.461**	0.027	5.78	7.33
		Correlation:	0.061	
		P-value:	0.443	

Table 9.58: Exemplary analysis for time budget of 60 seconds for Kj4 problem. The highest generation and the lowest statistics are in bold.

	Generation	Fitness	Test-set fitness	Program size
RhhTsTx	**100**	**0.17** ±0.03	0.63 ≤ **1.14** ≤ 3.39	**158.90** ±15.82
RhhTsSdx	**100**	**0.17** ±0.03	1.43 ≤ 2.23 ≤ 3.89	199.43 ±21.73
RhhTsSgx	**100**	0.96 ±0.03	2.19 ≤ 2.23 ≤ 2.29	10^{17} $\pm 10^{16}$
RhhTsCx	22	0.21 ±0.04	0.89 ≤ 1.67 ≤ 3.11	187.63 ±16.91
CiCtsTx	**100**	0.18 ±0.02	1.65 ≤ 2.52 ≤ 6.71	236.80 ±30.05
CiCtsSdx	76	0.18 ±0.02	1.98 ≤ 2.36 ≤ 2.88	248.17 ±41.28
CiCtsSgx	**100**	0.67 ±0.04	2.35 ≤ 2.38 ≤ 2.45	10^{17} $\pm 10^{16}$
CiCtsCx	12	0.67 ±0.04	1.79 ≤ 2.25 ≤ 3.06	201.43 ±37.26

where the latter one is faster than CiCtsSdx. Hence we consider canonical setup RhhTsTx as the fastest one and ex aequo CiCtsCx and CiCtsSdx as the slowest ones.

We verify if computational costs of *a particular setup* and size of the produced programs from Table 9.53 are correlated by calculating Spearman's rank correlation coefficients. The coefficients are presented with p-values of t test for significance of correlation in left part of Table 9.57. For both SDX and both SGX setups we observe strong positive correlation between size and computational costs, hence enforcement of limits on program size would decrease computational costs of a setup. Note that by definition of SGX, the application of a limit on size would effectively mean termination of the evolutionary run, since SGX is able to produce only bigger programs than the given parents. In turn the correlation for CiCtsTx is strongly negative. We hypothesize that this is an anomaly that comes from high computational costs of CI that constitute a major part of the total computational costs and as such may disturb the results. Negative but lower correlation is observed also for both CX setups, for which probably exists other factor than program size that explain computational costs.

Right part of Table 9.57 presents aggregated rankings of *all setups* took from Tables 9.53 and 9.55, Spearman's rank correlation coefficient and p-value of t test for significance of correlation. Unlike in similar analysis for mutation, there is no correlation between orderings of the operators in program size and computational costs. For instance SGX that produces oversized programs has one of the lowest computational costs. The obvious reason is that SGX only combines given parents and does not penetrate their structures like the other operators do.

However CX and SDX are characterized by higher computational costs than the remaining operators, we argue that given a limited computational time budget they may still achieve prominent results. To give a sight on the comparison of crossover operators in environment with limited time budget, similarly like in Section 9.3.5 we recommend to put a horizontal line in a plot in Figure 9.17 that reflects the budget, next read number of generations reached by each setup, finally compare the interested statistics from the previous sections in that generations.

Particular time budget is dependent on, e.g., methodological, technical and project schedule aspects, thus we leave to the reader a detailed analysis and present only an example of time budget of 60 seconds for Kj4 problem. Table 9.58 shows generation reached on average by each setup within 60 seconds and corresponding statistics of training and test-set fitness and size of produced programs. Both TX setups, both SGX setups and RhhTsSdx finished in the time limit. The slowest setup is CiCtsCx that reaches only 12 generations. RhhTsTx achieves the best training and test-set fitness and produces the smallest programs, however the training-set fitness of RhhTsSdx is indiscernible with RhhTsTx. What is more confidence intervals of all statistics of the best setup have non-empty intersections with confidence intervals for both SDX setups and RhhTsCx. This clearly indicates that even if computational costs of semantic operators are higher their results in limited time may be as good as of the canonical operators, however given more time, semantic operators are able to overcome the canonical ones.

9.5 Conclusions

The evidence gathered in the conducted experiments allows us to answer the research questions $1 - 4$ from Section 8.1.

The answer to research question 1 on superior effectiveness and geometry of CI to SDI and RHH is *yes*.

CI initializes populations without semantic duplicates, more diverse than RHH and equally diverse as SDI. CI also builds geometric population in fewer applications than SDI and RHH. What is more CI provides smoother fitness distribution than the other operators, which may indicate a more uniform distribution of semantics. Admittedly both SDI and CI pay a price in a program size and computational costs for effectiveness and, in case of CI also for geometry. While the increase of computational cost resulting from SDI's attempts to conduct an effective application seems to be low, the cost for performing a geometric application by CI is up to two orders of magnitude higher.

Note that both SDI and CI build programs incrementally, starting from single nodes and combining them gradually into larger and larger ones. Thus the more times the operator is applied the bigger programs it produces on average. Limiting size of the population would result in smaller average size of programs. This may be tempting, especially for CI, which requires less applications, i.e., smaller population, than the remaining operators to conduct the first geometric application.

The answer to research question 2 on superior effectiveness and geometry of CTS to TS and STS is *yes*.

CTS selects effective and geometric parents more frequently than TS and STS, however the gain in probability of geometric selection is too low to draw statistically significant conclusions. Unfortunately these features of CTS are burdened with higher computation costs of this operator than the remaining ones, however the absolute times are rather low.

The distribution of semantics in population biases the characteristics of the selection operators. CI-initialized population facilitates selection of distinct parents for all operators and lowers execution time of STS and CTS, while it also hinders selection of geometric parents. In general case, where the population may not be diverse (like, e.g., the RHH's one), it seems to be a better choice to employ CTS than an other selection operator.

The answer to research question 3 on superior characteristics of CM to TM, SDM, SGM, and on positive impact of use of CI in place of RHH is *partially positive and depends on problem domain*.

CM on average achieves better training-set fitness than TM, SDM and SGM, and if accompanied with RHH, also better median test-set fitness than TM, SDM and SGM. CM provides lower probability of effective mutations than SDM and SGM, however the probability is still relatively high (0.9) and higher than for TM. In turn CM maintains greater numbers of semantically distinct programs in a population than SGM during the course of evolution for most of the considered benchmarks. CM also keeps the smallest distance of an offspring to a parent of all operators. Unfortunately CM pays for these beneficial statistics in size of produced programs and in high computational cost.

From the perspective of fitness-based performance, the choice of initialization operator between RHH and CI is of little importance for both training and test sets. We only observe a small bias toward RHH that causes CM to achieve slightly better fitness. The initialization method also does not have statistically significant impact on probability of producing effective offspring, the distance of a parent to an offspring, or size of final programs produced by CM. The choice of CI negatively impacts only a computational cost.

Although programs evolved by CM are bigger than evolved by other mutation operators, they still generalize well and achieve better fitness on test-set. CM usually finds in $30 - 50$ generations a program with better fitness than any other mutation operator within 100 generations. What is more, the average sizes of a program produced by CM in 30 generation and a one produced by an other operator at the end of run are roughly the same. This remark also addresses computational costs: CM achieves better fitness in fewer generations and with the same time budget like other operators.

The answer to research question 4 on superior characteristics of CX to TX, SDX, SGX, and on positive impact of use of CI and CTS in place of RHH and TS is *partially positive and depends on problem domain*.

CX achieves superior training-set fitness to TX, SDX and SGX in symbolic regression domain, and is the second best after SGX in synthesis of Boolean expressions. Although SGX converges to an optimum very quickly, it roughly doubles sizes of given parents in offspring. This characteristic in turn may be considered as a source of major overfitting of SGX and calls its practical applicability into the question. CX is free of above mentioned disadvantages, it outperforms SGX in terms of test-set fitness and average size of produced programs. TX and SDX still produce smaller programs than CX, however the gap in program size between them and CX is lower than for CX and SGX. From the perspective of features that motivated the design of CX, CX maintains high ratio of effective applications for entire evolutionary run, however SGX and SDX are characterized by slightly higher ratios. CX maintains the second highest probability of producing geometric offspring, giving the way only to exact geometric SGX. This demonstrates that the design goals are achieved. Unfortunately, CX pays for its features in computational cost comparable with the cost of SDX and higher than the costs of TX and SGX. Nevertheless, given even a strict time budget, CX is still able to provide competitive results.

The support of CI and CTS in general helps CX achieve better fitness than the use of RHH and TS. However the gain in fitness is noticeable, it is too weak to be confirmed statistically. The choice of initialization and selection method seems to have even lower impact on the fitness achieved by the remaining crossover operators. CI and CTS clearly support effectiveness of all operators, diversity in population and geometry of crossover applications, however causes increase of overall computational cost. Therefore, we recommend use of CI and CTS together with CX.

Chapter 10

Evaluation of combinations of Competent Algorithms for operators

In the previous chapter, we investigated characteristics of competent algorithms in isolation. In this chapter, we use them together, investigate how well they cooperate with each other, and determine the most efficient proportions of crossover and mutation, answering so the research question 5. We compare fitness-based performance on training and test-set, size of produced programs and computational costs.

In all setups we use CI and CTS as initialization and selection operators, respectively. We use three *mixed setups* with different proportions of CX crossover from Section 7.6 and CM mutation from Section 7.5, respectively: 75% : 25%, 50% : 50% and 25% : 75%. There are two *reference setups* that act as baselines of pure crossover and pure mutation setups: one employs CX and one CM as the only variation operator. The reference setups were used in experimental analysis in Sections 9.4 and 9.3, respectively. This allows us to compare the mixed setups with the ones presented previously. In total, there are five experimental setups: CiCtsCx, CiCtsCx75Cm, CiCtsCx50Cm, CiCtsCx25Cm, CiTsCm (see Section 8.3 for explanation on naming convention).

10.1 Fitness-based Performance

Figure 10.1 presents average and 95% confidence interval of fitness of the best of generation program. Table 10.1 presents the same data for the best of run program.

The characteristics of setups are close, and the trials of the mixed setups are even more tightly related. The performance of the reference setups is noticeably worse than the performance of the mixed setups for 15 out of 18 problems, however it is difficult to clearly determine whether CiCtsCx or CiTsCm is worse. CiCtsCx75Cm and CiCtsCx25Cm ex aequo achieve the lowest best of run fitness for 10 out of 18 problems. The former leads in 4 regression and 6 Boolean problems, and the latter in 3 and 7 problems, respectively. CiCtsCx50Cm achieves the best fitness in 8 out of 9 Boolean problems and none of symbolic regression problems. CiTsCm achieves the best fitness in only two problems, which is the worst result.

We carry out Friedman's test on the averages of the best of run fitness to determine whether there are significant differences between the setups. The test results in p-value of 7.52×10^{-6} at significance level $\alpha = 0.05$, so at least two setups differ significantly. The post-hoc analysis presented in Table 10.2 confirms that CiTsCm is significantly worse than all mixed setups and CiCtsCx is worse than CiCtsCx25Cm. Therefore CiCtsCx25Cm has the highest number of outrankings and can be considered as the best setup.

To estimate the probability of solving benchmark problems by each setup we count the number of runs that found an optimal program p^*, i.e., $s(p^*) = t$, divide the result by number of all runs,

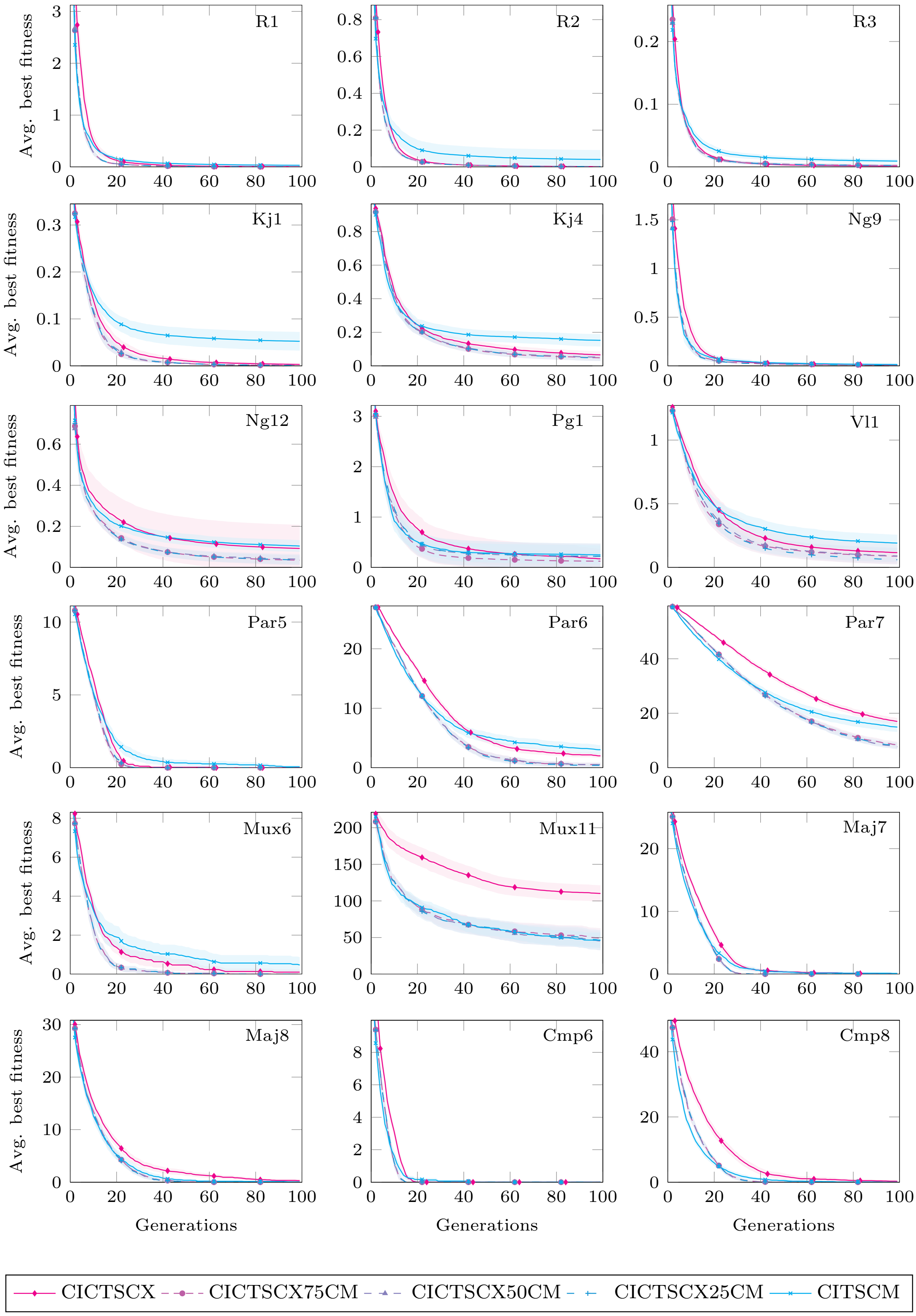


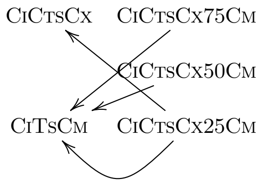
Figure 10.1: Average and 95% confidence interval of the best of generation fitness.

Table 10.1: Average and 95% confidence interval of the best of run fitness (the best in bold).

Problem	CiCtsCx		CiCtsCx75Cm		CiCtsCx50Cm		CiCtsCx25Cm		CiTsCm	
R1	0.007	±0.002	**0.002**	±0.001	0.002	±0.001	0.002	±0.001	0.032	±0.010
R2	**0.002**	±0.001	0.002	±0.001	0.002	±0.001	0.002	±0.001	0.041	±0.051
R3	**0.001**	±0.001	0.002	±0.002	0.002	±0.002	0.002	±0.002	0.009	±0.003
Kj1	0.003	±0.002	**0.001**	±0.000	0.001	±0.001	0.001	±0.000	0.052	±0.020
Kj4	0.065	±0.020	0.048	±0.019	0.051	±0.020	**0.047**	±0.019	0.152	±0.037
Ng9	0.011	±0.004	0.009	±0.004	0.012	±0.006	**0.009**	±0.004	0.014	±0.007
Ng12	0.092	±0.115	**0.035**	±0.024	0.040	±0.025	0.036	±0.024	0.104	±0.029
Pg1	0.168	±0.196	**0.124**	±0.183	0.220	±0.254	0.221	±0.253	0.250	±0.189
Vl1	0.115	±0.079	0.087	±0.065	0.090	±0.064	**0.060**	±0.018	0.191	±0.066
Par5	**0.000**	±0.000	**0.000**	±0.000	**0.000**	±0.000	**0.000**	±0.000	0.067	±0.089
Par6	1.967	±0.339	0.500	±0.303	0.500	±0.303	**0.367**	±0.269	3.000	±0.767
Par7	16.900	±1.258	8.133	±1.179	**7.500**	±0.994	7.867	±1.108	14.733	±1.660
Mux6	0.100	±0.142	**0.000**	±0.000	**0.000**	±0.000	**0.000**	±0.000	0.467	±0.400
Mux11	109.867	±11.099	48.933	±13.718	**45.200**	±12.708	46.267	±13.714	46.400	±10.347
Maj7	0.033	±0.064	**0.000**	±0.000	**0.000**	±0.000	**0.000**	±0.000	0.100	±0.193
Maj8	0.300	±0.210	**0.000**	±0.000	**0.000**	±0.000	**0.000**	±0.000	0.067	±0.089
Cmp6	**0.000**	±0.000	**0.000**	±0.000	**0.000**	±0.000	**0.000**	±0.000	**0.000**	±0.000
Cmp8	0.267	±0.332	**0.000**	±0.000	**0.000**	±0.000	**0.000**	±0.000	**0.000**	±0.000
Rank:	3.583		2.306		2.472		2.111		4.528	

Table 10.2: Post-hoc analysis of Friedman's test conducted on Table 10.1: p-values of incorrectly judging a setup in a row as outranking a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	CiCtsCx	CiCtsCx75Cm	CiCtsCx50Cm	CiCtsCx25Cm	CiTsCm
CiCtsCx					0.296
CiCtsCx75Cm	0.066		0.997		**0.000**
CiCtsCx50Cm	0.150				**0.000**
CiCtsCx25Cm	**0.021**	0.995	0.947		**0.000**
CiTsCm					



and present the probabilities (i.e., *success rate*) in Table 10.3.[1] Unfortunately 8 out of 9 regression problems and 2 out of 9 Boolean problems remain unsolved in all runs of all setups. In 7 out of solved problems, the highest probabilities are achieved by CiCtsCx25Cm, where all seven problems are Boolean ones. What is more CiCtsCx25Cm solves six of these problems in every run. The runners up are ex aequo CiCtsCx75Cm and CiCtsCx50Cm that solve with the highest probability of 1 six Boolean problems. In turn the only solved regression problem, Ng9, is solved with the highest probability by CiTsCm.

To statistically verify the differences among setups, we conduct Friedman's test on the probabilities. Given $\alpha = 0.05$, the test results in p-value of 1.47×10^{-2} thus we carry out post-hoc analysis in Table 10.4. The analysis reveals that CiCtsCx is outranked by CiCtsCx50Cm and CiCtsCx25Cm, what makes CiCtsCx the worst setup.

Table 10.3: Empirical probability and 95% confidence interval of solving problems (the highest in bold).

Problem	CiCtsCx	CiCtsCx75Cm	CiCtsCx50Cm	CiCtsCx25Cm	CiTsCm
R1	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
R2	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
R3	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
Kj1	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
Kj4	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
Ng9	0.10 ±0.11	0.10 ±0.11	0.13 ±0.12	0.13 ±0.12	**0.43** ±0.18
Ng12	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
Pg1	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
Vl1	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
Par5	**1.00** ±0.00	**1.00** ±0.00	**1.00** ±0.00	**1.00** ±0.00	0.93 ±0.09
Par6	0.00 ±0.00	0.70 ±0.16	0.70 ±0.16	**0.77** ±0.15	0.03 ±0.06
Par7	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
Mux6	0.93 ±0.09	**1.00** ±0.00	**1.00** ±0.00	**1.00** ±0.00	0.83 ±0.13
Mux11	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00	**0.00** ±0.00
Maj7	0.97 ±0.06	**1.00** ±0.00	**1.00** ±0.00	**1.00** ±0.00	0.97 ±0.06
Maj8	0.77 ±0.15	**1.00** ±0.00	**1.00** ±0.00	**1.00** ±0.00	0.93 ±0.09
Cmp6	**1.00** ±0.00	**1.00** ±0.00	**1.00** ±0.00	**1.00** ±0.00	**1.00** ±0.00
Cmp8	0.87 ±0.12	**1.00** ±0.00	**1.00** ±0.00	**1.00** ±0.00	**1.00** ±0.00
Rank:	3.53	2.83	2.72	2.64	3.28

Table 10.4: Post-hoc analysis of Friedman's test conducted on Table 10.3: p-values of incorrectly judging a setup in a row as outranking a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	CiCtsCx	CiCtsCx75Cm	CiCtsCx50Cm	CiCtsCx25Cm	CiTsCm
CiCtsCx					
CiCtsCx75Cm	0.102				0.517
CiCtsCx50Cm	**0.036**	0.995			0.285
CiCtsCx25Cm	**0.015**	0.959	0.998		0.160
CiTsCm	0.903				

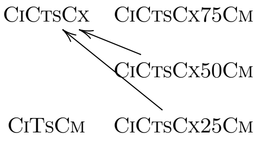


Table 10.5: Median and 95% confidence interval of test-set fitness of the best of run program on training-set (the best in bold).

Problem	CiCtsCx	CiCtsCx75Cm	CiCtsCx50Cm	CiCtsCx25Cm	CiTsCm
R1	0.01 ≤ **0.02** ≤ 0.09	0.01 ≤ 0.06 ≤ 0.23	0.01 ≤ 0.09 ≤ 0.31	0.01 ≤ 0.09 ≤ 0.33	0.02 ≤ 0.04 ≤ 0.18
R2	0.01 ≤ 0.04 ≤ 0.15	0.01 ≤ **0.03** ≤ 0.14	0.01 ≤ **0.03** ≤ 0.06	0.01 ≤ **0.03** ≤ 0.14	0.02 ≤ 0.04 ≤ 0.07
R3	0.01 ≤ 0.02 ≤ 0.08	0.01 ≤ 0.03 ≤ 0.65	0.01 ≤ 0.01 ≤ 0.63	0.01 ≤ 0.02 ≤ 0.63	0.00 ≤ **0.01** ≤ 0.02
Kj1	0.03 ≤ 0.06 ≤ 0.20	0.02 ≤ **0.05** ≤ 0.09	0.02 ≤ 0.05 ≤ 0.45	0.02 ≤ 0.05 ≤ 0.45	0.09 ≤ 0.14 ≤ 0.28
Kj4	1.48 ≤ 1.81 ≤ 2.71	1.03 ≤ **1.76** ≤ 3.26	1.06 ≤ 2.03 ≤ 2.88	1.30 ≤ 2.09 ≤ 3.32	1.05 ≤ 2.19 ≤ 2.67
Ng9	0.01 ≤ 0.02 ≤ 0.03	0.01 ≤ 0.01 ≤ 0.02	0.00 ≤ 0.01 ≤ 0.02	0.01 ≤ 0.01 ≤ 0.02	0.00 ≤ **0.00** ≤ 0.02
Ng12	0.02 ≤ 0.03 ≤ 0.04	0.01 ≤ **0.02** ≤ 0.06	0.02 ≤ 0.03 ≤ 0.08	0.02 ≤ 0.02 ≤ 0.05	0.05 ≤ 0.07 ≤ 0.11
Pg1	3.65 ≤ 5.14 ≤ 6.78	4.78 ≤ 5.39 ≤ 8.40	4.19 ≤ 5.28 ≤ 8.40	4.96 ≤ 5.61 ≤ 8.40	3.84 ≤ **4.72** ≤ 5.41
Vl1	0.47 ≤ 0.98 ≤ 2.44	0.45 ≤ 0.96 ≤ 2.77	0.46 ≤ 0.90 ≤ 2.24	0.52 ≤ **0.86** ≤ 1.96	0.78 ≤ 1.31 ≤ 1.68
Rank:	3.44	2.50	2.89	2.94	3.22

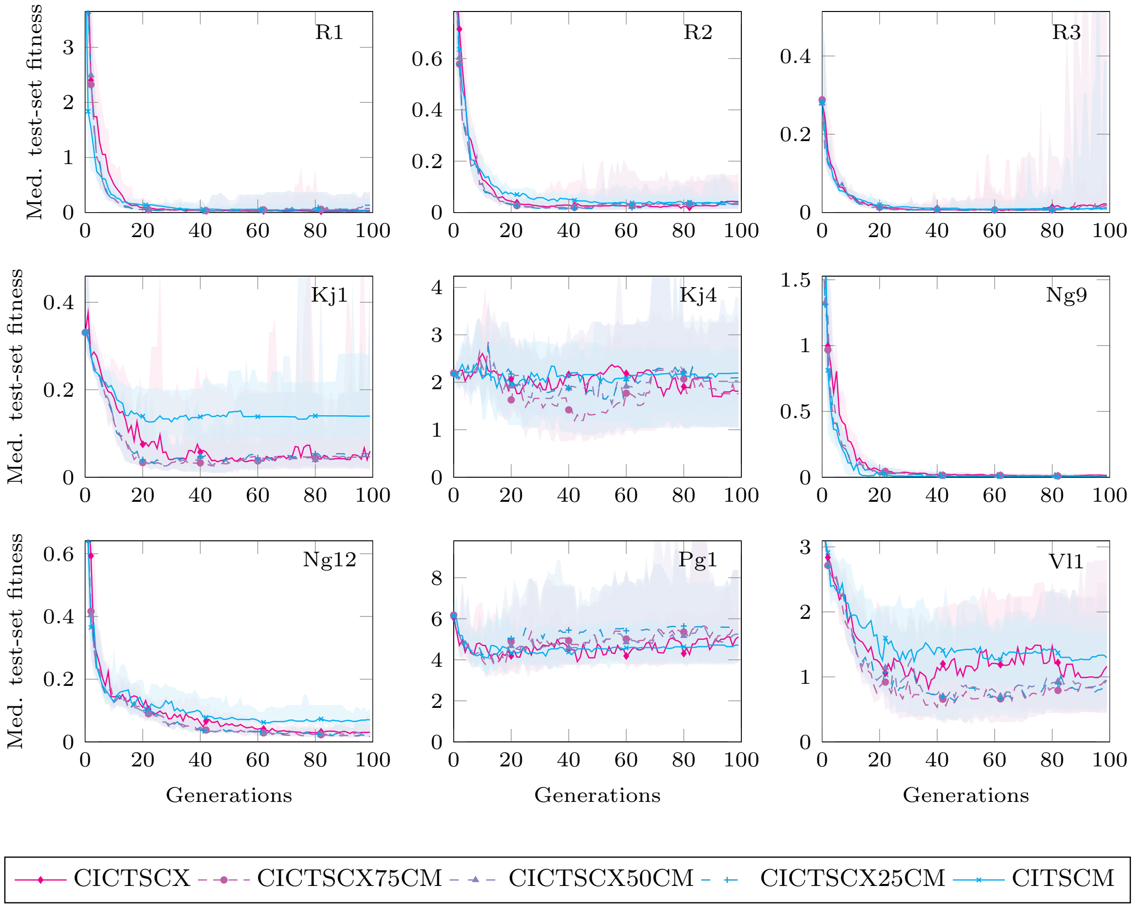


Figure 10.2: Median and 95% confidence interval of test-set fitness of the best of generation program on training-set.

10.2 Generalization abilities

To verify how well the evolved programs generalize, we calculate for each setup the median fitness on test-set of the best of generation program on training set, and present them in Figure 10.2. Table 10.5 presents the same values for the best of run program.

We do not observe noticeable differences between series presented in plots. Although in three problems CiTsCm achieves the worst test-set fitness for almost entire run, in most problems it is difficult to clearly separate series from each other. Concerning the best of run performance, in 4 out of 9 problems CiCtsCx75Cm achieves the best results and CiCtsCx25Cm is runner up with only two best results.

Friedman's test conducted on the median of test-set fitness of the best of run program on training-set results in p-value of 0.701, thus we accept at significance level $\alpha = 0.05$ hypothesis on lack of differences between operators. Thus, we conclude that there is no significant difference between the considered setups, however CiCtsCx75Cm might turn out to be better than the rest given more benchmarks.

Figure 10.2 reveals signs of overfitting to training-set. For all setups in Kj4 and Pg1 problems after $30-50$ generations test-set fitness tends to deteriorate. Note that Figure 10.1 proves that the training-set one improves in an almost monotone way. Similar observation holds for the reference setups in Vl1 problem, but not for the mixed ones. Thus, the mixed setups seem to be slightly less

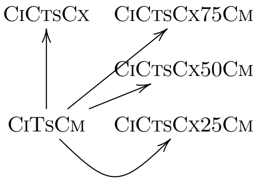
[1] We use similarity threshold, see Table 8.2.

Table 10.6: Average and 95% confidence interval of number of nodes in the best of run program (the lowest in bold).

Problem	CiCtsCx	CiCtsCx75Cm	CiCtsCx50Cm	CiCtsCx25Cm	CiTsCm
R1	569.33 ±68.11	644.40 ±74.68	626.50 ±68.66	643.30 ±70.46	**239.10** ±35.25
R2	654.67 ±82.84	642.77 ±68.05	648.60 ±70.07	650.97 ±71.08	**313.90** ±41.04
R3	664.00 ±77.81	641.30 ±78.91	654.03 ±78.52	642.33 ±77.65	**246.40** ±33.75
Kj1	582.57 ±88.00	568.80 ±113.22	576.17 ±114.18	530.53 ±102.26	**208.10** ±40.65
Kj4	622.27 ±64.47	636.53 ±61.55	633.60 ±65.37	645.40 ±67.45	**228.57** ±27.36
Ng9	569.57 ±108.27	501.53 ±103.86	513.43 ±103.25	473.80 ±102.09	**169.23** ±59.87
Ng12	503.70 ±85.67	494.67 ±64.31	483.53 ±58.49	466.30 ±55.34	**192.53** ±27.24
Pg1	392.13 ±88.47	360.87 ±67.39	367.33 ±77.03	358.63 ±69.44	**135.87** ±32.34
Vl1	463.03 ±45.41	483.23 ±51.19	499.93 ±46.18	501.03 ±42.74	**172.10** ±21.88
Par5	1551.87 ±143.66	1654.60 ±123.68	1711.67 ±116.60	1713.33 ±116.93	**1317.73** ±120.58
Par6	2045.47 ±163.67	2350.20 ±182.88	2330.53 ±187.81	2324.07 ±185.14	**1675.93** ±106.48
Par7	2965.20 ±185.99	3253.13 ±198.58	3335.73 ±202.45	3339.73 ±212.85	**2162.20** ±178.97
Mux6	635.00 ±118.69	808.07 ±101.73	808.07 ±101.73	811.67 ±100.93	**488.20** ±66.06
Mux11	588.60 ±181.63	490.00 ±135.87	510.87 ±138.04	509.07 ±138.69	**378.00** ±63.56
Maj7	1706.40 ±142.99	1696.53 ±143.29	1682.93 ±133.18	1688.47 ±132.54	**1101.73** ±101.66
Maj8	2043.80 ±159.88	1620.80 ±172.47	1608.13 ±174.17	1623.67 ±160.94	**910.80** ±76.69
Cmp6	886.07 ±98.91	858.53 ±89.95	883.87 ±90.14	857.80 ±90.92	**654.27** ±77.71
Cmp8	1476.73 ±134.48	1363.80 ±134.01	1385.53 ±147.53	1415.00 ±147.01	**771.67** ±90.80
Rank:	3.83	3.19	3.47	3.50	1.00

Table 10.7: Post-hoc analysis of Friedman's test conducted on Table 10.6: p-values of incorrectly judging a setup in a row as producing smaller programs than a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	CiCtsCx	CiCtsCx75Cm	CiCtsCx50Cm	CiCtsCx25Cm	CiTsCm
CiCtsCx					
CiCtsCx75Cm	0.743		0.985	0.978	
CiCtsCx50Cm	0.960			1.000	
CiCtsCx25Cm	0.970				
CiTsCm	**0.000**	**0.000**	**0.000**	**0.000**	



prone to overfitting than the reference ones at least in Vl1 problem. Overfitting is not evident for the remaining problems.

10.3 Program size

Figure 10.3 and Table 10.6 present average and 95% of confidence interval of the number of nodes in the best of generation and the best of run program, respectively.

The smallest programs are produced by CiTsCm for all problems. The average size varies from few hundreds up to over two thousand depending on a problem. For this setup, average program size at the end of run is roughly two times greater than in the initial population. CiCtsCx produces noticeably greater programs in 7 out of 18 problems. The mixed setups generate the largest programs with no clear ranking of them. Series that saturate involve runs that found the optimum before 100 generation and, as such, constitute a constant factor in averages of subsequent generations.

Concerning the best of run program size, Friedman's test conducted on average number of nodes in the best of run program is conclusive at the significance level of $\alpha = 0.05$ with p-value of 5.51×10^{-7}. Post-hoc analysis presented in Table 10.7 leaves no doubts that CiTsCm is the least

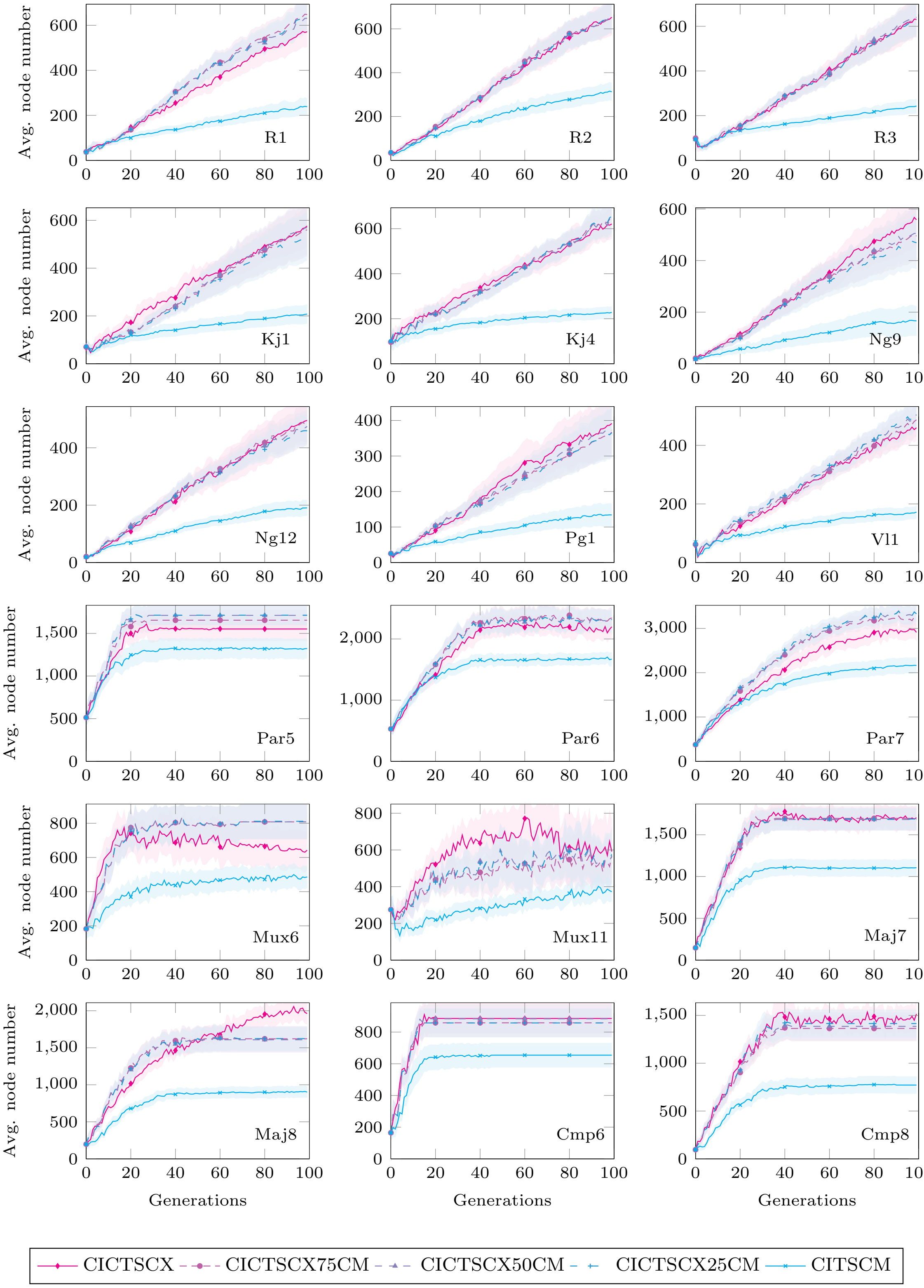


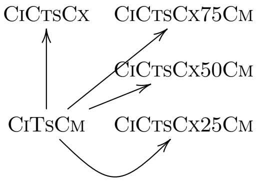
Figure 10.3: Average and 95% confidence interval of numbers of nodes in the best of generation program.

Table 10.8: Average and 95% confidence interval of total execution time spent to finish run (the lowest in bold).

Problem	CiCtsCx		CiCtsCx75Cm		CiCtsCx50Cm		CiCtsCx25Cm		CiTsCm	
R1	549.09	±40.57	595.88	±47.09	586.37	±32.88	576.84	±30.44	**42.60**	±7.64
R2	617.22	±35.49	636.71	±50.40	644.59	±52.43	627.48	±48.05	**50.92**	±7.75
R3	616.08	±36.44	629.78	±43.18	642.97	±42.95	632.96	±42.29	**47.51**	±7.75
Kj1	675.85	±51.20	634.85	±57.73	611.38	±51.78	600.03	±53.95	**45.24**	±6.23
Kj4	671.80	±35.91	678.35	±41.99	677.16	±44.05	653.88	±43.79	**49.24**	±6.99
Ng9	2074.41	±279.46	2397.03	±315.37	2351.62	±369.22	2220.60	±320.12	**394.26**	±111.60
Ng12	2274.75	±181.16	2581.34	±241.31	2498.71	±237.41	2516.86	±220.24	**393.74**	±95.28
Pg1	2426.83	±356.24	2216.00	±375.08	2143.71	±330.77	2262.93	±354.82	**145.16**	±36.25
Vl1	1756.09	±95.06	1866.99	±167.66	1954.77	±162.98	2075.71	±184.67	**165.14**	±42.54
Par5	62.89	±7.09	**53.22**	±4.18	56.38	±4.11	56.11	±3.59	56.77	±13.25
Par6	661.82	±30.88	488.61	±60.45	486.83	±59.13	464.45	±59.83	**300.63**	±20.99
Par7	1224.72	±62.87	1348.50	±58.30	1387.84	±63.33	1400.60	±60.39	**398.49**	±35.84
Mux6	107.99	±26.61	52.73	±9.50	52.41	±9.06	51.65	±8.30	**44.29**	±12.45
Mux11	4522.62	±406.16	3642.13	±402.38	3673.27	±397.12	3712.15	±391.21	**434.58**	±50.90
Maj7	349.82	±70.29	164.03	±10.88	162.39	±12.36	160.25	±11.91	**76.64**	±15.64
Maj8	1079.50	±171.16	455.06	±69.61	442.94	±66.30	437.58	±70.58	**104.68**	±34.70
Cmp6	38.86	±2.32	29.06	±2.37	29.42	±2.24	28.61	±1.98	**17.03**	±3.21
Cmp8	614.00	±124.29	236.63	±25.18	239.73	±24.84	242.67	±25.29	**75.14**	±17.06
Rank:	3.72		3.56		3.50		3.06		1.17	

Table 10.9: Post-hoc analysis of Friedman's test conducted on Table 10.8: p-values of incorrectly judging a setup in a row as faster than a setup in a column. Significant ($\alpha = 0.05$) p-values are in bold and visualized as arcs in graph.

	CiCtsCx	CiCtsCx75Cm	CiCtsCx50Cm	CiCtsCx25Cm	CiTsCm
CiCtsCx					
CiCtsCx75Cm	0.998				
CiCtsCx50Cm	0.993	1.000			
CiCtsCx25Cm	0.713	0.878	0.917		
CiTsCm	**0.000**	**0.000**	**0.000**	**0.003**	



bloating setup, however the remaining setups are indiscernible. Therefore even a small amount of CX added to a setup causes substantial increase in size of the produced programs.

10.4 Computational costs

Figure 10.4 shows average and 95% confidence interval of CPU time consumed by each setup over generations and Table 10.8 presents the same values for the entire run.

Undoubtedly CiTsCm has the lowest time requirements of all considered setups. All the remaining setups in symbolic regression use similar amount of computational resources, however in synthesis of Boolean programs CiCtsCx is the slowest, noticeably slower than the mixed setups. Series that saturate involve runs that found the optimum before 100 generation and do not contribute to the growth of the time.

To assess statistical differences between setups we conduct Friedman's test on the average total execution time. The test results in p-value of 1.07×10^{-5}, thus at significance level $\alpha = 0.05$ there is at least one pair of significantly different setups. The results of post-hoc analysis are shown in Table 10.9. The analysis confirms that CiTsCm is faster than all other setups, and there are no

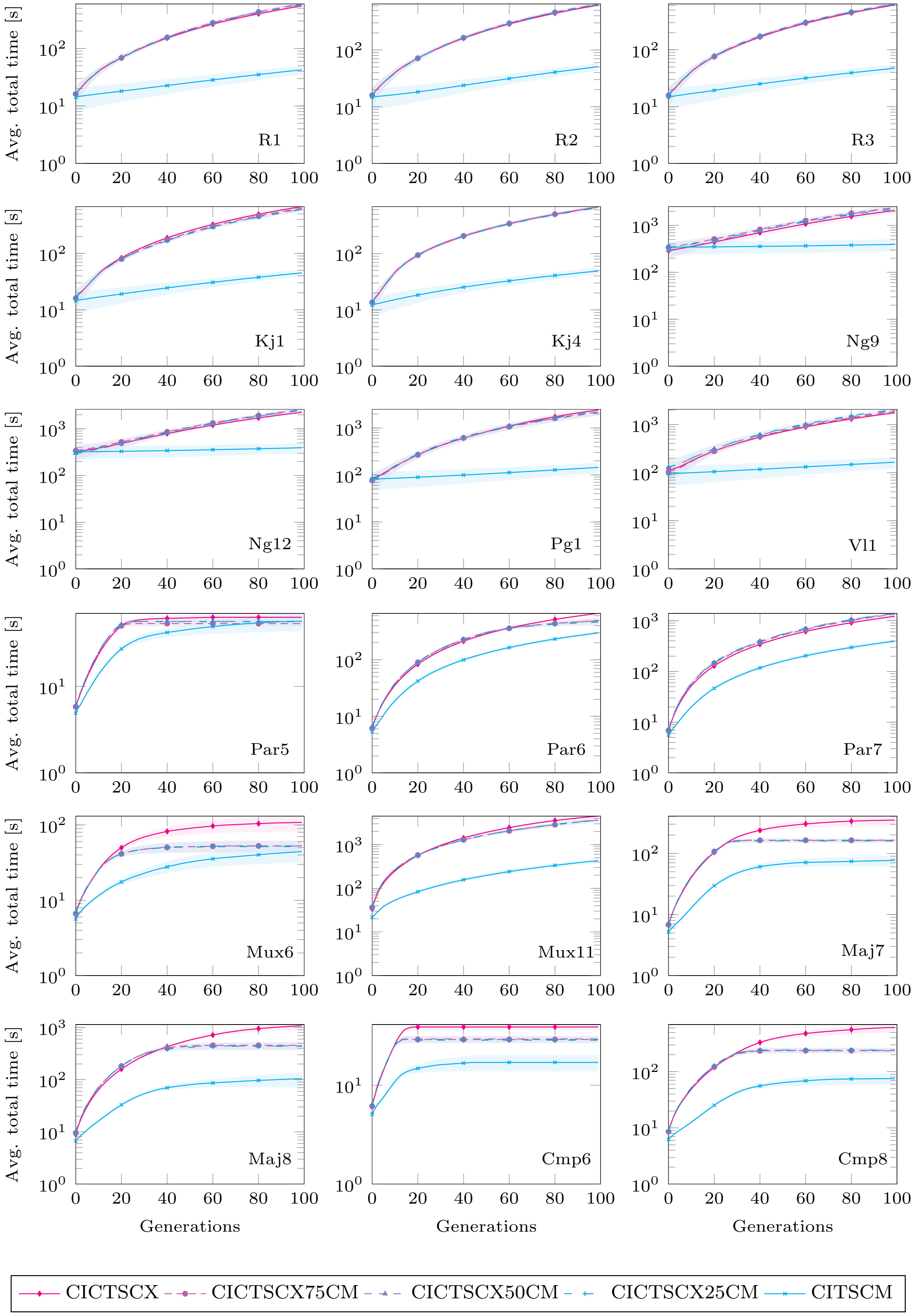


Figure 10.4: Average and 95% confidence interval of total execution time w.r.t. number of generations.

Table 10.10: Spearman's rank correlation coefficient of the number of nodes in produced programs and computational time consumed by a setup. Left: correlation over problems for each setup separately. Right: aggregated rankings of setups from last rows of Tables 10.6 and 10.8, and correlation of them. P-values of t test of significance of correlation, significant correlations ($\alpha = 0.05$) are marked in bold.

	Correlation	P-value	Size rank	Time rank
CiCtsCx	−**0.461**	0.027	3.83	3.72
CiCtsCx75Cm	−**0.701**	0.001	3.19	3.56
CiCtsCx50Cm	−**0.686**	0.001	3.47	3.50
CiCtsCx25Cm	−**0.695**	0.001	3.50	3.06
CiTsCm	0.020	0.469	1.00	1.17
		Correlation:	**0.965**	
		P-value:	0.004	

Table 10.11: Exemplary analysis for time budget of 300 seconds for Vl1 problem. The highest generation and the lowest statistics are in bold.

	Generation	Fitness		Test-set fitness			Program size	
CiCtsCx	21	0.112	±0.037	0.911 ≤	1.141	≤ 2.002	**129.93**	±17.23
CiCtsCx75Cm	21	**0.055**	±0.012	0.764 ≤	**0.898**	≤ 1.621	139.90	±15.61
CiCtsCx50Cm	19	0.067	±0.013	0.706 ≤	0.948	≤ 1.962	136.93	±17.51
CiCtsCx25Cm	19	0.064	±0.014	0.713 ≤	1.005	≤ 1.986	139.43	±18.02
CiTsCm	**100**	0.191	±0.066	0.919 ≤	1.297	≤ 1.680	172.10	±21.88

other significant outrankings. Hence use of any amount of crossover in a setup increases noticeably overall computation time.

The runtime of a GP algorithm is at least in part determined by the size of programs being evolved. To verify whether computational time is correlated with the size, we carry out two analyses. First, *for each setup separately* we calculate Spearman's rank correlation coefficient of the size and the time over problems. The coefficients together with p-values of t test for significance of correlation are presented in the left part of Table 10.10. For each setup that engages crossover we observe moderate to strong negative correlation of size and time, thus the bigger programs the setup produces the less time it takes. Since this observation does not hold for CiTsCm setup which does not employ crossover, we hypothesize that there must a factor inherent to CX that increases the computational time and simultaneously reduces or limits size of produced programs. For instance it may be implicit simplification of program structure that preserves semantics, however the true reason requires further investigation.

In the second analysis, we consider rankings of *all setups* with respect to the size and the time took from Tables 10.6 and 10.8, respectively. Juxtaposition of the rankings reveals strong positive correlation, what is confirmed by t test for significance of correlation, all of them presented in right part of Table 10.10. Therefore setups that tend to produce bigger programs, also tend to work longer.

Finally, we concern the assessment of the considered setups under limited time budget. To simulate this case, we use Vl1 problem and 300 seconds limit of CPU time. The numbers of generations reached within the limit by particular setups and the corresponding average training and median test-set fitness and average program sizes are presented in Table 10.11. Undoubtedly CiTsCm is the quickest setup that reaches 100 generations. Proportions of crossover and mutation seem to have a minor impact on speed. The best training and test-set fitness are achieved by CiCtsCx75Cm and the smallest programs are produced by CiCtsCx, however confidence intervals of all setups that involve CX overlap. This shows that even if CM is faster than CX, using a mixture of them (or even CX alone) may lead to better results than the use of CM solely.

10.5 Conclusions

We compared five setups of competent algorithms with different mixture of CX and CM to answer research question 5 from Section 8.1 on the optimal proportions of CX and CM in an evolutionary setup. In all characteristics except program size and computational time, the mixed setups achieve better results than the reference ones. In terms of program size and time, CiTsCm is the best. Nevertheless we showed that even under a limited time budget the mixed setups achieve better fitness and lower program size than CiTsCm. In most comparisons the mixed setups are statistically indiscernible from each other, however for the most important characteristics, i.e., the average best of run fitness and the probability of solving problems, CiCtsCx25Cm is slightly better than the rest. In conclusion, we recommend to use CX and CM in proportions of 25% : 75%.

In the context of building block hypothesis (cf. Section 2.3.3), crossover is often considered as the key ingredient of evolutionary algorithms due to its capability to exchange large fragments of solutions (here: program fragments) and so performing long-distance leaps in solution space. Our explanation for the low proportion of CX in the recommended setup concerns the design of CM and CX. CM strives to produce a program, which semantics is the target one, i.e., performs a *direct* search. In contrast CX only combines semantics of the given parents, thus performs *indirect* search. It is natural to expect that the direct search of semantic space results in quicker convergence to the optimum than the indirect one. However due to imperfections of the library backing both operators CM may be unable to achieve its goal, thus a support of CX may reveal an other way to build a solution.

Chapter 11

The closing conclusions

11.1 Summary

In this thesis, we presented the problem of program induction from samples of desired behavior, a common way of solving that problem using genetic programming. We also introduced recent variant of semantic genetic programming that involves semantics in program induction process. Within the SGP framework, we defined semantic distance, neutrality and effectiveness of genetic operators. We showed that when fitness function is a metric, fitness landscape is a cone. We also defined geometric genetic operators that utilize this fact and demonstrated their useful properties. We developed a suite of competent operator algorithms for program initialization, parent selection, crossover and mutation that share features of geometric and effective operators (Chapter 7). The competent operators alleviate many drawbacks of exact geometric operators and are characterized by superior performance over the remaining operators compared empirically in this thesis, including the effective ones. We achieved all research goals that we set off with in Section 1.3.

11.2 Original contributions

Below we summarize the most important original contributions of this thesis:

- Introduction of properties of semantic neutrality and effectiveness for all operators typically involved in GP: population initialization, parent selection, mutation and crossover, and verification of these properties for the state of the art operators.
  [Chapter 4]
- Demonstration that if fitness function is a metric, a fitness landscape that spans over space of programs and is proxied by semantics has a conic shape with the target in the apex.
  [Section 5.2]
- Definition of geometric operators for all operators involved in GP and theoretical derivation of their properties under multiple combinations of fitness functions and semantic distance metrics, e.g., bounds and properties of progress.
  [Sections 5.3 – 5.6]
- Demonstration that if parents lie in a sphere centered in the target, geometric $n \geq 2$-ary variation operators produce offspring not worse than the best parent. This is a clear hint to designers of future geometric mate selection operators.
  [Section 5.5, Theorem 5.16]
- Comprehensive survey of methods in semantic genetic programming.
  [Chapter 6]

- Introduction of desired semantics, semantic distance for desired semantics and an algorithm of backpropagation of semantics through program structure.
  [Section 7.4]
- Proposition of a complete suite of *competent algorithms* for operators that embrace population initialization, parent selection, crossover and mutation, all approximately geometric and effective by design. These beneficial properties are reflected in ability of the competent algorithms to solve difficult program induction problems, *quickly* (i.e., the best fitness in the lowest number of generations), *reliably* (low variance of performance), and *accurately* (the highest probability of success; see definition of 'competent' in Section 1.4).
  [Chapter 7]
- Extensive experimental analysis in which 28 different experimental setups were applied to 18 benchmark problems in a total of 15120 experimental runs that took almost 55 days. Operators were assessed individually and in various combinations. The analysis showed that the competent operators are indeed approximately geometric, effective, and less affected by bloating and poor generalization than the exact geometric operators. In terms of fitness and generalization performance, the competent algorithms are second to none across problems and domains.
  [Chapter 9]
- Experimental findings on the optimal proportions of competent crossover and competent mutation with respect to training and test set fitness, program size and computational costs for the Boolean and symbolic regression domains.
  [Chapter 10]
- Software implementation of all the competent operators as well as the state-of-the-art corresponding operators. The implementation is optimized to minimize the runtime and equipped with a module that collects statistics in a hierarchically structured database and offers versatile statistical functionality. The Java source code of the software is available at
  http://www.cs.put.poznan.pl/tpawlak/link/?PhD.
  [Section 8.4]

11.3 Key conclusions

The author finds the following overall conclusions and observations of this work particularly important and/or promising:

- Importance of effectiveness, i.e., non-neutrality. Caring for effectiveness is conceptually straightforward and its computational cost in most cases pays off in improved performance of search.
- Rationale for exploiting the tradeoff between the degree of geometry and program size. The inexact operators may converge slower, but produce reasonably sized programs.
- Importance of maintaining the target semantics in the convex hull of a population during entire evolutionary run using proper initialization, selection and variation operators.
- Impact of selection of fitness function and semantic distance on properties of geometric (e.g., competent) operators and efficiency of search.

11.4 Future work

During the work we identified several areas of research that particularly seem to deserve further investigation. Below we enumerate the most important of them.

- Geometric $n \geq 2$-ary variation operators are strongly progressive only under certain conditions (cf. Theorem 5.16), which is a consequence of construction of convex hull of parents' semantics. However, strong progress can be guaranteed by allowing offspring to be produced only in a certain a part of the convex hull of parents' semantics. This deserves further research to define the appropriate desirable properties of such operators and develop proper algorithms.
- Under certain conditions, a geometric $n \geq 2$-ary variation operator is unable to return offspring with better fitness than the best parent. This can be avoided by replacing convex hull in Definition 5.7 with an affine hull. This extension opens a way to a new class of operators with possibly different properties and characteristics.
- The competent operators produce programs of sizes in between other approximately geometric operators and the exact geometric operators. It is tempting to integrate program simplification procedures into the competent operators. This may increase computational costs, thus an additional analysis is required to determine whether such a modification pays off, i.e., whether the gains resulting from simplified programs evaluating faster are not offset by the time required for simplification.
- Concerning accuracy of the competent operators, although the probabilities of success for them in Sections 9.3.1, 9.4.1 and 10.1 are superior to other operators, they are still below 100%, thus there is a place for improvement and accuracy remains an open issue.
- The negative correlation of size of produced programs and run time of competent crossover (Sections 9.4.5 and 10.4) remains unexplained and requires further investigation.
- The experimental verification conducted in this thesis focused on the Boolean and symbolic regression domains. A natural follow-up is applying the competent operators beyond these domains, e.g., to categorical domain where a task is to induce an algebraic form that fulfills certain constraints, see e.g., [163, 88].

Index

Bibliography

[1] Revised report on the algorithmic language ALGOL 68. *Acta Informatica*, 5(1-3):1–236, 1975.

[2] ISO/IEC 15444-1:2004 Information technology – JPEG 2000 image coding system – Part 1: Core coding system, 2004.

[3] Amazon Elasting Computing Cloud (EC2). https://aws.amazon.com/ec2/, 20 October 2014.

[4] Flex: The fast lexical analyzer. https://flex.sourceforge.net/, 20 October 2014.

[5] Null httpd. http://sourceforge.net/projects/nullhttpd/, 20 October 2014.

[6] PHP: Hypertext Preprocessor. https://php.net/, 20 October 2014.

[7] Python. https://www.python.org/, 20 October 2014.

[8] The Annual "Humies" Awards – 2004 – 2014. http://www.genetic-programming.org/combined.php, 20 October 2014.

[9] Wireshark: Go Deep. https://www.wireshark.org/, 20 October 2014.

[10] Zune — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Zune, 20 October 2014. Discontinued product, no official website.

[11] David Andre and John R. Koza. Parallel genetic programming: A scalable implementation using the transputer network architecture. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 16, pages 317–337. MIT Press, Cambridge, MA, USA, 1996.

[12] David Andre and Astro Teller. A study in program response and the negative effects of introns in genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 12–20, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[13] Peter J. Angeline. Subtree crossover: Building block engine or macromutation? 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 9–17, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[14] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The fortran automatic coding system. In *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability*, IRE-AIEE-ACM '57 (Western), pages 188–198, New York, NY, USA, 1957. ACM.

[15] John W. Backus. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. In *IFIP Congress*, pages 125–131, 1959.

[16] KirbyA. Baker and AldenF. Pixley. Polynomial interpolation and the chinese remainder theorem for algebraic systems. *Mathematische Zeitschrift*, 143(2):165–174, 1975.

[17] Wolfgang Banzhaf. Genetic programming for pedestrians. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, page 628, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.

[18] R. Bartels, S. Backus, E. Zeek, L. Misoguti, G. Vdovin, I. P. Christov, M. M. Murnane, and H. C. Kapteyn. Shaped-pulse optimization of coherent emission of high-harmonic soft X-rays. *Nature*, 406(6792):164–166, July 2000.

[19] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Playing regex golf with genetic programming. In Christian Igel and Dirk V. Arnold et al., editors, *GECCO '14: Proceeding of the sixteenth annual conference on genetic and evolutionary computation conference*, pages 1063–1070, Vancouver, BC, Canada, 12-16 July 2014. ACM.

[20] 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.

[21] 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.

[22] Lawrence Beadle and Colin G Johnson. Semantically driven mutation in genetic programming. In Andy Tyrrell, editor, *2009 IEEE Congress on Evolutionary Computation*, pages 1336–1342, Trondheim, Norway, 18-21 May 2009. IEEE Computational Intelligence Society, IEEE Press.

[23] Andrzej Bielecki and Barbara Strug. An evolutionary algorithm for solving the inverse problem for iterated function systems for a two dimensional image. In Marek Kurzyński, Edward Puchała, Michał Woźniak, and Andrzej żołnierek, editors, *Computer Recognition Systems*, volume 30 of *Advances in Soft Computing*, pages 347–354. Springer Berlin Heidelberg, 2005.

[24] Andrzej Bielecki and Barbara Strug. Finding an iterated function systems based representation for complex visual structures using an evolutionary algorihm. *MG&V*, 16(1):171–189, January 2007.

[25] Alan W. Biermann. The inference of regular LISP programs from examples. *IEEE Transactions on Systems, Man and Cybernetics*, 8(8):585–600, 1978.

[26] Ray Brown. On solving nonlinear functional, finite, difference, composition, and iterated equations. *Fractals*, 7(3):277–282, 1998.

[27] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[28] R. Burden and J. Faires. *Numerical Analysis.* Cengage Learning, 2010.

[29] Edmund Burke, Graham Kendall, Jim Newall, Emma Hart, Peter Ross, and Sonia Schulenburg. Hyper-heuristics: An emerging direction in modern search technology. In Fred Glover and GaryA. Kochenberger, editors, *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science*, pages 457–474. Springer US, 2003.

[30] Constantin Carathéodory. Über den variabilitätsbereich der fourierschen konstanten von positiven harmonischen funktionen. *Rendiconti del Circolo Matematico di Palermo*, 32:193–217, 1911.

[31] Mauro Castelli, Davide Castaldi, Ilaria Giordani, Sara Silva, Leonardo Vanneschi, Francesco Archetti, and Daniele Maccagnola. An efficient implementation of geometric semantic genetic programming for anticoagulation level prediction in pharmacogenetics. In Luis Correia, Luis Paulo Reis, and Jose Cascalho, editors, *Proceedings of the 16th Portuguese Conference on Artificial Intelligence, EPIA 2013*, volume 8154 of *Lecture Notes in Computer Science*, pages 78–89, Angra do Heroismo, Azores, Portugal, September 9-12 2013. Springer.

[32] Mauro Castelli, Davide Castaldi, Leonardo Vanneschi, Ilaria Giordani, Francesco Archetti, and Daniele Maccagnola. An efficient implementation of geometric semantic genetic programming for anticoagulation level prediction in pharmacogenetics. In Christian Blum and Enrique Alba et al., editors, *GECCO '13 Companion: Proceeding of the fifteenth annual conference companion on Genetic and evolutionary computation conference companion*, pages 137–138, Amsterdam, The Netherlands, 6-10 July 2013. ACM.

[33] Mauro Castelli, Leonardo Vanneschi, and Sara Silva. Prediction of high performance concrete strength using genetic programming with geometric semantic genetic operators. *Expert Systems with Applications*, 40(17):6856–6862, 2013.

[34] Mauro Castelli, Leonardo Vanneschi, and Sara Silva. Semantic search-based genetic programming and the effect of intron deletion. *IEEE Transactions on Cybernetics*, 44(1):103–113, January 2014.

[35] Kit Yan Chan and Terence C. Fogarty. Experimental design based multi-parent crossover operator. 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 297–306, Essex, 14-16 April 2003. Springer-Verlag.

[36] D. Clark, B. Davey, J. Pitkethly, and D. Rifqui. Primality from semilattices, 2008. Preprint.

[37] Manuel Clergue, Philippe Collard, Marco Tomassini, and Leonardo Vanneschi. Fitness distance correlation and problem difficulty for genetic programming. In W. B. Langdon and E. Cantú-Paz et al., editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 724–732, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[38] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[39] H. Cramér. *Mathematical Methods of Statistics.* Almqvist & Wiksells Akademiska Handböcker. Princeton University Press, 1946.

[40] Ellery Fussell Crane and Nicholas Freitag McPhee. The effects of size and depth limits on tree based genetic programming. In Tina Yu, Rick L. Riolo, and Bill Worzel, editors, *Genetic Programming Theory and Practice III*, volume 9 of *Genetic Programming*, chapter 15, pages 223–240. Springer, Ann Arbor, 12-14 May 2005.

[41] Raphael Crawford-Marks and Lee Spector. Size control via size fair genetic operators in the PushGP genetic programming system. In W. B. Langdon and E. Cantú-Paz et al., editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 733–739, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[42] Ralph B. D'Agostino. An Omnibus Test of Normality for Moderate and Large Size Samples. *Biometrika*, 58(2):341–348, August 1971.

[43] Bruce Dawson. That's not normal – the performance of odd floats. https://randomascii.wordpress.com/2012/05/20/thats-not-normalthe-performance-of-odd-floats/, 29 March 2015.

[44] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 243–320. 1990.

[45] Richard O. Duda and Peter E. Hart. Use of the hough transformation to detect lines and curves in pictures. *Commun. ACM*, 15(1):11–15, January 1972.

[46] Jeroen Eggermont, Joost N. Kok, and Walter A. Kosters. Detecting and pruning introns for faster decision tree evolution. In Xin Yao, Edmund Burke, Jose A. Lozano, Jim Smith, Juan J. Merelo-Guervós, John A. Bullinaria, Jonathan Rowe, Peter Tiňo Ata Kabán, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VIII*, volume 3242 of *LNCS*, pages 1071–1080, Birmingham, UK, 18-22 September 2004. Springer-Verlag.

[47] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.

[48] A.E. Eiben, P-E. Raué, and Zs. Ruttkay. Genetic algorithms with multi-parent recombination, 1994.

[49] C. Ferreira. Genetic representation and genetic neutrality in gene expression programming. *Advances in Complex Systems*, 5(4):389–408, 2002.

[50] David B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press, Piscataway, NJ, USA, 1995.

[51] D.B. Fogel and J.W. Atmar. Comparing genetic operators with gaussian mutations in simulated evolutionary processes using linear systems. *Biological Cybernetics*, 63(2):111–114, 1990.

[52] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial intelligence through simulated evolution*. Wiley, Chichester, WS, UK, 1966.

[53] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In Guenther Raidl and Franz Rothlauf et al., 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.

[54] Zbysek Gajda and Lukas Sekanina. Gate-level optimization of polymorphic circuits using cartesian genetic programming. In Andy Tyrrell, editor, *2009 IEEE Congress on Evolutionary Computation*, pages 1599–1604, Trondheim, Norway, 18-21 May 2009. IEEE Computational Intelligence Society, IEEE Press.

[55] Edgar Galvan-Lopez, Brendan Cody-Kenny, Leonardo Trujillo, and Ahmed Kattan. Using semantics in the selection mechanism in genetic programming: a simple method for promoting semantic diversity. In Luis Gerardo de la Fraga, editor, *2013 IEEE Conference on Evolutionary Computation*, volume 1, pages 2972–2979, Cancun, Mexico, June 20-23 2013.

[56] Santiago Garcia Carbajal and Fermin Gonzalez Martinez. Evolutive introns: A non-costly method of using introns in GP. *Genetic Programming and Evolvable Machines*, 2(2):111–122, June 2001.

[57] James E. Gentle. *Numerical linear algebra for applications in statistics*. Statistics and computing. Springer, New York, 1998.

[58] Assaf Glazer and Moshe Sipper. Evolving an automatic defect classification tool. In Mario Giacobini, Anthony Brabazon, and Stefano et al. Cagnoni, editors, *Applications of Evolutionary Computing*, volume 4974 of *Lecture Notes in Computer Science*, pages 194–203. Springer Berlin Heidelberg, 2008.

[59] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[60] D.E. Goldberg. *The Design of Innovation: Lessons from and for Competent Genetic Algorithms.* Kluwer Academic Publishers, Boston, MA, 2002.

[61] Uli Grasemann and Risto Miikkulainen. Effective image compression using evolved wavelets. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2005.

[62] F. Gray. Pulse code communication, March 17 1953. US Patent 2,632,058.

[63] R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160, 1950.

[64] Akira Hara, Yoshimasa Ueno, and Tetsuyuki Takahama. New crossover operator based on semantic distance between subtrees in genetic programming. In *IEEE International Conference on Systems, Man, and Cybernetics (SMC 2012)*, pages 721–726, Seoul, Korea, October 14-17 2012.

[65] John H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI, 1975. second edition, 1992.

[66] John Henry Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* Bradford Books. MIT Press, 1992.

[67] Gregory. S. Hornby, Jason D. Lohn, and Derek S. Linden. Computer-automated evolution of an X-band antenna for NASA's space technology 5 mission. *Evolutionary Computation*, 19(1):1–23, Spring 2011.

[68] Torsten Hothorn, Kurt Hornik, Mark A. van de Wiel, and Achim Zeileis. Package 'coin': Conditional inference procedures in a permutation test framework.

[69] Paul V.C. Hough. Method and means for recognizing complex patterns, December 18 1962. US Patent 3,069,654.

[70] Daniel Howard, Simon C. Roberts, and Richard Brankin. Target detection in SAR imagery by genetic programming. *Advances in Engineering Software*, 30(5):303–311, May 1999.

[71] Mal'cev I.A. On general theory of algebraic systems (russian). *Matematicheskii Sbornik*, (35):3–22, 1954.

[72] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic.* August 2008.

[73] Muhammad Irfan, Qaiser Habib, Ghulam M. Hassan, Khawaja M. Yahya, and Samira Hayat. Combinational digital circuit synthesis using cartesian genetic programming from a NAND gate template. In *6th International Conference on Emerging Technologies (ICET 2010)*, pages 343–347, October 2010.

[74] David Jackson. Phenotypic diversity in initial genetic programming populations. In Anna Isabel Esparcia-Alcazar, Aniko Ekart, Sara Silva, Stephen Dignum, and A. Sima Uyar, editors, *Proceedings of the 13th European Conference on Genetic Programming, EuroGP 2010*, volume 6021 of *LNCS*, pages 98–109, Istanbul, 7-9 April 2010. Springer.

[75] 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.

[76] Anil K. Jain and David Maltoni. *Handbook of Fingerprint Recognition*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[77] J.L.W.V. Jensen. Sur les fonctions convexes et les inégalités entre les valeurs moyennes. *Acta Mathematica*, 30(1):175–193, 1906.

[78] G.K. Kanji. *100 Statistical Tests*. SAGE Publications, 1999.

[79] Robert E. Keller and Wolfgang Banzhaf. Genetic programming using genotype-phenotype mapping from linear genomes into linear phenotypes. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 116–122, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

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

[81] John R. Koza. Scalable learning in genetic programming using automatic function definition. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 6, pages 99–117. MIT Press, Cambridge, MA, USA, 1994.

[82] John R. Koza and Forrest H Bennett III. Automatic synthesis, placement, and routing of electrical circuits by means of genetic programming. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 6, pages 105–134. MIT Press, Cambridge, MA, USA, June 1999.

[83] Krzysztof Krawiec and Pawel Lichocki. Approximating geometric crossover in semantic space. In Guenther Raidl and Franz Rothlauf et al., editors, *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 987–994, Montreal, 8-12 July 2009. ACM.

[84] Krzysztof Krawiec and Tomasz Pawlak. Locally geometric semantic crossover. In Terry Soule and Anne Auger et al., editors, *GECCO Companion '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, pages 1487–1488, Philadelphia, Pennsylvania, USA, 7-11 July 2012. ACM.

[85] Krzysztof Krawiec and Tomasz Pawlak. Quantitative analysis of locally geometric semantic crossover. In Carlos A. Coello Coello, Vincenzo Cutello, Kalyanmoy Deb, Stephanie Forrest, Giuseppe Nicosia, and Mario Pavone, editors, *Parallel Problem Solving from Nature - PPSN XII*, volume 7491 of *Lecture Notes in Computer Science*, pages 397–406, Taormina, Italy, September 1-5 2012. Springer.

[86] Krzysztof Krawiec and Tomasz Pawlak. Approximating geometric crossover by semantic backpropagation. In Christian Blum and Enrique Alba et al., editors, *GECCO '13: Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 941–948, Amsterdam, The Netherlands, 6-10 July 2013. ACM.

[87] Krzysztof Krawiec and Tomasz Pawlak. Locally geometric semantic crossover: a study on the roles of semantics and homology in recombination operators. *Genetic Programming and Evolvable Machines*, 14(1):31–63, March 2013.

[88] Krzysztof Krawiec and Armando Solar-Lezama. Improving genetic programming with behavioral consistency measure. In Thomas Bartz-Beielstein, Juergen Branke, Bogdan Filipic, and Jim Smith, editors, *13th International Conference on Parallel Problem Solving from Nature*, volume 8672 of *Lecture Notes in Computer Science*, pages 434–443, Ljubljana, Slovenia, 13-17 September 2014. Springer.

[89] W. B. Langdon. The evolution of size in variable length representations. In *1998 IEEE International Conference on Evolutionary Computation*, pages 633–638, Anchorage, Alaska, USA, 5-9 May 1998. IEEE Press.

[90] W. B. Langdon. How many good programs are there? How long are they? In Kenneth A. De Jong, Riccardo Poli, and Jonathan E. Rowe, editors, *Foundations of Genetic Algorithms VII*, pages 183–202, Torremolinos, Spain, 4-6 September 2002. Morgan Kaufmann. Published 2003.

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

[92] William B. Langdon. The Genetic Programming Bibliography. http://www.cs.bham.ac.uk/~wbl/biblio/gp-bibliography.html, 22 July 2014.

[93] William B. Langdon, Terry Soule, Riccardo Poli, and James A. Foster. The evolution of size and shape. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press, Cambridge, MA, USA, June 1999.

[94] Pedro Larraanaga and Jose A. Lozano. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation.* Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[95] Orion Lawlor, Hari Govind, Isaac Dooley, Michael Breitenfeld, and Laxmikant Kale. Performance degradation in the presence of subnormal floating-point values. In *in Proceedings of the International Workshop on Operating System Interference in High Performance Applications*, 2005.

[96] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In Martin Glinz, editor, *34th International Conference on Software Engineering (ICSE 2012)*, pages 3–13, Zurich, June 2-9 2012.

[97] Leonid A Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973.

[98] Rui Li, MichaelT.M. Emmerich, Jeroen Eggermont, ErnstG.P. Bovenkamp, Thomas Bäck, Jouke Dijkstra, and JohanH.C. Reiber. Optimizing a medical image analysis system using mixed-integer evolution strategies. In Stefano Cagnoni, editor, *Evolutionary Image Analysis and Signal Processing*, volume 213 of *Studies in Computational Intelligence*, pages 91–112. Springer Berlin Heidelberg, 2009.

[99] 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.

[100] Jason D. Lohn, Gregory S. Hornby, and Derek S. Linden. Rapid re-evolution of an X-band antenna for NASA's space technology 5 mission. In Tina Yu, Rick L. Riolo, and Bill Worzel, editors, *Genetic Programming Theory and Practice III*, volume 9 of *Genetic Programming*, chapter 5, pages 65–78. Springer, Ann Arbor, 12-14 May 2005.

[101] 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.

[102] Sean Luke and Lee Spector. A comparison of crossover and mutation in genetic programming. 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 240–248, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[103] Sean Luke and Lee Spector. A revised comparison of crossover and mutation in genetic programming. In John R. Koza and Wolfgang Banzhaf et al., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 208–213, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[104] Steven Manos and Peter J. Bentley. Evolving microstructured optical fibres. In Tina Yu, David Davis, Cem Baydar, and Rajkumar Roy, editors, *Evolutionary Computation in Practice*, volume 88 of *Studies in Computational Intelligence*, chapter 5, pages 87–124. Springer, 2008.

[105] Steven Manos, Maryanne Large, and Leon Poladian. Evolutionary design of single-mode microstructured polymer optical fibres using an artificial embryogeny representation. In Dirk Thierens, editor, *GECCO (Companion)*, pages 2549–2556. ACM, 2007.

[106] 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 and Anne Auger et al., 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.

[107] Nicholas Freitag McPhee, Alex Jarvis, and Ellery Fussell Crane. On the strength of size limits in linear genetic programming. In Kalyanmoy Deb and Riccardo Poli et al., editors, *Genetic and Evolutionary Computation – GECCO-2004, Part II*, volume 3103 of *Lecture Notes in Computer Science*, pages 593–604, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[108] 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.

[109] Brad L. Miller, Brad L. Miller, David E. Goldberg, and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1995.

[110] Geoffrey F. Miller. Exploiting mate choice in evolutionary computation: Sexual selection as a process of search, optimization, and diversification. In Terence C. Fogarty, editor, *Evolutionary Computing*, volume 865 of *Lecture Notes in Computer Science*, pages 65–79. Springer Berlin Heidelberg, 1994.

[111] 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.

[112] Julian F. Miller and Stephen L. Smith. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 10(2):167–174, April 2006.

[113] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[114] Alberto Moraglio. Abstract convex evolutionary search. In Hans-Georg Beyer and W. B. Langdon, editors, *Foundations of Genetic Algorithms*, pages 151–162, Schwarzenberg, Austria, 5-9 January 2011. ACM.

[115] 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.

[116] Alberto Moraglio and Andrea Mambrini. Runtime analysis of mutation-based geometric semantic genetic programming for basis functions regression. In Christian Blum and Enrique Alba et al., editors, *GECCO '13: Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 989–996, Amsterdam, The Netherlands, 6-10 July 2013. ACM.

[117] Alberto Moraglio and Riccardo Poli. Geometric landscape of homologous crossover for syntactic trees. In *Proceedings of the 2005 IEEE Congress on Evolutionary Computation (CEC-2005)*, volume 1, pages 427–434, Edinburgh, 2-4 September 2005. IEEE.

[118] Alberto Moraglio, Riccardo Poli, and Rolv Seehuus. Geometric crossover for biological sequences. In Pierre Collet, Marco Tomassini, Marc Ebner, Steven Gustafson, and Anikó Ekárt, editors, *Proceedings of the 9th European Conference on Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 121–132, Budapest, Hungary, 10 - 12 April 2006. Springer.

[119] Alberto Moraglio and Dirk Sudholt. Runtime analysis of convex evolutionary search. In Terence Soule and Jason H. Moore, editors, *GECCO*, pages 649–656. ACM, 2012.

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

[121] J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[122] 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.

[123] 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.

[124] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction.* John Wiley & Sons, Inc., New York, NY, USA, 1992.

[125] Peter Nordin. A compiling genetic programming system that directly manipulates the machine code. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press, 1994.

[126] Peter Nordin, Frank Francone, and Wolfgang Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 6, pages 111–134. MIT Press, Cambridge, MA, USA, 1996.

[127] Radovan Ondas, Martin Pelikan, and Kumara Sastry. Genetic programming, probabilistic incremental program evolution, and scalability. In Joshua Knowles, editor, *WSC10: 10th Online World Conference on Soft Computing in Industrial Applications*, pages 363–372, On the World Wide Web, 19 September - 7 October 2005.

[128] Michael O'Neill and Conor Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*, volume 4 of *Genetic programming*. Kluwer Academic Publishers, 2003.

[129] Michael O'Neill, Conor Ryan, Maarten Keijzer, and Mike Cattolico. Crossover in grammatical evolution. *Genetic Programming and Evolvable Machines*, 4(1):67–93, March 2003.

[130] Tomasz Pawlak. Automated design of semantically smooth instructions spaces for genetic programming. Master's thesis, Poznań University of Technology, Faculty of Computing Science, Institute of Computing Science, 2011.

[131] Tomasz Pawlak. Combining semantically-effective and geometric crossover operators for genetic programming. In Thomas Bartz-Beielstein, Juergen Branke, Bogdan Filipic, and Jim Smith, editors, *13th International Conference on Parallel Problem Solving from Nature*, volume 8672 of *Lecture Notes in Computer Science*, pages 454–464, Ljubljana, Slovenia, 13-17 September 2014. Springer.

[132] Tomasz P. Pawlak and Krzysztof Krawiec. Guarantees of progress for geometric semantic genetic programming. In Colin Johnson, Krzysztof Krawiec, Alberto Moraglio, and Michael O'Neill, editors, *Semantic Methods in Genetic Programming*, Ljubljana, Slovenia, 13 September 2014. Workshop at Parallel Problem Solving from Nature 2014 conference.

[133] Tomasz P. Pawlak, Bartosz Wieloch, and Krzysztof Krawiec. Semantic backpropagation for designing search operators in genetic programming. *IEEE Transactions on Evolutionary Computation*.

[134] Tomasz P. Pawlak, Bartosz Wieloch, and Krzysztof Krawiec. Review and comparative analysis of geometric semantic crossovers. *Genetic Programming and Evolvable Machines*, 2015.

[135] K. Pearson. *"Note on Regression and Inheritance in the Case of Two Parents"*. 1895.

[136] Tim Perkis. Stack-based genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 148–153, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[137] A.F. Pixley. *Distributivity and Permutability of Congruence Relations in Equational Classes of Algebras*. American Mathematical Society, 1963.

[138] Gordon D. Plotkin. The origins of structural operational semantics. In *Journal of Logic and Algebraic Programming*, pages 60–61, 2004.

[139] Riccardo Poli. A simple but theoretically-motivated method to control bloat in genetic programming. 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 204–217, Essex, 14-16 April 2003. Springer-Verlag.

[140] 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.

[141] 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).

[142] Riccardo Poli and Nicholas Freitag McPhee. A linear estimation-of-distribution GP system. In Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, Anna Isabel Esparcia Alcazar, Ivanoe De Falco, Antonio Della Cioppa, and Ernesto Tarantino, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 206–217, Naples, 26-28 March 2008. Springer.

[143] I. Rechenberg. *Evolutionsstrategie : Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Number 15 in Problemata. Frommann-Holzboog, Stuttgart-Bad Cannstatt, 1973.

[144] Simon C. Roberts, Daniel Howard, and John R. Koza. Evolving modules in genetic programming by subtree encapsulation. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 160–175, Lake Como, Italy, 18-20 April 2001. Springer-Verlag.

[145] Alan Robinson. Genetic programming: Theory, implementation, and the evolution of unconstrained solutions. Division iii thesis, Hampshire College, May 2001.

[146] Leonard J. Rogers. An extension of a certain theorem in inequalities. *Messenger of Mathematics*, 17:145–150, 1888.

[147] Peter Ross. Hyper-heuristics. In EdmundK. Burke and Graham Kendall, editors, *Search Methodologies*, pages 529–556. Springer US, 2005.

[148] Franz Rothlauf. *Representations for genetic and evolutionary algorithms.* Springer, pub-SV:adr, second edition, 2006. First published 2002, 2nd edition available electronically.

[149] Stefano Ruberto, Leonardo Vanneschi, Mauro Castelli, and Sara Silva. ESAGP – A semantic GP framework based on alignment in the error space. In Miguel Nicolau, Krzysztof Krawiec, Malcolm I. Heywood, Mauro Castelli, Pablo Garcia-Sanchez, Juan J. Merelo, Victor M. Rivas Santos, and Kevin Sim, editors, *17th European Conference on Genetic Programming*, volume 8599 of *LNCS*, pages 150–161, Granada, Spain, 23-25 April 2014. Springer.

[150] Günter Rudolph. On correlated mutations in evolution strategies. In Reinhard Männer and Bernard Manderick, editors, *PPSN*, pages 107–116. Elsevier, 1992.

[151] Carl Runge. Über empirische funktionen und die interpolation zwischen äquidistanten ordinaten. *Zeitschrift für Mathematik und Physik*, (46):224–243.

[152] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[153] Conor Ryan, J. J. Collins, and Michael O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. 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 83–96, Paris, 14-15 April 1998. Springer-Verlag.

[154] R. P. Salustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):123–141, 1997.

[155] Kumara Sastry and David E. Goldberg. Probabilistic model building and competent genetic programming. In Rick L. Riolo and Bill Worzel, editors, *Genetic Programming Theory and Practice*, chapter 13, pages 205–220. Kluwer, 2003.

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

[157] Michael D. Schmidt and Hod Lipson. Solving iterated functions using genetic programming. In Anna I. Esparcia and Ying ping Chen et al., editors, *GECCO-2009 Late-Breaking Papers*, pages 2149–2154, Montreal, 8-12 July 2009. ACM.

[158] Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. Programming Research Group Technical Monograph PRG-6, Oxford Univ. Computing Lab., 1971.

[159] Lukas Sekanina, James Alfred Walker, Paul Kaufmann, and Marco Platzner. Evolution of electronic circuits. In Julian F. Miller, editor, *Cartesian Genetic Programming*, Natural Computing Series, chapter 5, pages 125–179. Springer, 2011.

[160] Terence Soule and James A. Foster. Removal bias: a new cause of code growth in tree based evolutionary programming. In *1998 IEEE International Conference on Evolutionary Computation*, pages 781–786, Anchorage, Alaska, USA, 5-9 May 1998. IEEE Press.

[161] C. Spearman. The proof and measurement of association between two things. *American Journal of Psychology*, 15:88–103, 1904.

[162] 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.

[163] Lee Spector, David M. Clark, Ian Lindsay, Bradford Barr, and Jon Klein. Genetic programming for finite algebras. In Maarten Keijzer and Giuliano Antoniol et al., 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.

[164] Lee Spector, Jon Klein, and Maarten Keijzer. The push3 execution stack and the evolution of control. In Hans-Georg Beyer and Una-May O'Reilly et al., editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1689–1696, Washington DC, USA, 25-29 June 2005. ACM Press.

[165] Lee Spector, Chris Perry, and Jon Klein. Push 2.0 programming language description. Technical report, School of Cognitive Science, Hampshire College, April 2004.

[166] Lee Spector, Chris Perry, Jon Klein, and Maarten Keijzer. Push 3.0 programming language description. Technical Report HC-CSTR-2004-02, School of Cognitive Science, Hampshire College, USA, 10 September 2004.

[167] Lee Spector and Alan Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, March 2002.

[168] Matej Sprogar. A study of GP's division operators for symbolic regression. In *Seventh International Conference on Machine Learning and Applications, ICMLA '08*, pages 286–291, La Jolla, San Diego, USA, 11-13 December 2008. IEEE.

[169] J. Michael Steele. *The Cauchy-Schwarz Master Class: An Introduction to the Art of Mathematical Inequalities.* Cambridge University Press, New York, NY, USA, 2004.

[170] Angus Stevenson, editor. *Oxford Dictionary of English.* Oxford University Press, Third edition, 2010.

[171] Phillip D. Summers. A methodology for lisp program construction from examples. *J. ACM*, 24(1):161–175, January 1977.

[172] Walter Alden Tackett and Aviram Carmi. The unique implications of brood selection for genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 160–165, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[173] Brook Taylor. *Methodus incrementorum directa et inversa.* Impensis Gulielmi Innys, 1717.

[174] Shigeyoshi Tsutsui, Masayuki Yamamura, and Takahide Higuchi. Multi-parent recombination with simplex crossover in real coded genetic algorithms. 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 1, pages 657–664, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[175] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[176] Andrew Turner and Julian Miller. Recurrent cartesian genetic programming. In Thomas Bartz-Beielstein, Juergen Branke, Bogdan Filipic, and Jim Smith, editors, *13th International Conference on Parallel Problem Solving from Nature*, volume 8672 of *Lecture Notes in Computer Science*, pages 476–486, Ljubljana, Slovenia, 13-17 September 2014. Springer.

[177] Nguyen Quang Uy, Nguyen Xuan Hoai, Michael O'Neill, Bob McKay, and Edgar Galvan-Lopez. An analysis of semantic aware crossover. In Zhihua Cai, Zhenhua Li, Zhuo Kang, and Yong Liu, editors, *Proceedings of the International Symposium on Intelligent Computation and Applications*, volume 51 of *Communications in Computer and Information Science*, pages 56–65. Springer, 2009.

[178] 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.

[179] Nguyen Quang Uy, Nguyen Xuan Hoai, Michael O'Neill, R. I. McKay, and Dao Ngoc Phong. On the roles of semantic locality of crossover in genetic programming. *Information Sciences*, 235:195–213, 20 June 2013.

[180] Leonardo Vanneschi, Mauro Castelli, and Sara Silva. A survey of semantic methods in genetic programming. *Genetic Programming and Evolvable Machines*, 15(2):195–214, June 2014.

[181] Leonardo Vanneschi, Sara Silva, Mauro Castelli, and Luca Manzoni. Geometric semantic genetic programming for real life applications. In Rick Riolo, Jason H. Moore, and Mark Kotanchek, editors, *Genetic Programming Theory and Practice XI*, Genetic and Evolutionary Computation, chapter 11, pages 191–209. Springer, Ann Arbor, USA, 9-11 May 2013.

[182] Heinrich Werner. Eine charakterisierung funktional vollständiger algebren. *Archiv der Mathematik*, 21(1):381–385, 1970.

[183] D. Whitley, J. Kauth, and Colorado State University. Department of Computer Science. *GENITOR: A Different Genetic Algorithm.* Technical report (Colorado State University. Department of Computer Science). Colorado State University, Department of Computer Science, 1988.

[184] 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.

[185] Bartosz Wieloch. *Semantic Extensions for Genetic Programming.* PhD thesis, Poznan University of Technology, 2013.

[186] Bartosz Wieloch and Krzysztof Krawiec. Running programs backwards: instruction inversion for effective search in semantic spaces. In Christian Blum and Enrique Alba et al., editors, *GECCO '13: Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 1013–1020, Amsterdam, The Netherlands, 6-10 July 2013. ACM.

[187] H.P. Williams. *Model Building in Mathematical Programming.* Wiley, 2013.

[188] Sewall Wright. The roles of mutation, inbreeding, crossbreeding and selection in evolution. *Proceedings of the Sixth International Congress of Genetics*, 1:356–66, 1932.

[189] L. Xu, E. Oja, and P. Kultanen. A new curve detection method: Randomized hough transform (rht). *Pattern Recogn. Lett.*, 11(5):331–338, May 1990.

[190] Kohsuke Yanai and Hitoshi Iba. Estimation of distribution programming based on Bayesian network. 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 1618–1625, Canberra, 8-12 December 2003. IEEE Press.

[191] Jie Yao, Nawwaf Kharma, and Peter Grogono. A multi-population genetic algorithm for robust and fast ellipse detection. *Pattern Analysis and Application*, pages 149–162, 2005.

[192] Byoung-Tak Zhang and Heinz Mühlenbein. Balancing accuracy and parsimony in genetic programming. *Evolutionary Computation*, 3(1):17–38, 1995.

[193] Zhechen Zhu, Asoke K. Nandi, and Muhammad Waqar Aslam. Adapted geometric semantic genetic programming for diabetes and breast cancer classification. In *IEEE International Workshop on Machine Learning for Signal Processing (MLSP 2013)*, September 2013.