Poznan University of Technology
Faculty of Computing Science
Institute of Computing Science

Master's thesis

AUTOMATED DESIGN OF SEMANTICALLY SMOOTH INSTRUCTION SPACES FOR GENETIC PROGRAMMING

Tomasz Pawlak

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

Poznań 2011

Tutaj przychodzi karta pracy dyplomowej;
oryginał wstawiamy do wersji dla archiwum PP, w pozostałych kopiach wstawiamy ksero.

Contents

“In the discrete world of computing, there is no meaningful metric in which ‘small’ changes and ‘small’ effects go hand in hand, and there never will be.”

Edsger Wybe Dijkstra [19]

Chapter 1

Introduction

This thesis concerns the design, implementation and evaluation of the concept of automatic design of semantically smooth instruction spaces for Genetic Programming. The work is focused on improving of Genetic Programming performance by providing sets of instructions that make more smooth transitions between one program instance to another one in terms of program semantics. On a general level the research problem is to invent generic methodology for creation of sets of abstract instructions that help the program search process to converge in global optimum of fitness landscape.

The improvement of search performance is achieved by embedding of original program space into new space, where coordinates of programs are optimized with regard to their semantic similarity. Unfortunately the number of programs, that can be created from given set of instructions is potentially unlimited, therefore it is proposed to split up the program into group of metainstructions – a program building blocks and optimize limited space of the blocks instead. The metainstruction space must be shaped somehow, therefore four methods are proposed for calculation of space dimensions. The analysis of all these methods as well as impact on the search performance caused by the space embedding is verified on the basis of Genetic Programming and compared with its canonical form.

1.1 Motivation

From the time of modern computer invention, the term 'computing' refers to two related technologies, automated calculation and programmability of the calculations. It is not irrational to ask, why not move the computing to a higher level of abstraction – automated programmability of the computer. The task of programming is clearly an intelligent approach aimed at specific goal, which is a program performing a certain job. A software developer, who knows the semantics of instructions and rules of the programming language can write a program with no troubles. As this task is time-consuming, therefore it is tempting to pass on part of the job to the automatic process. However, while human intelligence works by bringing facts together, the artificial one involves massive search trials. These search trials operate with varying performance, depending on difficulty of the problem being solved. Therefore any adjustment that improves the search performance is meaningful.

The main headache of searching through program spaces is connected with quite accidental relation between program code and its outputs. While traditional approaches build programs by operating exclusively in the syntactic space of given language, it is possible to make use of the space of program semantics. The knowledge on the semantics of program or even its fragments makes it potentially helpful in manipulation of program code in more foreseeable way. There is a hope, that this predictability leads program search process to faster finding of target solution.

1.2 Aim of the thesis

The thesis is aimed at development of methodology for design of semantically smooth instructions spaces for Genetic Programming. Thanks to the smoothness of the set, it is expected to achieve improved performance of search for desired program. The main objectives are: development of the methodology of automated construction of instruction sets, in particular design of instruction form, a way to effectively filter useless instructions from set of developed routines, a method for giving a shape to the set and a approach for smoothing of the instruction space, such that the Genetic Programming will perceive noticeably better correlation between program semantics and its location in the search space. In order to achieve the main goal, the auxiliary objectives must be done. They consist of preparation of software project for experimental framework, its implementation and use for tests. The last task will be focused on analysis and interpretation of obtained results. The analysis will cover the scope of entire methodology, from designing of the instruction set, through smoothing procedure, to behavior of the Genetic Programming algorithm.

The project described in this thesis lasted from February 2011 to June 2011 and it was split into five single-month phases. The first phase was devoted to analysis of learnable embeddings of program spaces concept as well as review of literature. The next phase was dedicated to the development of metainstruction and program concept. The third one was spent on implementation of algorithms and next one on computational experiments. The last phase was dedicated to the final review.

1.3 Organization of the thesis

The thesis is organized as follows. Chapter 2 presents an overview of Genetic Programming, its concept, current state of the art and common applications. Next, in Chapter 3, the major challenges for Genetic Programming and searching through program spaces in general are described. Chapter 4 briefly introduces to concept of learnable embeddings of program spaces, metainstruction definition and successive phases of automatic design of instruction sets for Genetic Programming. Chapter 5 analyzes results of computational experiment, the impact of program space embedding and comparison to performance of traditional Genetic Programming approach. Finally, Chapter 6 concludes the thesis and points the way for future work.

Chapter 2

Genetic Programming Overview

2.1 Introduction to Genetic Programming

Evolutionary Computation (EC) is biologically inspired approach to solve optimization problems. The assumptions of EC date back to discoveries of Charles Darwin in the 19th century about means of natural selection and survival of the fittest [17]. Darwin reasoned that immutable species would become progressively incompatible with their changing environment. He noticed similarity of children to their parents, which lead him to conclusion that traits pass from one generation to the next. He also observed little differences between siblings, which provide species with pools of unique individuals competing for food and place to live. From the above observations Darwin concluded that if environment changes, organisms best adapted to the new situation would bring forth offspring reflecting their successful traits. The process was named *natural selection* and Darwin believed that it is the most significant mechanism in evolution of species. He did not know the way characteristics are inherited from parent to children, he simply saw it happens.

The missing part was beyond the reach of science for about century after his death. In 1952 Hershey and Chase firstly confirmed that deoxyribonucleic acid (DNA) takes part in heredity [1]. A year after, in 1953, two biologists, Crick and Watson, published the first correct double-helix model of DNA structure [20, 21]. The DNA is essential in inheritance process because it encodes the chemical recipes for life's proteins, enzymes and, as a result, individual traits. Moreover it packs huge amount of data into very small space.

With analogy to the DNA as medium of information about certain organism, the Evolutionary Computation, also known as Evolutionary Algorithm (EA), involves digital information, which fully describes particular solution. In EA, each solution is called an *individual* and a group of individuals is named a *population*. Since the procedure works by evolving population of solutions, it belongs to class of population-based algorithms. The default schema [27, 30] for Evolutionary Computation consists of initialization and generation stages. In the first stage the initial population is randomly generated from scratch or by seeding using domain knowledge. Generations are repeated until a stopping condition is satisfied. Each generation can be split up into four phases: evaluation, selection, reproduction and mutation. The evaluation phase is aimed at scoring each individual for solving the particular problem of consideration. The score, often called *fitness*, is utilized in the subsequent *selection* phase to compare individuals. In the selection phase, the best suited individuals are chosen and then recombined in the reproduction phase. The crossover

operator responsible for recombination, with analogy to biological reproduction, is a procedure that creates a child by combining parts from parents' genomes. The intuition behind the process is to create a new individual that is a 'mixture' of its parents. The last phase is mutation, usually applied to a small percent of entire population. This phase is not always necessary but since crossover can bring the population to convergence in a single point of the search space, mutation is typically added to the process to maintain diversity by random changes in genome [57]. The pool of children from the current generation become parents in the next generation. The stopping condition predominantly used in practice are findind the global optimum or reaching limit of the number of generations.

The above description of EC is a bit general. The reason is a vast variety of evolutionary algorithms. They differ in representation of genotype, construction of genetic operators, the way the operators are applied to the population, etc. The most widely known classes of evolution algorithms are Genetic Algorithms [30], Evolution Strategies [78], Differential Evolution[86, 87], Evolutionary Programming [22] and Genetic Programming [42, 43, 77]. This thesis focuses on the last one – Genetic Programming.

Genetic Programming (GP) is a variant of Evolutionary Algorithm, whose unique feature is maintaining of population of solutions that are *programs*. The GP algorithm automatically builds computer program, that solves certain problem, with no requirement of user knowledge and no specification of structure of solution. In other words, GP is a domain-independent and systematic approach to automatically synthesize computer programs from a high-level instruction sets and information what needs to be done [77]. Programs generated by GP are usually much shorter than ordinary computer programs. They typically consist of several dozen instructions, thus they can be identified with single function instead of whole computer program. Moreover sets of instructions that takes part in GP usually are not Turing-complete, because they are limited to instructions required to express solution to certain problem only. Of course this is not a rule, there were many successful attempts to evolve Turing-complete programs by either providing indexed memory [89], creating recurrent program structures [96] or designing Turing-complete set of instructions [53]. Despite the power of Turing-completeness, there is a generic issue common for each approach: undecidability of the *halting problem*. The problem can be easily worked around by limiting number of operations executed in a program run. Furthermore involving *halt* instruction radically lowers percentage of non-terminating programs (in certain limit of executed operations) [53].

Evolution in GP stochastically transforms population of computer programs from one generation to the another, hopefully better generation. As a random process, GP cannot guarantee good results and convergence to the global optimum. But, like the nature, GP is very successful in extraordinary ways of solving problems. A human-competitive design of miniature antenna that is neither straight nor elliptic [58] or optical fibers with non-uniform but symmetric structure [61] are just some examples of surprising GP solutions. More Genetic Programming applications are described in Section 2.4. Furthermore advantage of non-deterministic characteristic of the GP trial is ease of escaping from traps (e.g. local optimum) in fitness landscape, that capture other deterministic methods [77].

The basic GP algorithm is specialized form of Evolutionary Algorithm in terms of genetic operators. Detailed procedure is shown in Algorithm 1. The initial population is randomly generated

Algorithm 1 The basic Genetic Programming algorithm.

1. Initialize startup population P_0 of randomly generated programs from set of available instructions
2. Evaluate fitness of each program $p \in P_0$
3. *Select* two parent programs $p_1, p_2 \in P_n$ with probability proportional to their fitness
4. Do one of the following with given probability:
5. α_c: *Crossover* parents p_1, p_2 to create 1 or 2 children c_1, c_2 and add them to pop. P_{n+1}
6. α_m: *Mutate* parents p_1, p_2 and add them to population P_{n+1}
7. $1 - \alpha_c - \alpha_m$: *Reproduce* parents p_1, p_2 into population P_{n+1}
8. Repeat steps 3 – 7 until population P_{n+1} achieves desired number of individuals
9. Evaluate fitness of each program $p \in P_{n+1}$
10. Let $n \leftarrow n + 1$
11. Repeat steps 3 – 9 until ideal individual is found or stopping condition is satisfied
12. Return best found individual

from given set of instructions. Before selection starts, each program must be evaluated. It is done by running the program against given set of test cases. The closer program is to desired outputs, the better fitness it gains. Fitness function can be any function compliant with user's optimization target. It can be measured as amount of error between desired output and program output, the accuracy of program classifying objects or cost (time, money, fuel) needed to bring system to desired state [77]. The main role of fitness function is to indicate areas of good solutions. After parents are chosen, crossover begins with given probability α_c, usually $\alpha_c = 90\%$ or higher. On the other hand mutation probability α_m is much smaller, typically about $\alpha_m = 1\%$. If the probabilities do not sum up to 100%, the additional operator – reproduction is used with rate $\alpha_r = 1 - \alpha_c - \alpha_m$. The reproduction operator simply copies parents to new population [77]. Genetic operators run in loop until new population achieves desired number of individuals and then it replaces the old one and new generation begins. After ideal individual is found or stopping condition is satisfied (e.g. maximum number of generations is reached) the algorithm returns best found individual.

2.2 Genotype representation

From all program representations in Genetic Programming the tree-based one is the most popular. The tree-based Genetic Programming is alternatively called *canonical GP*. Each tree node represents separate instruction in here and connections between them symbolize a control flow. Intermediate nodes take evaluation results of their children as arguments and returns own result to the parent. The root node returns program output. Since leaf nodes have no children they have to take their value from external source (e.g. program input) or constant. The instruction with one or more arguments is named *nonterminal*, on the contrary instruction with no arguments is called

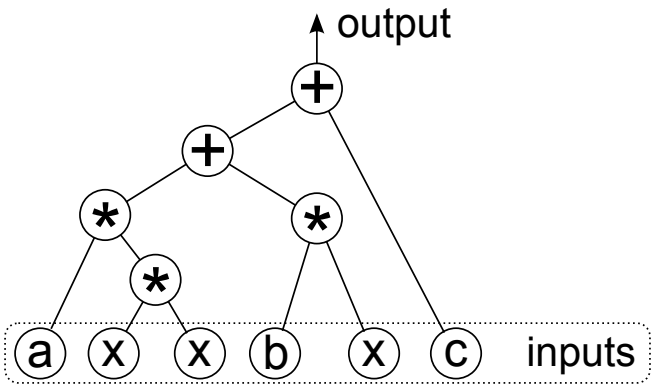

FIGURE 2.1: Example genome in canonical Genetic Programming. The tree in the picture represents function $ax^2 + bx + c$, where x is independent variable (function argument) and a, b, c are constants.

terminal. Therefore set of instructions can be split up into *nonterminals* and *terminals*. The first one is applicable to intermediate nodes, the second one to leafs. The term *instruction* can be used interchangeably with *primitive*.

The example genome can be found in the Figure 2.1. There is a second-degree polynomial in the picture represented as syntax tree by nonterminal set $\{+,*\}$ and terminal set $\{a,b,c,x\}$, where a, b, c are constants and x is independent variable. The terminal set is treated as program input with different meaning for each input. Note that, the program structure may be considered as redundant. There are three instances of x, which technically can be combined to a single node, but it is not usually done, because nodes may keep state information, different for each one. Moreover entire representation is redundant by definition. For example, consider the tree, where $a \times (x \times x)$ is replaced by $\sum_{i=1}^{a} x \times x$ for natural $a \geq 1$, the semantics is the same, but syntax is very different. The sum operator can be implemented as a subtree built from $\{+\}$ nodes only.

It is common in Genetic Programming that trees are written in *prefix notation* in text. The example tree from Figure 2.1 becomes written as $(+\ (+\ (*\ a\ (*\ x\ x))\ (*\ b\ x))\ c)$. This notation makes easier to extract subtrees from entire tree and it shows relationships between them at glance. The prefix notation will be used later in this master's thesis.

One might be aware of *type safety* in GP, because different primitives may have different types of arguments and outputs, therefore they may be incompatible in terms of *evaluation safety*. This is known problem in GP and there are two general approaches to solve it. The first one is *strongly typed* GP [67], the second one is to ensure *closure* property of instruction set [44]. The strongly typed GP assumes that each primitive has a type for its every argument and for return value. Therefore a tree is built taking into account type system's constraints. Moreover every genetic operation must guarantee that these constraints remain satisfied. The type system implies a formal grammar that may help implementing type-safe genetic operators and reduce program search space. On the other hand, in non-typed GP, there is assumption that instruction set has *closure* property [44]. Closure means here that every primitive in the set may get return value of every other primitive as an argument, therefore its return value may be given as an argument to another primitive. Sometimes it is possible to weaken type consistency by introducing automatic type conversion mechanism. For example integer value may be cast to floating-point, and boolean may be obtained from number by treating all non-negative values as truth and all negative as

false. The above mechanism may be considered as quite limiting, but usually simple refactoring of primitives can resolve apparent problems. Let us consider *if* instruction. Its definition has often three arguments: decision variable, if-true statement and if-false statement. The first of them is clearly boolean, however taking into account conversion mechanism discussed above it is possible to take numbers as all three arguments.

There are other forms of GP with custom program representations like *linear GP*, *graph GP* and *probabilistic GP*. In linear GP [8, 69], program is built upon list of instructions, therefore it is similar to ordinary assembler-like listing. The program usually executes on virtual machine, but sometimes it is generated for real CPU. The program stores its intermediate and final results in registers. Sometimes jump instruction is involved. Generally unrestricted usage of jumps leads to infinite loop, consequently additional steps must be done to cope with them.

Graph GP is natural extension to canonical GP since tree is special form of graph. There are few different approaches to graph GP. The first one, Parallel Distributed GP (PDGP) [72, 73] is similar to tree-based GP in program representation, but allows instruction to take as argument any other node, the constraint is that the graph must remain acyclic and connected. In addition PDGP gives ability to represent programs as logic networks, neural networks, recurrent transition networks and finite state automata. The other approach involving graphs is Cartesian GP [65, 66]. The programs are represented here as lists of integers. The list is decomposed into groups of three or four integers. Each group is associated with corresponding position in 2-D array. First integer in each group selects the primitive for that position in the array, the remaining integers point at arguments positions in the array. The instruction does not define itself where its output is used, this is done by other primitives. The output of primitive may be used multiple times or not at all, depending on the way in which the inputs of other functions are specified.

In probabilistic incremental program evolution (PIPE) [80, 81] the program is represented as tree of probability tables. Each row in the table defines probability that particular primitive will be chosen in specific node in the newly generated program tree. In each generation, population is formed from programs created upon current tree of probability tables. The process of generation begins by selecting root node from root probabilistic table and then control goes down the hierarchy and chooses instructions from lower tables. The process ends for particular branch if terminal is chosen.

The canonical tree-based representation is used in this master's thesis if not stated otherwise.

2.3 Genetic operators

2.3.1 Selection

The intuition behind evolution is that the good individuals (solutions, programs) are reproduced with higher probability than worse equivalents. However higher selection pressure, or in other words higher domination of better individuals over worse, leads to lower diversity between children. These two aspects of selection are conflicting, but both of them are desirable. The pressure of selection can be controlled by fitness scaling. Too high selection pressure leads to convergence in local optimum, but too weak pressure causes very high diversity, therefore evolution process behaves like random search algorithm. The most popular techniques of selection are listed below

[26, 27]. Let $f(p)$ be a fitness function and $f_i = f(p_i), \forall p_i \in P$ be a fitness value of the individual p_i and $e_i = |P| \times \frac{f_i}{\sum_{i=1}^{|P|} f_i}$ be an expected number of its copies in new population.

The *fitness proportionate* or *roulette wheel selection* is a method where individuals are assigned to fields on the roulette wheel with size proportional to their fitness f_i. The roulette wheel is spun $|P|$ times, selecting one individual each time. Roulette wheel selection is one of the simplest methods of selection, thus it has multiple disadvantages. First of all, the roulette scheme does not guarantee that the best fitted individuals will be chosen, however their probability of selection is high. Thus, this method may have run of the mill results, especially for small populations. Moreover the selection pressure changes over time. Consider four individuals with fitness values $\{1,3,4,5\}$, according to the algorithm the individuals have following probabilities of selection (in the same order as above) $\{0.08,0.23,0.31,0.38\}$. Now let us assume that evolution improved each solution by 50 fitness points. Consequently current fitness values of individuals are $\{51,53,54,55\}$ and their corresponding probabilities are $\{0.24,0.25,0.25,0.26\}$. As you see the relation between solutions did not change but probabilities did. The selection pressure lowered, therefore good solutions are selected less likely. Although the problem can be worked around by normalizing or standardizing fitness before computing probabilities. Regardless of mentioned disadvantages, roulette wheel selection is successively applied in many systems because of its simplicity and low computation complexity.

In the *random selection according to residuals without repetitions* each individual p_i is copied $\lfloor e_i \rfloor$ times to the new population. The remaining (single) place is assigned to individual p_i with probability $e_i - \lfloor e_i \rfloor$. On the contrary in *random selection according to residuals with repetitions* the remaining places are filled by roulette wheel selection, with field size proportionate to individuals fractional part of e_i. Very similar method is *deterministic choice* where "standard" places are assigned as previous, but remaining ones are filled in descending order of fractional part of e_i. The main advantage of these methods is guarantee that good individuals will get their places in upcoming population.

The very popular method is *tournament selection*, where fixed k individuals compete in the tourney. They are chosen to the tournament by roulette wheel selection. The individual with highest fitness becomes winner of the tournament and it is added to the new population. The entire process is repeated until there is no free space in new population. The tournament selection has advantage over fitness proportionate selection in constant selection pressure.

Ordered selection is a method where individuals receive ranks according to their fitness. The individuals are chosen with probability density function defined on their ranks. There is wide range of functions that are applied, linear and non-linear. The function parameters allow to control selection pressure. This method instead of taking into account amount of differences between individuals, it takes only fact of difference, therefore it guarantees that selection pressure is not dependent of fitness absolute values. Unfortunately someone may consider number of possible functions and their parameters as confusing, therefore a simpler method is usually involved.

2.3.2 Crossover

Like in the nature, aim of crossover is to mix genetic material in such way, the children inherit good traits of their parents. Because crossover needs two (or sometimes more) parents, the operator

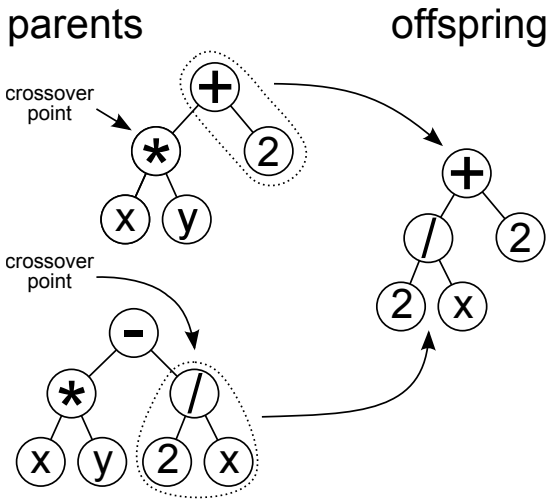

FIGURE 2.2: Subtree crossover operator in Genetic Programming.

usually takes its arguments from selection. Since there is a plenty of ways to mix genetic material in Evolutionary Algorithms, there is no best-one crossover operator and Genetic Programming is not an exception. Because of specific, tree-based, genome representation in GP, brand new operators have been proposed over the years. The few examples of them are described in this section.

The simplest and commonly used operator is *subtree crossover*. The operator takes pair of parents as arguments, then it randomly selects one node in each parent independently. The selected nodes are named *crossover points*. Afterwards the process creates child by copying first parent tree without subtree rooted at crossover point and replaces the missing branch by subtree of second parent rooted at its crossover point. The example of subtree crossover of two parents is illustrated in the Figure 2.2. It is possible to select crossover points multiple times, therefore creating whole family of children from single pair of parents. It is also possible that single crossover iteration creates two children by swapping parents in above routine, but it is not common behavior [77]. Note that, primitives except terminals usually have from 2 to 4 arguments, consequently average branching factor for entire tree balances between 2 and 4. Because that, leafs are $\frac{1}{4}$ to $\frac{1}{2}$ of all nodes. Therefore uniform distribution of crossover points causes exchange of small amount of genetic material due to lower nodes are chosen more frequently. In addition, many crossover runs may be reduced to simple swapping of two leafs. To work around the problem, Koza [44] proposed widely used technique to select intermediate nodes with probability 90% and leafs with probability 10%. This is part of GP setup often called *Koza-I* [44].

The previous crossover operator does not preserve position of genetic material due to parent subtree may be moved far away to totally different place in the child tree. This is other behavior than in the nature, where genes in child genome correspond to parent genes at approximately the same position. The crossover, that preserves genes position is called *homologous crossover*. There has been proposed several approaches to homologous crossover, few of them are described below. The popular one is *one-point crossover* [74, 76] shown in the Figure 2.3. Given two parents, the operator aligns them to determine their *common region*. The common region is defined as topologically equal part of both trees starting at root node. Then *common crossover point* must be

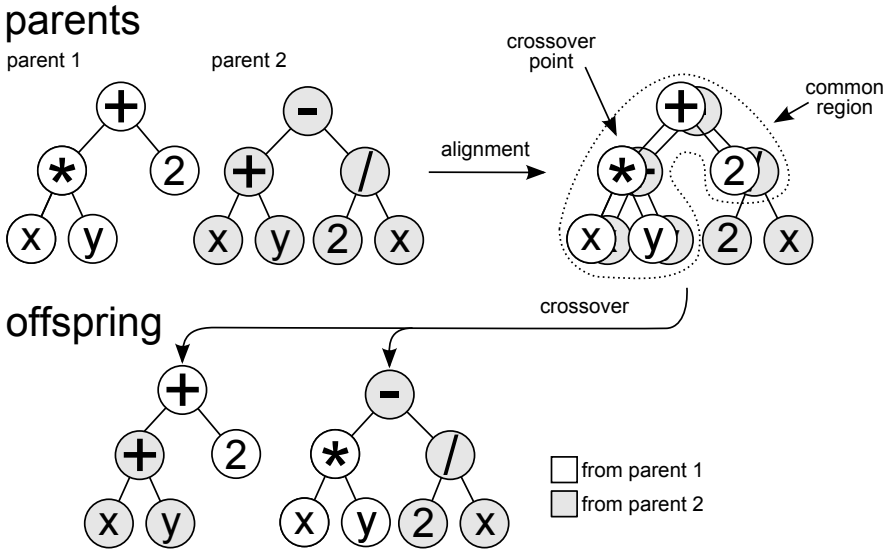

FIGURE 2.3: One-point crossover operator in Genetic Programming.

selected upon common region. The point is selected randomly from available nodes. The operator produces two children by swapping parents subtrees rooted at crossover point. Since common region in both parents has the same shape, it is safe to replace entire subtree without violating homology constraint.

The next homologous operator is *uniform crossover* [75]. The method works like uniform crossover in Genetic Algorithm, therefore it goes through the tree and draws lots whether pick a node from first parent or another. Moreover if picked node is nonterminal node then entire parent subtree is also copied. Since the trees can have different shapes, only nodes inside common region can be chosen to not violate arity constraints. The common region is determined identically like in one-point crossover. This method tends to mix genetic material stronger near the root than other crossover operators.

The *context-preserving crossover* [18] behaves similar to one-point crossover, but the crossover points are not limited to the common region. Instead, there are additional constraints put on path from root node to the point. There are given coordinates to each node. The coordinate is defined as vector of choices on path from root to the node. The choice is defined as number of selected branch in particular node. In strong context-preserving crossover, only nodes with exact coordinates can be selected as crossover points. In weaken version of crossover, the point in first parent is selected as previous, but in second parent, any node in subtree of valid crossover point coordinates may be selected. The strong version of context-preserving crossover may reduce diversity, on the contrary weak version works around the problem.

The *size-fair crossover* [51] works similar to the subtree crossover. The crossover point in the first parent is chosen randomly, then the size of subtree to be removed is computed. The crossover point in the second parent is selected with regard to the computed size in order to limit unfairly big structures. Therefore the method is helpful with reducing code bloat.

Regardless of the method, crossover has a heavy preference for constructing non-uniform distributions of tree sizes [71]. The crossover operator generates short programs more often than longer ones. Usually only few generations are required to completely rearrange tree shapes and

sizes. The mutation operator can be helpful in coping with the problem.

However there is some evidence [32], that GP crossover affects inheritance of good parents traits by their children, the same evidence shows that crossover may cause random-like changes in the program and therefore loss in program fitness. This is known issue in the Genetic Programming caused by complex relation between program code and program results. It is very hard, or even impossible to define how similar is one subroutine (e.g. subtree) to the another. Consequently it is hard to tell, how to change program (e.g. how to select crossover points), to improve its results. Someone may suggest that crossovers near the leafs causes less changes in the program code, therefore it causes lower changes in its output. This is not particularly true. With analogy to rounding errors of floating-point arithmetic, the small reorganization of computation order may dramatically change the results. In other words small adjustments of intermediate results in deep tree nodes, can completely change a way, the higher nodes behave. The problem will be addressed later in Section 3.2 of this master's thesis.

2.3.3 Mutation

Since crossover may lead entire population to premature convergence to local optimum [57], there is another genetic operator utilized to preserve diversity – *mutation*. In common sense, the operator makes random changes to the genome, more formally it moves search process from one part of solution space to another. The following paragraphs describe the most common mutation operators for canonical GP.

The most broadly used form of mutation is *subtree mutation*. The method randomly selects one node in the tree as *mutation point*. Then it replaces original subtree under mutation point by new one generated randomly. The subtree mutation is sometimes implemented as subtree crossover between original program and random generated one. The operation is also known as *headless chicken crossover* [5, 33]. Moreover there has been proposed an amended version of operator, that limits new subtree depth to grow no more than 15% [39]. The modified method is useful to fight with bloat of code. Furthermore Langdon proposed another modification of this behavior called *size-fair subtree mutation* [50]. The operator guarantees that new subtree will be, on average, the same size as replaced one. The method split up into two cases. In first case it samples uniformly in program space with guarantee that length of subroutine will be in the range $[50\%, 150\%]$ of original subroutine. On the contrary, in the second case, the operator samples uniformly in the space of program lengths limited to the same range. Experiments have shown that sampling in program space causes far more code bloat than in space of program lengths.

The other common method is *point mutation*, an equivalent of bit-flip mutation in Genetic Algorithms [27]. For each node the method chooses with certain probability whether the node will be mutated and if so, it replaces the node with another primitive with equal arity. If there is no other primitive with the same arity, the node remains unchanged. The notable difference between two above approaches is that the first one modifies entire subtree, therefore shape of the tree may change. On the other hand, point mutation does not change shape of tree, but only semantics of particular nodes.

In *shrink mutation* [4] the mutation point is chosen randomly and then whole subtree is replaced by randomly generated terminal. Thus, this is special case of subtree mutation, where

the tree is replaced by terminal. Obvious motivation to shrink mutation is reduction of program length.

The *permutation mutation* is applied to at least binary nodes. The method randomly selects children of the same parent and then permutes their positions with entire subtrees. It is worth noting that this operator makes no changes in program semantics if chosen primitive is symmetric, such as addition or multiplication.

However some researchers argued that mutation is not necessary [44] and Genetic Programming performs well without it, others have shown that mutation can be advantageous. There has been demonstrated that combination of six mutation operators and no crossover operator performs as well as standard GP with crossover [14]. The work compared both quality of results and computation effort on 14 well known problems. Some other researches have shown that hill climbing with mutation can outperform standard crossover-based GP without mutation [28]. Generally situation is complex, as described in [60], the impact of mutation and crossover depends on a problem and details of GP system. Moreover there is no evidence that any particular combination of both crossover and mutation performs significantly better than either mutation or crossover alone. Nowadays, it is advised to involve mutation in GP trial with low probability [70].

2.4 Applications of Genetic Programming

Over the years Genetic Programming has proven its high usability by extraordinary number of successful applications. Since its prevalence by John Koza in early 1990s [43, 44] there was written more than 8000[1] of reports, articles and books about GP and its number is growing rapidly. The Genetic Programming is especially useful where relationship between relevant variables is unknown or poorly understood or the structure of solution is unknown. In addition, as a method of machine learning it works well if significant amount of learning data is available in systematic form. Moreover Genetic Programming has mechanisms to deal with noise in data aswell. Note that GP, like the nature, does not guarantee that it finds the best possible solution, it rather create "good enough" approximation of the best one. It is important to say that GP is very good scalable process, therefore it can replace conventional mathematical analysis if its computation time is unacceptable.

Genetic Programming has numerous achievements, it copes well with curve fitting, data modeling and time series prediction [34], control of industrial process [13, 41], synthesis of electronic circuits [9, 45], robot controlling [52], image and signal processing [31, 46, 88], lossy [68] and lossless [37] compression and more.

The most common and widely analyzed [7, 38, 44, 94] task for GP is symbolic regression problem with all its variations. The symbolic regression is the problem where goal is to find a function whose output matches desired values. In ordinary sense *regression* means finding coefficients of predefined function that best matches data. Although if given function does not describe data good enough, the traditional regression fails. Then the experimenter must try with more and more models as long as he or she finds the best one. The work is laborious even for experienced analyst. Furthermore, even expert analysts tend to have mental biases when choosing models. The

[1] Bibliography on Genetic Programming http://liinwww.ira.uka.de/bibliography/Ai/genetic.programming.html

symbolic regression solves all these problems without user attention. It tries to fit the data as much as possible with given set of instructions, without making any assumptions about structure of the output function.

Before Genetic Programming trial starts, some preparatory steps must be done. First of all, instruction set must be chosen. The set should be sufficient to express acceptable or even best solution[2] to the problem but semantic redundancy is allowed. For example multiplication and power with constant integral exponent are redundant, since all equations that can be represented with power may be also represented by multiplication, but power can speed up evolution process and simplify results, especially for high exponents. Note that, oversized instruction sets do not tend to slow down evolution very much, but in some cases they may bias system in unexpected way [77]. The symbolic regression problem typically works on set of $\{+,-,*,/,0,1,x\}$ instructions. The set contains division, therefore division by 0 must be handled in some way. It is usually done by returning 0 if divisor is 0. The modified version of division is called *protected division*. It is common to use *protected* versions of other primitives like logarithm, square root or modulo. The alternative to protected primitives is penalization mechanism. The mechanism relies on run-time error checking and reflects them in fitness penalty. However if tendency to generating invalid expressions is significant, it can lead to many individuals in population with nearly the same, poor fitness. This may impede selection to choose which individuals may be good parents [77]. In next step constraints to the size of tree, genetic operators and their probabilities need to be chosen. Usually standard values as described in Section 2.1 and its subsections are sufficient. The final step is preparation of test set. It commonly contains from tens to hundreds of test cases. Each test case is a $k-tuple$ consisting of $k-1$ values of independent variables and 1 value of dependent variable. The relationship between $k-1$ independent variables and dependent variable ought to be found.

However the symbolic regression problem naturally works on continuous variables, the Genetic Programming behaves very well on discrete problems. Most of considered of them are logic problems. Usually the task is defined as synthesis of logic gate for given set of inputs and desired outputs. The objective is to find logic function that matches all of the test cases. Typically evolved gates are multiplexers, comparers and parity gates. A multiplexer gate takes $a \geq 1$ address bits and 2^a data bits as inputs and returns as output a bit indicated by the address. Common problem instances are 3-, 6- and 11-multiplexers. The numerical id states for sum of address and data bits $(a+2^a)$, consequently it clearly and unambiguously defines the instance. The another gate – comparer takes two equal-length sequences of bits, each representing a number, and returns 1 if and only if the numbers are equal and 0 otherwise. Alternatively the comparer can verify if one of the numbers is less than the second one. The transition function of last one – the parity gate checks if vector of input values contains more ones or zeros. All these gates are commonly employed in many digital circuits, particularly in CPUs, RAMs and network controllers. The nonterminal set of available instructions, used to synthesize the gates, usually consists of logic 'and', 'or', 'not' and sometimes conditional 'if' statement, while terminal set is built upon input variables. This faction of the Genetic Programming paves the way to the new branch of science called Evolvable Hardware [9, 35, 55].

[2]Formally instruction set is sufficient if the set of all possible recursive compositions of primitives contains at least one solution [77].

Because of clarity of symbolic regression, it will be involved in many examples in this master's thesis and logic problems will be used as benchmarks in the experiments due to discrete nature of discussed approach.

Chapter 3

Challenges for Genetic Programming

3.1 Typical problems and troubleshooting

3.1.1 Code bloat

Previously we have mentioned few times about *code bloat*, currently it is time to describe this phenomenon. In Genetic Programming, or more generally in Evolutionary Computation, the most significant improvement of the objective function value is noticeable in the early generations of run. The later generations usually provide less enhancement, but search process operate as usual checking more and more solutions. It causes that the GP browses syntactically different programs and from time to time it adds additional level of depth to the tree. It is worth noting, that number of syntactically different trees rises exponentially with increase of their depth. Therefore the trees become progressively more complex, but the increase in their fitness is usually not significant. This observation is called bloat of code.

However the code bloat is very widely analyzed problem in Genetic Programming, there is no generic and universally-accepted explanation for wide spectrum of observations [77]. One may argue that the bloat is caused by evolution of *dead code*. The dead code, alternatively named *intron*, is branch of tree, that does not contribute to program output. For example consider tree with 'multiply' instruction in an intermediate node. If one of its children always evaluates to 0, evolution of the second child does not change anything. Thus the second subtree is dead code. Since modification of dead code does not change program outputs, it can grow limited only by additional constraints given on tree shape, which usually are not very restrictive. Experiments [85] have shown that dead code tends to be placed in lower parts of trees, consequently the dead branches are typically smaller than average size of subtree. The same experiments demonstrate that crossover in dead regions causes code to grow. This explanation is known as *removal bias theory*.

The *nature of program search spaces theory* [54] states that code bloat is associated with variable-length program representation. The empirical results have statistically confirmed that there is no meaningful correlation between length of program and its fitness distribution. Because there is much more long programs than shorter ones, it is more likely that the longer programs appear in the population. Therefore the population is only a sample of entire program space.

Finally, another theory states that there exists the *replication accuracy force* [62], that biases

the evolution to promote individuals, which produce functionally similar offspring. However existence of functionally similar trees leads to better average fitness and lowers its standard deviation, mixing of similar genetic material decreases diversity and causes bloat.

Bloat would not be problematic from practical point of view unless it affected performance and caused difficulties in interpretation. That is, long and complex programs are computationally expensive during execution as well as during evolution. In consequence genetic operators such as crossover and mutation must process more nodes in average case. The complex programs are also hard to interpret and understand. Additionally they may encounter poor generalization features.

There have been proposed several effective methods to troubleshoot the code bloat. The most common of them are listed below. The simple, intuitive and one of the oldest method is *limiting of size and depth*. The method was firstly discussed by Koza [44], it is no more than hard limiting of maximal depth of tree and less likely number of nodes. Therefore if genetic operator generates tree that violates the above constraints, the tree can be either rejected or pruned. If tree produced by crossover is rejected, one of its parents is returned instead of the child. It is clear, that this approach is effective, but it has serious disadvantage. The programs that are more likely to violate constraints are copied unmodified into new population more frequently than others. Consequently entire population is led into state, where it is filled by programs that nearly break the restrictions. The tree pruning even more likely tends to similar consequences. Experiments [16] have shown that depth threshold biases population to be filled by very bushy programs with majority of branches reaching depth limit. On the other hand, limiting of node number produces very tall and simultaneously sparse trees that reaches the size limit in several generations. To cope with above issues, one may not return parent if genetic operation fails, but repeat operation either with new parents or new crossover or mutation points few times before giving up. Note that minimal size and depth limits must be coordinated to allow formulation of proper solution. Moreover it is essential to give some additional space to evolution by expanding acceptable range of values, typically to $[50\%, 200\%]$ of expected solution size [77]. Usually the limits are chosen by some trial and error runs.

Next approach is known as *anti-bloat genetic operators*. As the name states, it involves designing of genetic operators that either directly or indirectly prevents formation of bloat. In general words, the operator tries several configurations, calculates projected tree size or depth and makes a decision to limit unattended growth of program. For example see *size-fair crossover* and *size-fair mutation* discussed previously in Sections 2.3.2 and 2.3.3.

Other common technique is called *anti-bloat selection*. There are many variations of this technique. The widely known *parsimony pressure* [44] method will be discussed here. The method operates by punishing programs for their sizes. It defines new fitness function $f'(p) = f(p) - c \times l(p)$ that gives certain amount c of penalty for each involved program building element. The $l(p)$ states for length of program p and $f(p)$ states for original fitness function. The selection mechanism takes into account new fitness function $f'(p)$ in order to choose individuals. Note that original fitness function may be still required to recognize solutions and stop trial.

Concluding, bloat of code is significant problem in Genetic Programming. It causes programs to grow above the norm, in consequence they become difficult to understand, expensive to both evaluation and evolution and above all they poorly generalize. However there are several techniques to fight with the phenomenon, its general origin is not known. Overall, it is recommended

to control bloat on the problem basis with use of trial and error methodology.

3.1.2 Other problems in Genetic Programming

Despite of code bloat, in Genetic Programming, there are many other problems, to which attention must be paid. The checklist of issues that need to be inspected while debugging GP trial is shown below.

Because GP is random search process, different runs may return different results. Therefore inferences have to be formulated very carefully and thoroughly reviewed. One may consider 10 runs of GP trial, all failing, in consequence he or she may conclude that GP (with specific settings) is not good framework for particular job. Although if success ratio, for particular settings, is 5%, failing probability in all 10 runs is $\binom{10}{0} \times 0.05^0 \times 0.95^{10} \approx 59.8\%$. However there is reasonable probability of unsuccessful, it decreases with successive runs. Obviously there is no hard threshold indicating that the repetition should be discontinued. Therefore it is recommended to make use of appropriate statistical tests to confirm that inferences are significant in terms of numbers.

The consecutive runs lead us to problem of *high computational effort* of Genetic Programming. Since every single GP run usually takes from minutes to hours depending on population size, number of individuals involved, quantity of test cases and available computing resources, the *n* repetitions pull the process into hours or even days. Because of that, there have been developed several methods for process parallelization and distribution over available machines. The process can be distributed in many ways. For coarse-grained parallesism each GP run can be executed on different machine and results would be aggregated at the end of computation for future analysis. In the contrary, in fine-grained parallelism model, the population can be divided into several partitions and each partition can be processes by another machine or processor. It is worth noting here that program fitness evaluation is usually the most computationally expensive part of entire trial. Therefore it is sufficient to parallelize only this part of algorithm. If one would like to parallelize a process himself, it is important to take into account concerns of synchronization mechanisms and maximal possible speedup coming from parallelization, which can be calculated according to Amdahl's law [2]. Let us describe another commonly known distribution model called *island model* [3], where population is split up into subpopulations and each of them settles another island (a physical computer). Then evolution acts independently on each machine. Every few generations a random sample of individuals is chosen and moved to another island in order to bring there a breath of fresh air.

From above considerations we have come to *maintaining of diversity* problem. The problem is essential to every population-based algorithm, not only to Genetic Programming. As mentioned previously in Section 2.3.2, crossover operators tend to converge population into small area in search space, therefore all solutions are nearly the same. It causes that most areas, likely promising, remains unexplored by search process. Therefore it is recommended to add diversity preserving mechanism to the process, such as mutation. However the mutation is important, its high probability turns search process into completely unguided random trial. Therefore mutation probability should be as low as 1%, what was described in Section 2.1. Note that, measurement of diversity can be distinguished into two kinds. The *frequency of primitives* determines sharing rate across solutions for each primitive separately. If the rate drops down to meaningless values,

it may mean that this primitive is either unnecessary or there is an problem with diversity. Manipulating of mutation rate should be helpful here. On the other hand *variety of population* measure computes percentage of distinct individuals in the population. If it falls down below 90% [77], it may indicate problems. Although high values do not mean that everything is right. Intuitively completely random search process would have the highest possible value of diversity. Additionally sometimes programs produced by GP contain introns. Since the introns do not contribute to the program's output, the programs that are not syntactically identical, may behave in the same way. Because they are different in their syntax, they contribute to diversity measure, but it does not mean they are different in useful manner. Consequently one may consider measurement of semantic diversity instead of syntactic diversity. For future investigation refer to the [63]. More information about relation between program syntax and its semantics can be found in Section 3.2.

Genetic Programming, like any other Evolutionary Algorithm, has plenty of parameters that need to be tuned. These parameters differ mostly in their nature. Some of them, like genetic operators probabilities or population size, are continuous (of course there cannot be half of an individual, but the range of proper values is infinite, therefore we call it 'continuous' in common sense). Others, like instruction set and collection of genetic operators, are naturally discrete. The amount of available parameters gives a headache even to the most experienced experimenters. Additionally the discrete parameters are not as easily adjustable as continuous ones. Consider addition or removal of single primitive, how it will affect the results? There is no easy answer to that question. The user must ask himself if the instruction is widely used by individuals, if it has semantic equivalent produced from other primitives, if the instruction is required to build a target program. Changes in the discrete parameters have often more radical effects than one may suppose. In consequence it is advised to not make a hasty decisions, but make use of statistical analysis, as it was mentioned before.

In conclusion, do not let early failures to discourage your efforts. There are many factors influencing success of GP trials. However most of them are hardly tunable, statistical analysis comes to the rescue. Despite its power, it requires array of data to get proper inferences. That amount of data rises computational effort needed for its generation. Fortunately there are multiple models of parallel and distributed computing for Genetic Programming, that are very handy in lowering time of processing. Moreover some of ready-to-go GP frameworks bring such functionality with itself.

3.2 Locality and global convexity in context of Genetic Programming

Each program individual in population must be evaluated in order to compare with other individuals. However quality assessment of program itself is very hard or even impossible, it is possible to evaluate it against set of test cases. Each test case consists of set of program inputs and desired output. The vector of program outputs for given collection of test cases will be called *semantics of program*. Formally, let $\mathscr{P}$ be a space of all possible programs and $p \in \mathscr{P}$ be a program in that space. Let $s(p)$ be a semantics of program p and S be a metric space of semantics, therefore s is a *semantic mapping* where $s : \mathscr{P} \to S$. It is assumed that s is surjective (e.g. $S \equiv image(s)$).

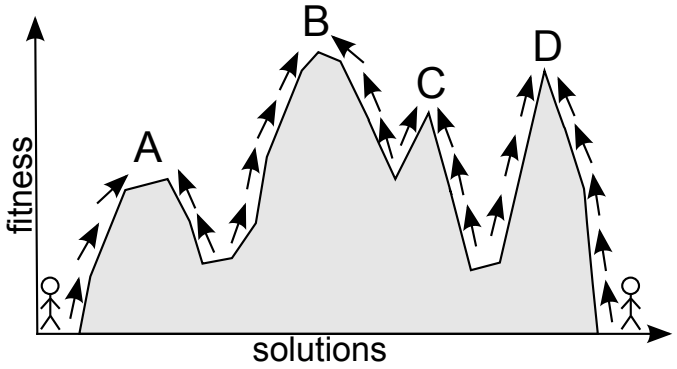

FIGURE 3.1: Example fitness landscape. Arrows indicate improvement paths, A, B, C, D stand for local optimums and B is additionally global optimum, a travelers denote search process start points. If search starts to the left of A, then local optimum A can be found without any problems, but B is separated by valley, therefore, only search algorithm, that is able to jump over valley can find it. Note that if search process starts to the right of D, it is far more difficult to find B because the valley between C and D is much deeper than between A and B and there is another local optimum C on the path.

Usually semantic mapping is not invertible because program space is much bigger then semantics space ($|\mathscr{P}| \gg |S|$). Consequently it is possible, that there is set of syntactically unique programs but semantically equal. In particular, two programs varying only in swapped subtrees under symmetric node have this property. More formally, let define $[p] = \{p' : s(p') = s(p)\}$ as *class of semantic equivalence* of program p. Since any program $p' \in [p]$ is indistinguishable by fitness function with any other $p'' \in [p] \backslash \{p'\}$, it does not matter which one will be chosen as representative of entire equivalence class. The space of semantically unique programs will be denoted by $\hat{\mathscr{P}} = \{\hat{p} : any([p]), p \in \mathscr{P}\}$, therefore $|\hat{\mathscr{P}}| = |S|$.

The semantic mapping is identified with genotype – phenotype mapping in the nature. The features of individual phenotype (here: program semantics) influence reproductive success. The replication rate of each individual is known as fitness function, mentioned previously. The better fitted individual, the higher its replication rate, the worse fitted one, the higher chances for extinction. By combining space of genotypes with fitness function obtained from their phenotypes as additional dimension, a new space, called *fitness landscape* [15, 40, 95], will be created. The landscape is nonuniform, with plenty of hills and valleys. Assuming that higher values are better, the hills represent local optimums and at least one of them is globally optimal, thus the optimal one is the highest of them. Although finding the global optimum is not trivial. See Figure 3.1 for example. Since there are many valleys separating hills, the simple hill climbing method does not guarantee finding of global optimum. Consequently finding of optimum requires jumping from one hill to another, climbing to the peak and then checking another one. The problem with this approach is hidden in unexplored space of solutions. It is not known where are other hills until search process finds it. In addition, the genotype space is usually very large or even infinite, therefore it is not possible to explore all its parts. Furthermore there are better and worse areas. Thus good area surrounded by broad bad region can be a trap to any non-exhaustive optimization algorithm. In contrary, global optimum surrounded by very poor solutions may not be found because entire area is not promising.

Fortunately, all is not lost, the fitness landscape of majority of real-world NP-hard problems has property called *global convexity*. The term global convexity refers to the fitness landscape.

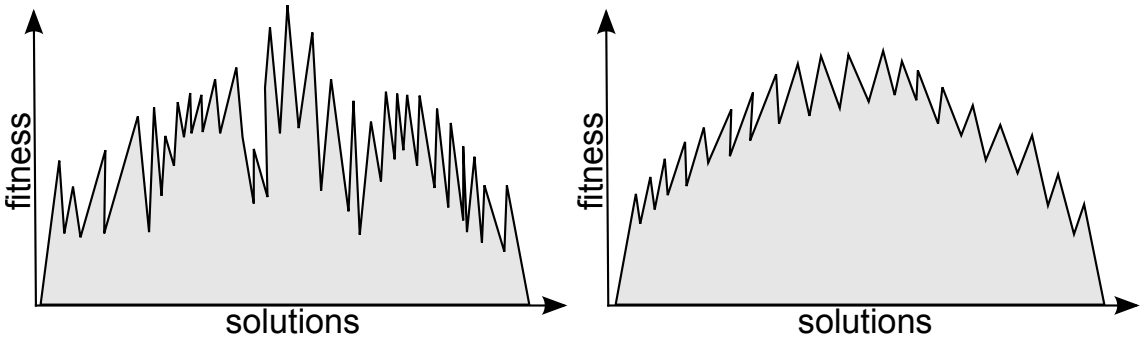

FIGURE 3.2: Global convexity comparison. Left: barely visible global convexity, there are plenty of local optimums, the rising trend is disregarded. Right: the convexity is well outlined, variation of landscape is reduced.

It means that there is noticeable convexity of fitness function extended over entire landscape. As mentioned before, fitness landscape contains multiple pits and hills, therefore the term *convexity* here does not follow classic mathematical definition, it would rather mean convex in common sense. Note that, the landscape is smoother, the uphill climbing to the optimum is easier. For example see Figure 3.2. Landscape to the left is far more jagged, then right one. Therefore a search process, that is able to jump over small valleys can find global optimum in second case, but may stuck in local optimum in first case. Note that it has been proven that if objective function is non-convex and non-linear (in its mathematical definition), the optimization problem is NP-hard [49]. Therefore it is natural to consider problems with noticeable convexity as easier than more jagged ones.

It is intuitive that similar solutions have similar values of fitness function, therefore it can be expected that they lie near each other. However the statement is generally true, this is very weak property of Genetic Programming. That is because the nature of computer programs is digital, therefore even the smallest possible perturbation in program (e.g. change of single instruction) can have the most drastic consequences to the outputs [19]. Moreover the space of programs have at least three undesirable properties [59] from search process point of look. First of all the space is *non-orthogonal*, it means that every formula can be represented by multiple ways. For example consider addition of zero or multiplication by one, there is no effect in semantics, but syntax of expression changes. Similar redundancy can be found in Boolean formulas ($x \vee x \iff x$) and conditional expressions (*if x then y else z* $\iff$ *if* $\neg x$ *then z else y*). Since often there is no effective way to eliminate these redundancies, the program space is greatly expanded, consequently it is far from orthogonality. The next property is *oversampling of behaviorally distant programs*. The main cause for that is chaotic program execution and overrepresentation mentioned above. The overrepresentation itself leads to simpler programs being heavily oversampled, while complex ones are rare. Simplicity is defined here in terms of minimal program length that produces the same semantics. Finally, *semantically close programs are undersampled.* As shown previously, syntactically different programs can have the same behavior. Redundancy as well as non-redundant programs that achieve the same result by different ways can be responsible for this state of affairs. Consider expression $x+x+x+x$, it can be alternatively represented as $4 \times x$. As you see, both representations are syntactically far. Now look at expression $4.1 \times x$, it is very similar in syntax and semantics to the previous one, but it cannot be represented by addition. Therefore from point

of view of part of program space, where programs are constructed from additions the semantically similar expression is distant. In conclusion, the semantic mapping in Genetic Programming has very low locality and correlation between program fitness and distance.

However there is no method to measure global convexity itself, because it does not follow mathematical definition of convexity, it has been shown [24, 25, 29, 91] that in most Evolutionary Computation systems genotype distance correlates with locality in space of phenotypes. On the basis of Genetic Programming, high phenotype locality means that for each program in the program space, its direct neighbors produce similar semantics. This statement can be defined formally as follows. Let $N(p)$ be the *neighborhood* of program p ($N : \mathscr{P} \to 2^{\mathscr{P}} \setminus \emptyset, p \notin N(p)$). Normally the neighborhood is set of programs that can be derived from p by introducing small changes in it, like replacement of single primitive. From context of Genetic Programming, $N(p)$ can be identified with set of all mutants of p. The following expression is proposed to measure locality of single program p in its neighborhood $N(p)$:

$$l(N,p,s) = \frac{1}{|N(p)|} \sum_{p' \in N(p)} \frac{1}{1 + \|s(p'), s(p)\|} \tag{3.1}$$

The particular definition of neighborhood relation $N(p)$ is provided from outside. The $s(p)$ denotes semantic mapping and $\|s(p'), s(p)\|$ indicates metric between semantics of programs p' and p (e.g. a distance between semantic vectors). It is obvious that codomain of $l(N,p,s)$ is in range $(0;1]$. The value of 1 clearly say that all neighbors of p has the same semantics, in contrary values near 0 correspond to notable dissimilarities in neighborhood. It is worth noting that $l(N,p,s)$ is nonlinear, therefore high values of distance have much less influence to entire measure than small values. Moreover the localities of different programs in different spaces are incomparable because of unrestricted range of distances. For example consider a program space, where average distance between programs is 3 and standard deviation is 1, therefore, assuming normal distribution, average value of locality metric is $\int_0^\infty \frac{1}{\sqrt{2\pi}} e^{-\frac{(x-3)^2}{2}} \frac{1}{1+x} dx \approx 0.27$. Now consider another program space that differs only in average distance, which is 30 now, the average value of locality is $\int_0^\infty \frac{1}{\sqrt{2\pi}} e^{-\frac{(x-30)^2}{2}} \frac{1}{1+x} dx \approx 0.03$. As you see, the second distribution of distances between programs is only shifted on numerical axis, but locality metric changed a lot, even though distribution is pretty the same. To work around this problem *normalization* or *standardization* of distance metric is advised. However outliers have disadvantageous impact to the first one, the standardization causes that some values may be negative. To comply with the problem modified standardization is proposed. The modification consists of increase of standardization result by triple standard deviation, which is 1 after the operation, and then rounding up of negative values to the zero. Let define modified standardization as follows:

$$std(x,\mu,\sigma) = max\left\{\frac{x-\mu}{\sigma} + 3; 0\right\} \tag{3.2}$$

where x is value to standarize, μ is average and σ is standard deviation in population. Assuming normal distribution, only $1 - \int_{-3}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx \approx 0.13\%$ of observations will be rounded up to zero. The standardized distance metric will be used later in this master's thesis if not stated otherwise.

Let us generalize the locality concept to entire program space by averaging locality of program

neighborhood ($l(N,p,s)$) over all programs $p \in \mathscr{P}$:

$$L(N,\mathscr{P},s) = \frac{1}{|\mathscr{P}|}\sum_{p\in\mathscr{P}} l(N,p,s) \tag{3.3}$$

It is clear that $L \in (0;1]$, but $L = 1$ should not be expected in practice, as it happen if and only if all programs in $\mathscr{P}$ have the same semantics. Since semantic mapping s and its codomain S is given as part of problem specification and cannot change, the last argument of function will be suppressed, therefore shortest form $L(N,\mathscr{P})$ will be used later.

The method of increasing locality in program space will be discussed later in Chapter 4, whereas in the current paragraph we consider effects of locality improvement. The natural consequence of high locality is drop in diversity. The diversity is defined here as number of semantically different programs in the entire program space. In common sense, greater number of programs leads to finding better suited solution. Moreover the diversity is important from completeness perspective, since it increases probability of producing any result by a program.

Concluding, higher values of program space locality correlate with smoother fitness landscape and therefore noticeable global convexity. The convexity indicates the direction of improvement, consequently any non-random search algorithm, in particular Genetic Programming, follows in that direction. If the fitness landscape were convex in mathematical sense, even the simplest local search algorithm would find solution that lies very near global optimum. An slightly more sophisticated procedure would almost guarantee finding of the optimum. The space of programs is broad or even infinite, therefore exhaustive searching in it is futile. The program space can be reduced by extracting semantically unique programs to the new space. That space is computationally hard to produce, but with some constraints on program size, it is possible to be generated in finite time. The main advantage of semantically unique program space is lack of redundancy.

Chapter 4

Automatic Design of Semantically Smooth Instruction Spaces

4.1 Embeddings of program spaces

As mentioned previously in Section 3.2, many of program spaces are characterized by very low locality, because of the complex relation between program syntax and its semantics. There were many attempts to solve the problem by designing new semantically smooth genetic operators, such as semantically-aware crossover and semantic-similarity crossover, both described in [92] or geometric crossover [48]. However these methods do not redesign the program space, are not generic, and assume that programs are produced by combining two or more other programs. Therefore these operators do not apply in non-GP context.

In this thesis it is proposed to take advantage of program space reorganization in such way, that the new "optimized" space can be utilized by any search algorithm. The new approach, called *learnable embeddings of program spaces*, was firstly presented in [47]. The key idea is to design a mapping from some abstract space into target program space, such that it maximizes locality with regard to semantics of program. The outline of the slightly developed (in comparison to [47]) version of this methodology is illustrated in the Figure 4.1 as well as described below.

Let $\mathscr{P}$ be a given space of programs. Because the number of possible programs is very big, it is assumed that space is limited to semantically unique programs of given length only and denoted by $\hat{\mathscr{P}}$. Let X be a space of all programs from $\hat{\mathscr{P}}$, called *prespace*. and organized into d-dimensional toroidal hyperrectangle of integral sizes $n_1, n_2, ..., n_d$. Note that $\forall_{i=1..d}\, n_i \geq 2$, because having $n_i = 1$ for the i-th dimension degenerate it, therefore the space is one dimension smaller indeed. It is clear that prespace consists of $\prod_{i=1}^{d} n_i$ unique locations. Of course, the number of semantically unique programs may not factorize to d components, therefore the precise shape of the space must be managed in some way. The particular approach to optimize the layout of program space will be described later in Section 4.3. However, even the after prespace optimization there may still remain a few empty places (e.g. if the number of programs is a prime number). These empty places can be discarded by wrapping the last dimension of space or can be filled by randomly chosen programs from X. The random fulfillment causes redundancy, but simplifies implementation, therefore it will be used later in this master's thesis if not state otherwise.

Let N_X be a neighborhood relation in prespace X. For future considerations, let us define

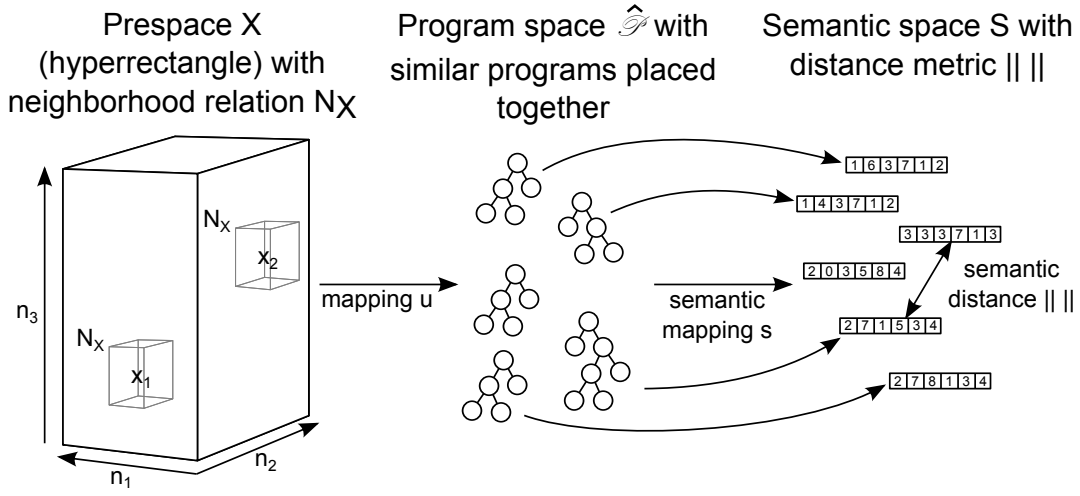

FIGURE 4.1: Learnable embeddings of program spaces.

$N_X(p)$ as a subset of programs in X such that their city-block distance from p is exactly 1. Since the prespace is toroidal, the distance has to be computed modulo n_i for particular dimension $i = 1..d$. Note that, the definition in a natural way corresponds to minimal changes in coordinates of p.

Let u be a bijective mapping from prespace X to *program space* $\hat{\mathscr{P}}$, such that $u : X \to \hat{\mathscr{P}}$ and $|X| = |\hat{\mathscr{P}}|$. To provide clear naming, $\hat{\mathscr{P}}$ will be referred as *original* program space. There are many possible ways to define u. To narrow spectrum of definitions, it is assumed that u is permutation of elements in X. Consequently u can be considered as a form of *embedding* of $\hat{\mathscr{P}}$ in X.

The objective of optimization is to find the optimal triple (N_X, X, u) that maximizes locality of $\hat{\mathscr{P}}$ with respect to the mapping u and neighborhood relation N_X. Since X and N_X are fixed, as described above, the objective boils down to learn the optimal embedding u^* of $\hat{\mathscr{P}}$ in X, such that:

$$u^* = \underset{u}{argmax}\{L(N_X, X, s \circ u)\} \tag{4.1}$$

where $s \circ u^*$ is the best locality preserving mapping from prespace X to semantic space S through program space $\hat{\mathscr{P}}$.

It is easy to see that the traveling salesman problem transforms polynomially to the probelm of finding an embedding, where u defines an order of program (city) visiting in X and $\|s(p'), s(p)\|$ (from expression 3.1) is distance between programs (cities). Since in the current problem we maximize the formula consisting distance metric in denominator, an formula defined on sum of distances is being minimized. Consequently finding of optimal u^* is a NP-hard problem. Since the computational effort required for an exhaustive search (or, e.g., branch & bound) is exponential, it can be equally well consumed to find an optimal program by enumerating all programs in the original program space, even without the described procedure.

Thus, the computation effort is too big for majority of real-world sized X. However, an important tenet of this thesis is that even suboptimal mapping provided by local search algorithm or metaheuristics can smother fitness landscape, emphasize global convexity, and reduce the search effort required to find a well-performing solution (program).

For instance, consider a simple symbolic regression problem. The task is to find an expression

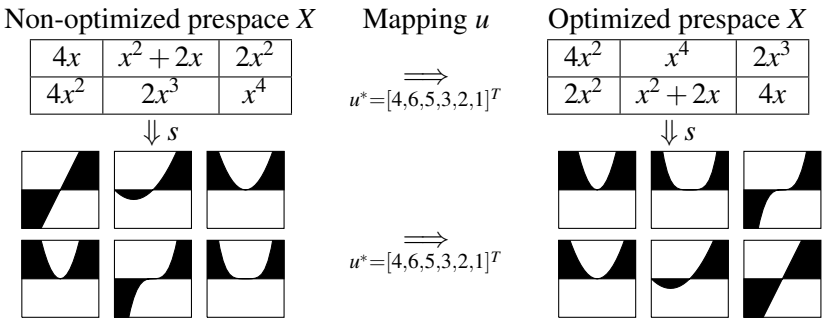

TABLE 4.1: Example of program space embedding. The left table represents non-optimized prespace X of randomly assigned, in terms of semantic similarity, programs from $\hat{\mathscr{P}}$. The mapping u transforms prespace X into optimized space by reordering of programs inside. Each chart at the bottom represents the semantics of corresponding programs plotted as function graphs in the interval $[-2,2]$. As a result of the optimization process, the optimized program space is split up into two separated groups, even-degree polynomials (the four in the left part of embedding) and odd-degree polynomials (the two in the right part). The major reason is that semantics of both $2x^3$ and $4x$ go to $+\infty$ for positive arguments and to $-\infty$ for negative values, while the analyzed even-degree polynomials go to $+\infty$ in both cases.

(program) that has specific behavior (semantics). The program space is built upon the set of nonterminals $\{+,*\}$ and only one terminal – the independent variable x. Let us limit the space of solutions to binary trees of depth equal to 3, therefore each program consists of 3 instructions and 4 leafs. Since the leafs are fixed to be the independent variable (there are no other terminals), any two programs built upon these assumptions differ only in the internal nodes. It is clear that there 8 syntactically different programs can be produced under these assumptions, but only 6 of them are semantically unique, because of the symmetry of addition and multiplication operators. The unique programs are:

1. $x+x+x+x = \mathbf{4x}$
2. $x+x+x\times x = x\times x+x+x = \mathbf{x^2+2x}$
3. $x\times x+x\times x = \mathbf{2x^2}$
4. $(x+x)\times(x+x) = \mathbf{4x^2}$
5. $(x+x)\times(x\times x) = (x\times x)\times(x+x) = \mathbf{2x^3}$
6. $x\times x\times x\times x = \mathbf{x^4}$

Let us organize the space of semantically unique expressions into 2-D, non-toroidal rectangle X with dimensions 3×2. Let us define the neighborhood $N_X(p)$ of program p as set of programs, whose positions differ from p no more than 1 under Hamming distance. For instance, in the left-hand side of Table 4.1 the neighbors of $4x$ are x^2+2x and $4x^2$. Define program p semantics $s(p)$ as vector of program values for the sequence of values of x from -2 to 2 with step 0.2 and the semantic distance as a normalized Euclidean distance between the vectors of semantics.

Firstly, let us assign the programs from $\hat{\mathscr{P}}$ randomly to locations in the rectangle, then transform the prespace X into optimized space by reordering of programs using mapping u, such that, the locality measure L will be maximized. The Table 4.1 shows example program assignment before and after applying optimized mapping u^*. The optimized mapping is the permutation of

programs, represented here as their identifiers: $u^* = [4,6,5,3,2,1]^T$. The locality of random assignment is $L = 0.69$ and the optimized one is $L = 0.76$. Maximization of locality causes semantically similar expressions to be placed together, consequently smoothing program fitness landscape and simplifies finding of program that has desired behavior. This is visible in the plots, which clearly group according to visual similarity. Note that the natural, or random in terms of semantic similarity, assignment of programs in prespace X can produce multiple local optima, therefore causing the search algorithm to more likely stuck in one of them.

In conclusion, it is worth noting, that pair of X and N_X implicitly specifies new program representation space, where program is addressed by d-tuple of coordinates that corresponds to the location in prespace X. The location in hyperrectangle is totally separated from syntax of program, unlike it is in 'standard' representation space, where composition of instructions (primitives, symbols) determines location and distance between programs. Therefore any program $p \in X$ can be syntactically unrelated with its neighbors $p' \in N_X(p)$. Moreover, the space $\hat{\mathscr{P}}$ acts as a transparent proxy between prespace X and semantic space S by quiet translation of program coordinates. Additionally, the representation maintains explicability, because every coordinates in the prespace X refer to the definition of particular program in original space $\hat{\mathscr{P}}$.

4.2 Program representation

Previously in Section 2.2, the representation of GP-like program has been described. However the form of canonical GP program is used in this paper, its structure needs to be constrained here in order to fill our task requirements. Before the representation can be described, the notion of binary decision diagram must be introduced.

4.2.1 Binary Decision Diagrams

Logic function is a function in form $f : \{0,1\}^k \to \{0,1\}$, for natural k. In more descriptive words, it is a function, which domain is k-ary vector of logic values – $true(1)$ and $false(0)$, and its codomain is single logic value. Moreover if $k = 0$, the function is constant. The logic function is often called *Boolean function,* named after its inventor George Boole, by whom was firstly described in 1854 [11]. In mathematics and computing science, there are multiple ways to symbolize a logic function. It is commonly expressed as *truth table*. The truth table consists of k columns representing input values (input vector) and one output column. The table typically contains 2^k rows, one for each combination of input variables. The value of function, for given input vector, is obtained by selecting row that matches input vector values with input columns and returning value of output column of that row. In conditional part of truth table, it is possible to introduce "don't care" value $(*)$ literal, that matches both 0 and 1 and therefore reduces number of rows, but it will not be considered here. The other representation is *Karnaugh map*, introduced in 1953 [36]. In the map, input values are split into approximately equal in cardinality groups, and put on columns and rows according to principles of Gray code. The interior of the table represents output values. Thanks to Gray code, the adjacent table cells differ only in single input value. The value of the function is obtained by treating input values as coordinates in the map and returning pointed

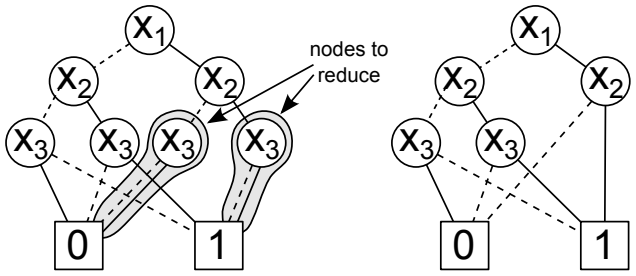

FIGURE 4.2: Binary decision diagram for expression $f = \neg x_1 \neg x_2 \neg x_3 \vee x_1 x_2 \vee x_2 x_3$. The dotted lines represent decision "0", solid lines indicate decision "1". Left: unreduced binary decision diagram, right: reduced ordered binary decision diagram.

value. The Karnaugh map is often utilized to minimization of logic functions and synthesis of logic circuits.

This master's thesis takes advantages of other representation – a binary decision diagram (BDD) [56, 90] introduced by Lee in 1959. BDD is a directed acyclic graph, with only one starting vertex (diagram root) and two terminal nodes – $true(1)$ and $false(0)$. The graph is connected. Each intermediate node, with root included, represents a single Boolean input variable and therefore is named *decision node*. The node has exactly two outgoing arcs, the first arc leads to the node referred as *low child*, the second to *high child*. The children are named after value chosen in parent node – low child represents an assignment of the parent variable to 0, high child to 1. The value of function is obtained by going along the path from the root to the terminal and selecting particular way depending on input values. The value of terminal determines value of function for given input. The example of BDD is illustrated in the Figure 4.2. As you see, the structure is special case of Genetic Programming tree, therefore BDD will be implemented as the tree. The details of implementation will be discussed later in Section 4.2.2. Now, let us introduce some modifications of binary decision diagrams.

The binary decision diagram is called *ordered (OBDD)*, if variables on all paths from root to the leafs are placed in the same order. The gaps in numbering are allowed. The good variable ordering is crucial to minimize size of BDD. Unfortunately, there has been shown that finding of optimal variable ordering is NP-hard problem [10]. Additionally, even for any constant $c > 1$ there is no polynomial time algorithm that guarantees the resulting OBDD is at most c times larger than optimal one [83].

The OBDD is called *reduced ordered binary decision diagram (ROBDD)* if and only if both of the following conditions are satisfied:

1. There is no node, which both outgoing arcs point at the same child,

2. There is no group of nodes, labeled with the same variable, where subgraphs of each node are isomorphic (e.g. have equal layout).

In order to reduce OBDD, the following procedure must be done to satisfy above conditions. For each node, that has both outgoing arcs pointing at the same child, remove the node and attach its incoming arc directly to the child. For each group of equally labeled nodes, whose subgraphs are isomorphic, replace entire group by single node along with its subgraph.

Finally, it has been proven, that there exists only one ROBDD for particular function and given variable order [12]. This property is particularly useful in logic function equivalence checking and functional technology mapping. However in the literature it common to use term BDD within the meaning of reduced ordered BDD, these two concepts will be distinguished in this master's thesis.

4.2.2 Instruction and program representations

Previously the concept of binary decision diagrams was introduced. In this section, we present a program representation involving these diagrams. The proposed program structure consists of set of subroutines composed in strictly defined way. The term 'subroutine' should be identified with procedure or even macro in ordinary program. For this paper use, routine r of depth k is defined as follows. r is an ordered, but not reduced, binary decision diagram, with k inputs referring to decision nodes. The single-bit output of the OBDD is also output of r. The routine defined above, will be called *metainstruction*, because it usually consists of sequence of only few instructions. The space of all metainstructions of depth k will be denoted by I_k. The big advantage of metainstruction definition as OBDD is that the set I_k consists only semantically unique routines. The routine is implemented as ordinary GP tree with decision variables as nonterminals and set of $\{0,1\}$ as terminals. Moreover there is only one pair of terminal objects, since they do not store any state information, thus multiple decision nodes can directly point at the same objects. Note that relation between tree depth d and OBDD depth k is given by the formula $d = k+1$. From the above definition, it concludes that total number of nodes in metainstruction of depth k is determined by formula $2^k + 1$ and cardinality of space of all possible metainstructions of depth k is given by expression:

$$|I_k| = 2^{2^{d-1}} = 2^{2^k} \tag{4.2}$$

Note that there is no need to consider metainstructions shallow than k, because semantically they are simply reduced versions of metainstructions from some subset of I_k. In other words $\forall_{l=1..k}, I_l \underset{s}{\subset} I_k$ or recursively $I_{k-1} \underset{s}{\subset} I_k$ for $k \geq 2$, where $A \underset{s}{\subset} B$ means that set of semantic equivalence classes of elements from A is contained in set of semantic equivalence classes of elements from B.

Once metainstruction has been defined, let us discuss structure of entire program. Since metainstruction takes k one-bit arguments and returns a single bit, the input vector can be treated as binary representation of an integer, but there is a problem with output value, because it is no use considering output as an integer. To work around this issue, a special *binary neural network* structure of program is proposed. The metainstructions act as neurons in that network and their depth clearly defines number of neuron inputs. The network can be constructed either from homogeneous or heterogeneous set of routines, in terms of their depth. Considering heterogeneous set, there is no problem generating network where each neuron in one layer is connected to all neurons in the next layer. Unfortunately the homogeneous network highly constraints space of available architectures. However the restriction on peer connections between neurons can be weaken, it is assumed, in order to simplify the architecture, that all layers except the output one have exactly k neurons and cardinality of the last layer remains unrestricted. Note that the connections in neural network defined in that way are not weighted due to logic characteristics of transferred signals.

Consequently the metainstruction becomes transition function of the neuron and it can ignore or notice particular input itself without using weights. The program search process will be aimed at looking for proper assignment of metainstructions for desired program characteristics. Search process will be discussed more precisely in Section 4.5. Finally, it is worth noting, that program structure as well as metainstruction definition are directly implementable in hardware, making logic circuit synthesis extremely easy.

In conclusion, the metainstruction space, although smaller than program space, still grows up rapidly with rising k. For relatively small $k = 5$, there are $2^{2^5} \approx 4.29 \times 10^9$ possible instructions, with $2^5 + 1 = 33$ nodes each. Assuming that each node utilizes 32 bytes of memory, entire space will take about $4.13TB$! Because of this observation, we are forced to sample metainstruction set for $k \geq 5$. However the sampling gives us ability to enforce arbitrary size of metainstruction space, it must be taken into account that only uniform sampling in semantic space of metainstruction preserves completeness. In this thesis, uniform sampling will be done in space of programs in order to clarify the process.

4.3 Shape optimization of metainstruction space

As mentioned previously, program space, for our purposes, is built upon set of semantically unique programs. Cardinality of this set is unknown until all programs are checked for uniqueness and cannot be effectively predicted in most cases. Once it is done, the number of semantically unique programs $|\hat{\mathscr{P}}| = |S|$ is known. Since the program space is represented here as d-dimensional hyperrectangle, where d is given, it is needed to determine adequate lengths of dimensions before filling the space, in order to accommodate all the $|S|$ programs. It is clear that there may not be such vector $n = [n_1, n_2, ..., n_d]^T$ of dimension sizes containing integral values that matches the equality $|S| = \prod_{i=1}^{d} n_i$. Therefore the number of free places must be reduced in some way. It can be done by manipulating n_i values.

For our use the program space is reduced to metainstruction space. Since metainstruction is represented here as binary decision diagram, one may seem that it is known a priori how many semantically unique metainstruction are in the space. However it is true and there has been provided formula for that (see Equation 4.2), it has been also noticed that for $k \geq 5$ decision nodes it is impossible to store entire space in memory of today's computer. Therefore the uniform sampling of the space has been proposed. Obviously the number of sampled programs can be given in advance, although it should be as big as possible in order to make the best feasible representation of entire space. Because the number of free places in the hyperrectangle grows exponentially with increasing dimensionality, it may be futile to store and process all programs in that rectangle, especially for high dimensionality. Therefore, for given d, the number of sampled programs can be set to value that is much smaller that desired hyperrectangle size.

In both cases, where number of programs is unknown a priori or the space must be sampled, the number of programs may not fit perfectly any hyperrectangle of given dimensionality. Therefore it is needed to optimize lengths of dimensions. Formally the objective is to find vector $n = [n_1, n_2, ..., n_d]^T$ of d integral values representing lengths of particular dimensions, that minimizes unused space in the hyperrectangle with regard to given cardinality of program $|\hat{\mathscr{P}}|$,

Algorithm 2 Factorization of given $|S|$ algorithm.

1. Let $n \leftarrow |S|$
2. Let $C \leftarrow \emptyset$ be vector of components
3. Let Q be the sequence of prime numbers between 2 and $\lfloor \sqrt{n} \rfloor$
4. For each $q \in Q$ (in increasing order) do
5. If n is divisible by q then
6. Add q to the component vector C
7. $n \leftarrow n/q$
8. Go to step 9
9. Repeat steps 4 – 8 until n does not change
10. If $n > 1$ then
11. Add n to the component vector C
12. Return component vector C

metainstruction $|I_k|$ or semantics space $|S|$. Consequently there are suggested four methods to solve the problem.

Firstly it is proposed to take advantage of $|S|$ *factorization* into vector C of prime components. The detailed procedure is given in the Algorithm 2. The algorithm at the beginning computes sequence of prime numbers that are greater or equal to 2 and less or equal to $\left\lfloor \sqrt{|S|} \right\rfloor$. The components greater than $\left\lfloor \sqrt{|S|} \right\rfloor$ need not to checked, because if they exist, they are implicitly tested by lower prime numbers. Let us draft a proof, if there is prime component q such that $q > \sqrt{|S|} \geq \left\lfloor \sqrt{|S|} \right\rfloor$, there must be related component $q' = \frac{|S|}{q} < \sqrt{|S|}$. If q' is prime number, the q has been already checked by smaller q', else if q' is composite number, it is built upon smaller prime numbers that indirectly have checked q already, otherwise the q is not component of $|S|$. The sequence of prime numbers can be produced by any algorithm. The currently known most efficient algorithm for that purpose is *Sieve of Atkin* [6] with $O\left(\frac{N}{log(log(N))}\right)$ time complexity, where N is the maximal number to which primes must be found. For our use, where primes are less than 1 million, the simpler *Eratosthenes Sieve* algorithm is sufficient. For comparison, its time complexity is $O(N \times log(log(N)))$.

It is worth noting, that the factorization process returns arbitrary number $|C|$ of prime components. Therefore it can be greater or less than desired d. If the number is greater, the situation is simple. In each step the repair procedure chooses two the smallest components $c_1, c_2 \in C$ and replaces them by their multiplication $C \leftarrow C \backslash \{c_1, c_2\} \cup \{c_1 \times c_2\}$. The procedure terminates when cardinality of $|C|$ is equal to desired d. In this case it is guaranteed that there is no free place left in the hyperrectangle and the sizes of dimensions are being balanced as much as possible. On the contrary, if number of returned components is less than desired d, some components must be artificially created. However it is trivial to add ones for missing values, these ones represent dimensions

that are degenerated and consequently useless. To work around the issue, the following repairing procedure is proposed. Take the maximum $c \in C$ and remove it from the component vector. Then factorize number $c+1$ into new component vector C'. Afterwards add all components $c' \in C'$ into vector C. Repeat the procedure while $|C| < d$. If $|C| > d$ then apply repairing procedure from the first case. Choice of the biggest c guarantees that there are added as less as possible free places. The procedure in one step adds $\prod_{c' \in C \setminus \{c\}} c'$ new places to the rectangle. Therefore the product is the smallest, when choosing maximum c. Moreover if $c \geq 3$, the procedure guarantees there will be added at least one component in particular step.

Someone may ask why not to add one to the $|S|$ instead of c – the number of wasted places would be increased by only one. That is true, but dramatically increases time complexity of entire procedure and does not guarantee improvement. For example consider $|S| = 9$ and $d = 3$. The factorization returns vector $C = [3,3]^T$. Factorizing $|S| = 10$ will result in $C = [2,5]^T$, as you see the numbers totally changed, but that is not a problem, there are still two components. Let us again add one to the $|S|$. Now factorization returns only one number $C = [11]^T$. It is clear that this method does not guarantee that there would be added an component in single run, moreover the deterioration in results is possible. Additionally it highly increases computation times.

The factorization method produces dimensions that are prime (or composed from low number of primes after postprocessing). This is not crucial property from program space point of view. Therefore let us involve *mathematical programming problem* that does not make any assumptions about primality. Our objective is to minimize vacancy in the hyperrectangle of given dimensionality d. The number of required places $|S|$ is given as previously. Therefore the vacancy is provided by expression $\prod_{i=1}^{d} n_i - |S|$, where n_i states for length of i-th dimension, which must be at least 2 in order to produce right dimension. Moreover let us reduce the search space by requiring the lengths of dimensions to be ordered. This constraint removes from the search space almost all permutations of dimension lengths of certain solution, leaving only permutations containing two or more equal numbers. In consequence this constraint lowers size of the search space about $d!$ times. Let us define the problem as follows:

$$(min) \prod_{i=1}^{d} n_i - |S| \tag{4.3}$$

$$\prod_{i=1}^{d} n_i \geq |S| \tag{4.4}$$

$$\underset{i=2..d}{\forall} n_{i-1} \leq n_i \tag{4.5}$$

$$\underset{i=1..d}{\forall} 2 \leq n_i \leq |S| \wedge n_i \in \mathbb{N} \tag{4.6}$$

As you see, the problem is neither linear or even quadratic, therefore it can be converted to linear form by taking an logarithm from both sides of problematic equations, but it will lead us to loss of some (maybe optimal) solutions. Consequently it is proposed to take advantage of *constraint programming (CP)* solver. The above problem definition can be directly transferred into the CP form without modification. It is important to note here, that constraint programming optimization is NP-hard problem, moreover even linear programming with integer constraints is

NP-hard. Therefore the only algorithm that guarantees finding of global optimum is exhaustive search, with all its variations, like branch & bound. Despite the computation complexity, it is possible to speed up the process hundred times. The clue is hidden in selection of proper search strategy. For example the values can be assigned to variables from the lowest to the highest in range or in reverse order, the sequence of variable assigning can be altered and additional heuristics can be used to change branching order. The implementation that takes part in this master's thesis will be described later in Section 5.1.1.

The above methods have common unwanted property of constructing unbalanced dimensions. That is, it is typical to them to produce values that vary even a hundreds orders of magnitude. The very unbalanced list of dimensions usually contains multiple low values. These low values have serious impact to the size of the program neighborhood. Consider the neighborhood relation $N(p)$ defined as Hamming distance from program p less or equal to 1. Therefore the dimension of toroidal space with length 2 causes the neighborhood to wrap around and consequently covering place of another program twice. It has effective consequences in dropping of neighborhood size and therefore may reduce performance of locality optimization algorithm. For factorization method, the cause for unbalanced results is hidden in the nature of numbers. The low components are very common, but the higher ones are rare. Although factorization looks for vector of only prime numbers. Therefore in sequence of twos and threes a value greater than one hundred may occur once or twice. In the CP problem formulated as above, orders of variables and values assigning are crucial. Unfortunately there are no easy way to define a good strategy. Therefore the following modifications of above CP problem are proposed.

Consider the most balanced, in terms of dimension length, hyperrectangle. It reduces to the hypercube, a structure where each dimension has equal integral length n. Thus the number of places in the d-dimensional hypercube can be easily computed as n^d. Consequently, for given number of programs $|S|$, the optimal continuous hypercube would have dimension of size $\sqrt[d]{|S|}$. This is the starting point for our balancing formula. Of course this is very unlikely to obtain the program number that produces integer $\sqrt[d]{|S|}$. If do that, the solution is clear, let pack program space into the d-D hypercube of dimension length $\sqrt[d]{|S|}$. Otherwise an error formula must be defined and error must be minimized. For our purposes, a *sum of square errors (SSE)* will be used. The error is defined here as difference between size of dimension n_i and its optimal, but unreal, value $\sqrt[d]{|S|}$. It is clear that the constraints must remain unchanged due to they define the allowed shape for the hyperrectangle. The CP problem is defined as follows:

$$(min) \sum_{i=1}^{d} \left(n_i - \sqrt[d]{|S|} \right)^2 \tag{4.7}$$

$$\prod_{i=1}^{d} n_i \geq |S| \tag{4.8}$$

$$\underset{i=2..d}{\forall} n_{i-1} \leq n_i \tag{4.9}$$

$$\underset{i=1..d}{\forall} 2 \leq n_i \leq |S| \wedge n_i \in \mathbb{N} \tag{4.10}$$

The above definition leads CP solver to leave many places empty. Currently it is clear that

there need to be taken into account both dimension balancing and minimization of wasted space. Therefore the problem turns into multiobjective optimization problem. There are multiple of ways to combine optimization objectives. The method that guarantees there is no one objective that outranks any other, is search for *Pareto non-dominated* [93] solutions, which lie on the curve called *Pareto front*. The solution s_1 is non-dominated in Pareto sense if and only if there is no other solution s_2 that is at least as good as s_1 on each criterion and is strictly better on at least one criterion. However the method is impartial, the Pareto front contains plenty of solutions that are incomparable. It is not good feature for automatic processing of solutions, because there is no way to select the best one. There were proposed bunch of other methods for aggregating objectives, such as Lorenz dominance, outranking relation [79] or utility function [84]. For our purposes, we make use of the last one, due to its simplicity and easiness for choosing the most valuable solutions. The utility function tends to assign the highest value only to the one solution. It is very unlikely that two different solutions got the same ranking. However if it happen, the arbitrary decision must be made, but probability of the situation converges to the zero.

In order to construct the utility function, it is proposed to make a weighted sum of objective functions from two previous CP problems. The $\alpha \in [0,1]$ coefficient is given to the first formula, that minimizes waste space. On the other hand, the $\frac{1-\alpha}{d}$ is given to the second one, that minimizes sum of square errors between length of the resulting dimensions and the optimal one. The divisor d is attached to the second coefficient, because both formulas usually differ in an order of magnitude, moreover the divisor turns the SSE into *average square error (SE)*. The new CP problem is defined below:

$$(min)\ \alpha\left(\prod_{i=1}^{d} n_i - |S|\right) + (1-\alpha)\frac{1}{d}\sum_{i=1}^{d}\left(n_i - \sqrt[d]{|S|}\right)^2 \tag{4.11}$$

$$\prod_{i=1}^{d} n_i \geq |S| \tag{4.12}$$

$$\underset{i=2..d}{\forall} n_{i-1} \leq n_i \tag{4.13}$$

$$\underset{i=1..d}{\forall} 2 \leq n_i \leq |S| \wedge n_i \in \mathbb{N} \tag{4.14}$$

Note that the α coefficient gives us the ability to manipulate weight of the both criterions. The $\alpha = 1$ turns the last CP problem into first problem (Equation 4.3) and $\alpha = 0$ into the second one (Equation 4.7). Note that the scaling factor $\frac{1}{d}$ does not have any influence to the optimization direction and results. For later use, let us arbitrary set $\alpha = 0.5$, with regard to the notes in the following paragraph.

In conclusion, the number of programs in the considered program spaces is either unknown a priori or the space is too large to store it in memory of today's computer. Therefore in both cases it is required to build the space upon arbitrary given set of programs. This arbitrary number of programs may not, especially when it is a prime number, fit any d-dimensional hypercube. Therefore, there is a need for shape optimization with regard to minimization of wasted places and dimension balancing. In this section, there were proposed four methods to comply this task. The detailed performance comparison of all four shape optimization methods as well as for different

Algorithm 3 Steepest algorithm for embedding optimization.

1. Let u be a random permutation of metainstructions from I_k
2. Let *bestRunJ*, *bestRunGain*, *prevL* be an auxiliary variables
3. For each metainstruction $i \in I_k$ do
4. $bestRunGain \leftarrow 0$
5. For each metainstruction $j \in N(i)$ do
6. $prevL \leftarrow u[i] + u[j]$
7. Swap metainstructions $u[i]$ and $u[j]$
8. If $l(u[i]) + l(u[j]) - prevL > bestRunGain$ then
9. $bestRunGain \leftarrow l(u[i]) + l(u[j]) - prevL$
10. $bestRunJ \leftarrow j$
11. Revert metainstructions $u[i]$ and $u[j]$
12. If $bestRunGain > 0$ then
13. Swap metainstructions $u[i]$ and $u[bestRunJ]$
14. Go to step 4
15. Else
16. Go to step 3
17. Return u

α coefficients can be found in Section 5.2.

4.4 Locality optimization of metainstruction space

Previously there has been discussed the learnable embedding of program spaces. The embedding in its assumptions transforms original program space into optimized one with regard to the semantic locality. However the principle defines optimization objective, it does not indicate the way to achieve that. Moreover we have proven that the problem is NP-hard. There are plenty of optimization algorithms suitable for this task, starting from simple local search, through metaheuristics and evolutionary algorithms, ending with branch & bound and exhaustive search. Note that only the last group guarantees finding of the optimal embedding, but computational complexity of the group's algorithms is unacceptable for practical embedding sizes. However for our use, the program space is reduced to space of metainstructions, the reduction does not result in loss of generality. The same optimization algorithms, as for program space, can take part here.

It is proposed to take advantage of *steepest* algorithm for optimization of embedding locality task. The algorithm belongs to the class of local search heuristics. All algorithms from that class starts from a given (or random) solution, analyze its neighborhood and then advances to the better

position in it or terminates if there is no such position. The way how the procedure chooses the neighboring solution depends on algorithm itself. The steepest algorithm altered to our optimization task is shown in Algorithm 3. It starts from a random embedding u of metainstruction space I_k in prespace X. The embedding is represented as a permutation of metainstruction representatives indeed. Since the permutation is single-dimensional structure and the space is multidimensional, there is a need to apply an arbitrary ordering of linear addresses of places in the space. In order to convert d-dimensional address into linear one, each position a in d-tuple is multiplied by product of lengths of previous hyperrectangle dimensions counting from right to left ($\prod_{i=a+1}^{d} n_i$) and then the products are sum up together $linear = \sum_{a=1}^{d} address_a \times \prod_{i=a+1}^{d} n_i$. Therefore the address is encoded in Big Endian manner. The address converting routine can be computed in $O(d)$ time involving Horner scheme.

The steepest algorithm checks whole 2-swap neighborhood of current solution and then moves to the best found neighbor or terminates if there is no one with better locality value. Because the entire space locality L (see Equation 3.3) is defined additively, the optimization process can be decomposed, with no change in search strategy, into sequence of neighborhood locality l optimization tasks for each metainstruction r separately. Consequently there is no need to globally recalculate L and the search process becomes sped up multiple times. Moreover the algorithm is made effective even for very big program spaces. Since the computation of locality in neighborhood $l(N_X, r, s \circ u)$ for single metainstruction r requires $O(|N_X|)$ time, the full scan of all 2-swaps for single metainstruction requires $O(|I_k|)$ operations, there are j improvements in worst case for each metainstruction, and there are $|I_k|$ metainstructions, the algorithm complexity is $O(j|I_k|^2|N_X|)$. It is worth noting, that the most resource demanding part of the algorithm is calculation of semantic distance between metainstructions. Therefore it is crucial to cache the distances in order to lower the computation times. However the cache lowers computation times significantly, it requires additional $O(|I_k|^2)$ bits of memory. It can be effectively implemented as half of the distance matrix, although the metainstruction addresses must be given in arbitrary order and swapped if the order is wrong.

In this section, there was proposed the way to optimize program space with regard to locality measure L. It was also noted that metainstruction space optimization task can be directly transformed into program space optimization problem with no change in its definition. Additionally there was proposed the local search algorithm to suboptimally solve this problem. Empirical analysis of the proposed approach will be discussed later in Section 5.3.

4.5 Search in optimized metainstruction space

Previously there have been introduced the concept of learnable embeddings of program spaces and the way to optimize this space with respect to the semantic locality. Currently it is time to discuss the practical applications of this approach. As mentioned previously, for this paper scope, the program space is reduced to space of metainstructions. The metainstruction space optimization directly highers semantics – distance correlation between metainstructions and in consequence indirectly highers the fitness – distance correlation between entire programs. One may ask why not to optimize program space instead. The answer is in the giant cardinality of the space. Therefore it is expected that indirect improvement of the locality in program space can be

utilized in order to improve the efficiency of space search algorithm. However smoothness of the space should improve performance of any non-random search algorithm, it is proposed to guide the process, as discussed below, to unlock the full potential of the method.

In order to apply current methodology to the world of Genetic Programming, it is proposed to take advantage of successful GP algorithm scheme as search procedure. This decision requires some preliminary work to adapt the GP concept to use features of optimized space. Firstly, there was in Section 4.2.2 mentioned, that the program has binary neural network structure. Each neuron with k inputs is implemented as a metainstruction of depth k. Therefore first layer of the network accepts whole input vector of bits and last layer returns output of entire program. The network can be constructed from either homogeneous or heterogeneous set of metainstructions, nonetheless each of them belongs to a space that has been optimized. If the network is heterogeneous, metainstruction sets must optimized separately due to different lengths of their semantics. The standard GP crossover and mutation operators may be ineffective to this program specification, because they do not pay attention to the instruction coordinates in the optimized space. Consequently it is proposed to design new simple genetic operators for both crossover and mutation.

The both operators work in per – metainstruction manner. It means that they replace whole neuron in the network, although the way to achieve that differs in both cases. Let us define the crossover operator. It takes two parent programs p_1, p_2 as inputs and for each network element it draws with certain probability $\beta_c \in [0,1]$ whether to interbreed the neuron r_1 of one parent with corresponding neuron r_2 of the second parent or not. The operation of mixing neurons takes into account vectors of their d-dimensional coordinates in optimized space of metainstructions. Note that the neurons belongs to the same space, since they must have the same depth. However it is intuitive to calculate average position between both parent's metainstructions and then choose the indicated routine as mixture of them, the average of coordinates favors metainstructions located in the center of the metainstruction space over the outside ones. Moreover it does not take advantage of toroidal nature of the space. The problem is illustrated in the Figure 4.3. To cope with the issue, it is proposed to introduce slight modifications to the averaging principle. To describe these modifications, let us move first parent subtree r_1 in sequence by each combination of lengths of space dimensions in both directions, forward and backward. In each step compute the Euclidean distance between the moved metainstruction r_1 and the second unmoved metainstruction r_2. Finally select position that minimizes the distance and choose average between that position and metainstruction r_2 as new neuron. To clear naming, let us call this method *toroidal crossover*. Note that, the crossover defined in this way works similarly to uniform crossover, because it interbreeds program elements separately with certain probability. Therefore it preserves position of genes and maintains uniform distribution of children.

Once there have been defined crossover operator, let us define the mutation one. It, similarly to the above crossover method, iterates over elements of the network and chooses with certain probability $\beta_m \in [0,1]$ whether to mutate the metainstruction or not. It is proposed to replace the metainstruction by randomly chosen one from the same metainstruction space. Because mutation defined in that way is very destructive, it is proposed to set β_m probability to low values. Neverthless of its damaging behavior, the characteristics can be very helpful in escaping from local optima and premature convergence of search algorithm.

In conclusion, there was proposed the way to involve successful GP scheme to build programs

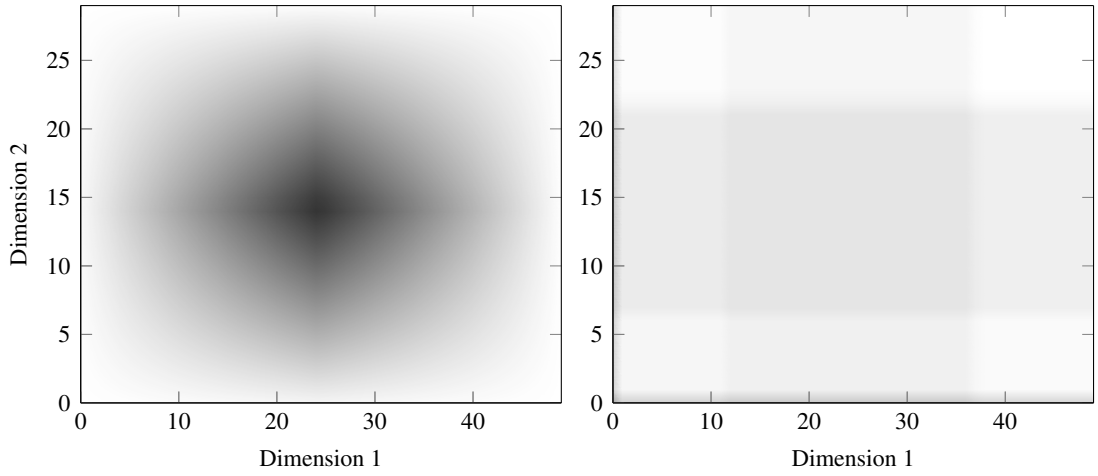

FIGURE 4.3: Probability distribution of metainstruction selection for two crossover types in 2-dimensional toroidal hyperrectangle of dimensions $\{50, 30\}$. The darker areas indicate higher probability, the brighter lower probability. Left plot corresponds to the simple averaging of coordinates of parent's subtrees, the right one is related to toroidal crossover.

by searching through optimized metainstruction space for proper instruction coordinates. However proposed program representation was especially designed for task of logic function synthesis, the metainstruction, or more generally binary decision diagram, can take part in many other applications, such as decision supporting, classification and industrial process or robot controlling. The general GP scheme does not change in all these applications, moreover the proposed genetic operators may give rise to new ones aimed at different structure of program. The search performance will be analysed later in Section 5.4.

Chapter 5

Empirical Results

5.1 The experiment

While Chapter 4 introduced the concept of automatic design of semantically smooth instruction spaces for Genetic Programming, in this chapter, an experiment illustrating the entire approach on nontrivial-scale problems will be performed and described. Before the experiment, there is a need for a few preliminary steps. Firstly, the set of metainstructions must be chosen. For our purposes, three instruction sets built upon OBDDs of depth $3, 4$ and 5 will take part in the experiment. Then, the semantic vectors of all metainstructions will be computed, in order to calculate semantic distance between each pair of them in the next step. Finally, when the distance matrix is ready, we will launch the main experiment.

The space of metainstructions will be packed into d-dimensional hyperrectangle, as mentioned in Section 4.3. Because the number of metainstructions is arbitrary given, there may not exist a hyperrectangle such that the metainstruction space fits it perfectly (i.e., of the same cardinality). Therefore, there is a need for optimization of lengths of particular space dimensions aimed at minimizing the unused space. Comparison of four methods solving this task can be found in Section 5.2.

Once the metainstruction space got its shape, the main part of the experiment begins. It involves smoothing of the space by maximization of the semantic locality metric. The direct consequence of the optimization is a rearrangement of metainstruction positions such that the semantically similar metainstructions lie in compact clusters, while semantically different metainstructions are placed far away from each other. Since the optimization problem is NP-hard, using an exhaustive search algorithm is futile. Therefore the analysis of steepest local search heuristics will be discussed later in Section 5.3.

Finally, the optimized metainstruction space takes part in a search for a program built upon this set of instructions. Since the optimization process tends to increase fitness-distance correlation between instructions, any non-random search algorithm that utilizes an optimized (smooth) instruction space should achieve better results than the same algorithm working on non-optimized space. It is worth noting, that the space embedding must be done only once and then the optimized space can be used in multiple applications with no need for recomputation. Therefore it should be considered as advantageous to spend additional time under space optimization process in order to achieve better results. Performance comparison of a Genetic Programming algorithm solving

logic problems using both optimized and non-optimized metainstruction spaces will be analyzed in Section 5.4.

5.1.1 Implementation

The experimental software for this paper has been written in *Java*, involving Eclipse as the development environment. The choice of Java was dictated by availability of a very good framework called *Evolutionary Computation in Java (ECJ)* [82] for this platform. The ECJ comes with a wide range of GP tools like automatic, parametrized GP tree builder, variety of genetic operators, availability of typed and non-typed instruction sets, and support for multiple trees per individual, in particular for automatically defined functions [44]. Moreover the framework gives us ability to define an entire experiment in easy-to-write parameter files. The file contains information about the instruction set to be used, the constraints for the instructions, particularly helpful with strongly-typed GP, and the definition of evolution pipeline. The pipeline defines control flow of the whole evolution process, making it possible to redefine and extend the steps of the canonical GP algorithm. Additionally, each part of the pipeline is fully configurable and replaceable. Furthermore, the same format as ECJ parameter files has been used to store computation results of the experiments.

The task of shape optimization of metainstruction space involves solving constraint programming (CP) problems, therefore from the wide repertoire of CP solvers for Java, the *CHOCO Solver* [23] has been chosen. The CHOCO solver distinguishes itself from the others in supporting floating-point constants and variables, which is a unique feature among open source and freeware CP solvers. This feature is crucial for our purposes due to definition of the dimension balancing problem (see Equation 4.7) and the weighted sum problem (Equation 4.11). Moreover, the solver provides an ability to define an entire strategy of solution search process and limits to be imposed on the execution time as well as on the number of analyzed nodes or solutions.

Since statistical analysis requires multiple runs of the algorithm with different parameters, several *PowerShell* [64] scrips have been used in order to automate the job. Though PowerShell was especially developed to perform administrative tasks, it provides convenient way to manipulate different kinds of files, like XML, CSV and obviously plain text. Additionally PowerShell gives us unlimited access to full .NET Framework platform, in particular to its useful data structures and tools. All this functionality makes PowerShell perfect tool for automation of experimental process and initial analysis of the acquired data.

5.1.2 Runtime environment

All trials were carried out on a single computer equipped with a dual-core Intel Core i5 650 processor clocked at 3.6GHz. The 64-bit Oracle Java Virtual Machine version 6 update 24 running on Windows 7 x64 was used as execution environment. The JVM process was operating at high priority.

5.2 Shape optimization of metainstruction space

The optimization of program space shape is the first step of methodology described in this master's thesis. It involves finding an optimal vector of dimension lengths for a given number of dimensions$d$ such that it decreases the number of unused places or balance of space dimensions. In Section 4.3, four methods were proposed for this task, each with other characteristics. These include: factorization and three constraint programming problems: minimization of free space, dimension balancing and weighted sum of both previous.

The minimization of free space and dimension balancing CP are focused on a certain objective, therefore these methods produce optimal results with respect to their goals. On the other hand the factorization method is heuristic, because it does not guarantee finding an optimum in terms of any considered objective. Since the weighted sum method is also a variant of CP, its solution is optimal, although not in terms of any single goal, but their composition.

The above separation of method classes reflects in the computation times. A run of factorization method lasts less than $10^{-4}s$ on the test machine. On the other hand, the CP algorithms need from a few seconds up to a few hours to find the optimum. Therefore, their runs were limited in time to $10s$ for both free space minimization and dimension balancing, and $15s$ for weighted sum, because of the more complicated objective function. Only the runs that found an optimum within the assumed time limit are included in the following comparison. The comparison is provided for dimensionality $d \in \{2,4,6\}$ and cardinality of program space $|S| \in [1000,5000]$. See Figure 5.1 for the charts corresponding to the first three methods and Figure 5.2 for the analysis for the fourth method, i.e., weighted sum analysis.

From what can be seen in the charts, the factorization and minimization of free space have both strong preference to minimization of unused places. However the minimization obviously produces optimal results for each dimensionality, the factorization tends to higher unused space with rising dimensionality and program number. Note that, for $d = 2$ the factorization can produce at most 1 empty place. It happens if and only if $|S|$ is a prime number. Consequently factorization has equal results as free space minimization for $d = 2$. However for higher d the factorization results are worse than minimization results, the computation times are few orders of magnitude smaller, making factorization the good alternative for free space minimization method. The similar behavior of the both methods can be also seen in charts of sum of square errors. The results of both approaches compose to characteristic, overlapping curves. The curves can be approximated by square function of the longest dimension produced by both of the methods. However the general characteristics is similar, in case where factorization produces too low amount of components, the repairing procedure removes the higher component, causing lowering of the error. Thus the factorization can be considered as a bit better approach, with regard to sum of square errors, than minimization of free space. Obviously solving of the dimension balancing problem produces optimal results on this objective. Despite of good results, the tests have shown that the dimension balancing method has the highest computation times of all three methods described here, what reflects in timing out and therefore low number of chart points for this method. Since the method optimizes sum of square errors, the results are very bad in terms of unused space. The characteristic lines can be found in the free space charts. The starting (left) point of each line corresponds to the situation where $|S|-1$ matches perfectly some hyperrectangle with dimensions differing at

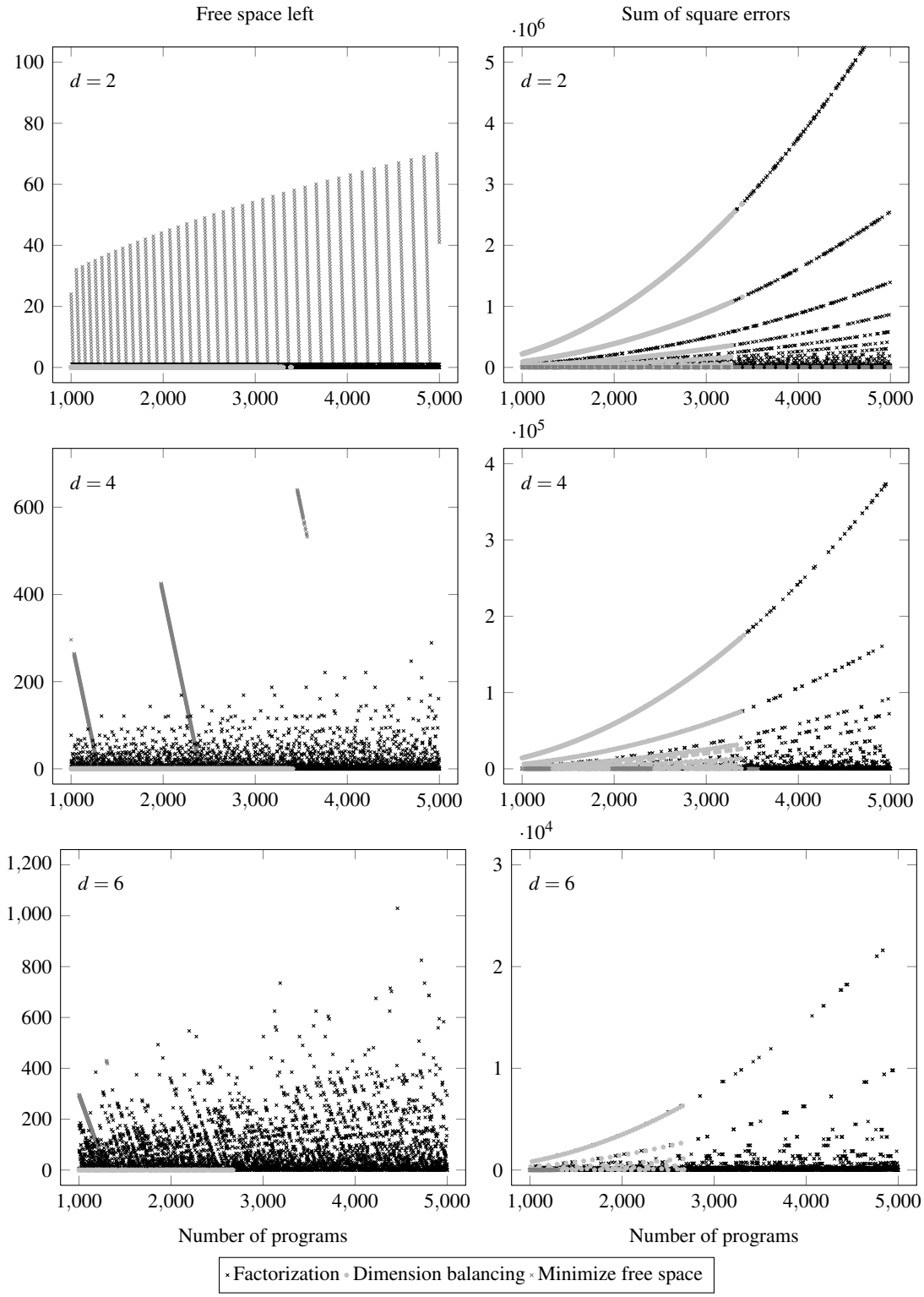

FIGURE 5.1: Comparison of hyperrectangle shape optimization methods: factorization, free space minimization and dimension balancing. The left column shows free space characteristics of each method, the right one refers to the sum of square errors between generated dimensions and optimal hypercube dimension size $\sqrt[d]{|S|}$, which usually is not integral value. Each row represents different dimensionality $d \in \{2,4,6\}$ of the space.

most in 1 value with each other, but $|S|$ is too big for that hyperrectangle, therefore the method produces a bit bigger hyperrectangle by adding 1 to one of dimensions, consequently generating about $|S|^{\frac{d-1}{d}}$ new places. Generally amount of wasted space increases and sum of square errors decreases with rising dimensionality for all methods, except the optimal one for particular criterion. Moreover average value of both criteria increases with rising number of programs.

The comparison of weighted sum CP problem for different weights $\alpha \in \{0.2, 0.5, 0.8\}$, space dimensionality $d \in \{2, 4, 6\}$ and number of programs $|S| \in [1000, 4000]$ is shown in the Figure 5.2. It is clear that for $\alpha = 0.2$ the optimization process puts pressure on dimension balancing, on the other hand for $\alpha = 0.8$ the process favors lowering of wasted space. Therefore $\alpha = 0.5$ can be interpreted as neutral value, although values of both criteria have different sizes, and additionally they change with rising dimensionality d in opposite directions, consequently the neutral point can be moved from high α for low dimensionality d to low α for high d. The above considerations are reflected in computation results in the charts. For $\alpha = 0.2$ wasted space is the highest and sum of square errors is the lowest of all methods for all $d \in \{2, 4, 6\}$, on the other hand for $\alpha = 0.8$ the unused space is the smallest, but error is the biggest. Moreover for $\alpha = 0.2$ and $d = 2$ dimensions the method is clearly dominated by sum of square errors minimization. On the contrary for $\alpha = 0.5$ and $\alpha = 0.8$ and $d = 2$ dimensions, the results are pretty the same and causes small values of both measures, but no one criterion significantly dominates. Therefore the true neutral point must lay somewhere between these two α values. Moreover for $\alpha = 1.0$ the method transforms into free space minimization, consequently causing lowering of free space and increasing of error. For higher dimensionality the results of all methods more clearly separate from each other. In conclusion, for the testing range of parameters, the best weight α that does not favor any criterion over the remaining is $\alpha \approx 0.5$.

It is worth noting that both measures, free space and sum of square errors, are much smaller in case of weighted sum compared to the single criterion in case when second criterion is optimized separately. Moreover the results are much better than these produced by factorization. Of course, these results are paid by the much more computation effort. Therefore the computation times of the weighted sum method can be unacceptable for greater program spaces. However the time cost, a suboptimal results can be obtained by breaking computation after desired time limit. These results can be also better than factorization outputs for 'high enough' time limits.

Finally, the experiment compared four methods for shape optimization of program space. The fastest one is factorization of program number into vector of numbers. The method operates heuristically, therefore it does not put pressure on any criterion, but quickly produces 'good enough' solutions even for very big space cardinalities. There were also analyzed two constraint programming problems, one that minimizes the free space and one that minimizes sum of square errors between generated hyperrectangle dimensions and optimal, but unreal, hypercube dimensions. However both CP problems guarantee finding of global optimum for certain criterion, the results are not satisfactory, because the criteria are conflicting. Consequently good value on the first criterion corresponds to the bad value on the second one, and vice versa. This leads us to combine both CP problems into weighted sum. The analysis of weights has shown that for range of small dimensionalities, the results are acceptable for equal weights of both criteria. Moreover the values of both measures are much smaller for the combined problem, than for factorization. Therefore the weighted sum method is recommended for future use.

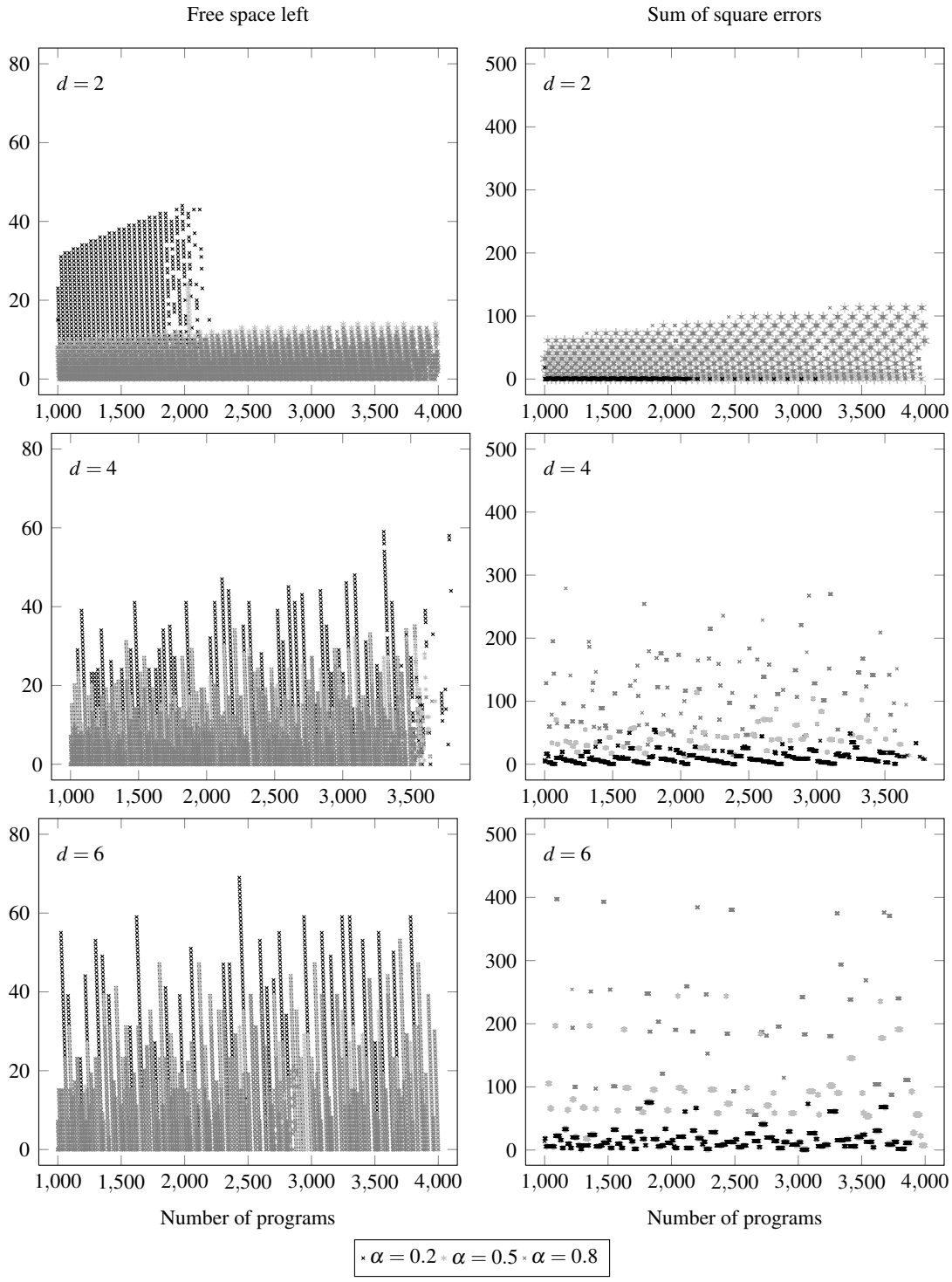

FIGURE 5.2: Comparison of weighted sum method for hyperrectangle shape optimization problem for different weights $\alpha \in \{0.2, 0.5, 0.8\}$. The left column of charts shows amount of wasted space produced by each method, the charts to the right correspond to sum of square errors between optimal hypercube with dimension length $\sqrt[d]{|S|}$, which usually is not integral value. Each row represents different dimensionality $d \in \{2, 4, 6\}$ of the program space.

5.3 Locality optimization of metainstruction space

The previous analysis was focused on preliminary step, in which the shape of metainstruction space is prepared. The main step, which is optimization of the space with regard to the semantic locality, will be studied here. As shape optimization routine, the weighted sum CP problem with $\alpha = 0.5$ has been chosen due to its advantageous properties to minimize both number of unused places and imbalance of dimensions. In each run, the shape optimization routine found the global optimum in given time limit of 60 seconds. The locality optimization analysis is made for $d \in [2,10]$ dimensions of metainstruction space and $k \in \{3,4,5\}$ decision nodes for each metainstruction. Since locality optimization problem is NP-hard, as there has been proven in Section 4.1, the usage of exhaustive search algorithm is futile, therefore the local search steepest algorithm (see Section 4.4) is involved here. The locality L of entire metainstruction space is computed according to the Equation 3.3. The measure makes use of metainstruction neighborhood relation N_X. In the following analysis, the neighborhood relation of particular metainstruction r is defined as set of all instructions, whose positions in the hyperrectangle vary only in 1 in terms of Hamming distance. Note that, the size of the neighborhood defined in that way is dependent on dimensionality d of the metainstruction space and it is given by equation $|N_X(r)| = 2d$. The Hamming distance is also involved in measurement of distance between metainstructions, because their semantics are vectors of zeros and ones. As mentioned previously, the L is defined additively, therefore the computation time of the algorithm can be effectively reduced by calculating only local changes of locality l in metainstruction neighborhood according to the Equation 3.1. Even making use of this optimization, the evaluation of objective function is the most computationally demanding part of the algorithm. The calculation time can be reduced even more by caching of semantic distances between metainstructions. The cache is implemented as half of distance matrix, consequently its size depends quadratically on number of programs. The exact size of the matrix is given by statement $\frac{|I_k|\times(|I_k|-1)}{2} \times 4B$. The matrix rapidly grows in memory with increasing number of metainstructions and their number rises exponentially with increasing number of decision nodes, in consequence, requiring us to sample the metainstruction space. Consider $k = 4$ decision nodes, the distance matrix would take $\frac{2^{2^4}\times(2^{2^4}-1)}{2} \times 4B \approx 8GB$, while entire process requires about $10GB$ of memory. Due to memory limitation of test computer, we are forced to sample the space for $k \geq 4$ decision nodes. For future use, let us denote $\hat{I}_k$ as sampled part of entire metainstruction space I_k. The sample sizes for considered k are shown in the Table 5.1. The table additionally shows number of available places in generated hyperrectangle. As you can see the numbers produced by weighted sum shape optimization method are very satisfactory, since the maximum percentage of wasted space does not exceed 0.58% in each case, except $k = 3$ and $d \geq 9$. In the mentioned case, the metainstruction space is clearly redundant, since there is no vector of $d \geq 9$ integer numbers ≥ 2 that multiplies to 256. Therefore for high d the space is filled by multiple copies of original space.

Computation effort of locality optimization routine is presented in the Table 5.2. The effort is given in terms of number of objective function evaluations. General observations have shown that cardinality of the metainstruction space is major factor that influences the effort. Deeper inspection has demonstrated that the rising hyperrectangle dimensionality only slightly increases the computation effort. The effort rises significantly only if the space becomes drastically redundant. The redundancy here comes hand in hand with rising cardinality of the space, therefore the

k	Number of metainstructions		Number of available places in hyperrectangle								
	Total	Sampled	$d=2$	$d=3$	$d=4$	$d=5$	$d=6$	$d=7$	$d=8$	$d=9$	$d=10$
3	256	256 (100%)	256	256	256	256	256	256	256	512*	1024*
4	65536	34406 (52.5%)	34408	34410	34425	34425	34425	34425	34496	34496	34560
5	4.29×10^9	34359 (0.0008%)	34368	34410	34391	34398	34375	34398	34496	34496	34560

TABLE 5.1: Total and sampled cardinality of space of semantically unique metainstructions for $k = \{3,4,5\}$ decision nodes. Numbers in braces refer to the percentage of total number of instructions. Right part of the table refers to numbers of places for metainstructions in generated hyperrectangle, depending on space dimensionality $d \in [2,10]$.

k	$\lvert\hat{I}_k\rvert$	Computation effort								
		$d=2$	$d=3$	$d=4$	$d=5$	$d=6$	$d=7$	$d=8$	$d=9$	$d=10$
3	256	1.5×10^5	1.6×10^5	1.6×10^5	1.5×10^5	1.6×10^5	1.3×10^5	1.6×10^5	$5.9 \times 10^{5*}$	$2.4 \times 10^{6*}$
4	34406	3.0×10^9	3.2×10^9	3.2×10^9	3.4×10^9	3.6×10^9	3.7×10^9	3.4×10^9	3.4×10^9	3.4×10^9
5	34359	2.6×10^9	2.7×10^9	2.8×10^9	2.8×10^9	2.8×10^9	2.7×10^9	2.8×10^9	2.8×10^9	2.8×10^9

TABLE 5.2: Computation effort of steepest algorithm for given number of decision nodes k, cardinality of sampled metainstruction space $|\hat{I}_k|$ and space dimensionality d. The effort is provided as number of objective function evaluations.

$k \backslash d$	2	3	4	5	6	7	8	9	10
3	0.271	0.270	0.269	0.266	0.267	0.268	0.268	0.267*	0.271*
4	0.269	0.269	0.270	0.269	0.270	0.270	0.270	0.270	0.270
5	0.270	0.269	0.269	0.270	0.270	0.269	0.270	0.270	0.269

TABLE 5.3: Locality L_0 of metainstruction space produced by random assignment of metainstructions to positions in the space. The comparison is made for $k \in \{3,4,5\}$ decision nodes and $d \in [2,10]$ dimensions.
*– the metainstruction space is redundant, values left for comparison only.

implicit growth of the cardinality is true factor that makes the computation more difficult.

The Table 5.3 shows locality L of entire metainstruction space for hyperrectangle produced by random assignment of instructions to the empty places in the space. The random assignment becomes start point for locality maximization algorithm, therefore it is referred as L_0. All values oscillate around 0.269. It may be an effect of the standardization of semantic distance, according to the Equation 3.2, which recalculates average distance to 3 and standard deviation to 1. The random assignment algorithm commonly tends to have weak results, because it does not take into account nature of the problem or search space. However the random assignment sometimes creates very promising results, the same algorithm can produce very unsatisfactory solutions. Thus the common results converge to the average of available score. Assuming normal distribution of semantic distances and above parameters, the average score is given by equation $\bar{L} = \int_0^\infty \frac{1}{\sqrt{2\pi}} e^{\frac{-(x-3)^2}{2}} \frac{1}{1+x} dx \approx 0.268$, what explains well the obtained results.

The results produced by local search algorithm are very promising. Optimized locality L^* values varies from 0.385, up to 0.727 and they are from 1.421 to 2.697 times better than randomly generated solutions. The detailed values are shown in the Tables 5.4 and 5.5. It is worth noting, that the algorithm achieves better results for small dimensionality of the space. This observation may be an effect of interplay between two factors. As neighborhood size of metainstruction rises with space dimensionality, the search algorithm has more opportunities to swap two instructions, and consequently it can explore greater area of the search space. On the contrary, the higher neighborhood introduces more interactions between metainstructions, all of which contribute to

$k\backslash d$	2	3	4	5	6	7	8	9	10
3	**0.441**	0.412	0.382	0.396	0.392	0.385	0.385	0.415*	0.444*
4	**0.727**	0.638	0.577	0.542	0.541	0.527	0.555	0.557	0.507
5	**0.664**	0.577	0.524	0.489	0.462	0.464	0.494	0.491	0.456

TABLE 5.4: Optimized locality L^* of entire metainstruction space for $k \in \{3,4,5\}$ decision nodes and $d \in [2,10]$ dimensions of the hyperrectangle. The best value for each k is bolded.

$k\backslash d$	2	3	4	5	6	7	8	9	10
3	**1.631**	1.527	1.421	1.486	1.486	1.434	1.437	1.554*	1.637*
4	**2.697**	2.370	2.141	2.011	2.004	1.955	2.060	2.067	1.881
5	**2.462**	2.141	1.947	1.810	1.713	1.721	1.883	1.821	1.692

TABLE 5.5: Locality improvement ratio of optimized space, given by equation $\frac{L^*}{L_0}$. The best value for each k is bolded.
*– the metainstruction space is redundant, values left for comparison only.

the objective function L, consequently making the optimization process harder. However the results incline us to hypothesize, that the second factor dominates over the first one, the phenomenon influences distribution of metainstruction semantics in the semantics space S, making the problem more complex in general. Let us leave it open for now. One may point out that the optimized locality L^* increased for $k = 3$ decision nodes and $d \geq 9$ dimensions. What's more for $d = 10$ the L^* is the greatest of all runs. Although it must be said that the metainstruction space is drastically redundant in these both cases. Therefore the gain in objective function is paid by much higher computation effort. Since the results are very similar to these of low dimensions, the additional efforts may be considered as being wasted here. They can be more effectively utilized by repetitions of steepest algorithm starting from another points. Note that, good results of these two cases may be quite accidental here and need future investigations.

Let us analyze how embedding of metainstruction space influences geometricity of the space. Geometricity is property of the space indicating how much instruction coordinates in the program space correlate to its coordinates in the semantics space. In order to do that, let us introduce the following geometricity metric:

$$G = \frac{2}{(|I_k|(|I_k|-1))} \sum_{i=2}^{|I_k|} \sum_{j=1}^{i-1} \frac{||s(r_i) - s(r_j)||}{||s(r_i) - s(r)|| + ||s(r_j) - s(r)||} \tag{5.1}$$

where I_k is metainstruction space, $s(r_i)$ is semantic vector of instruction $r_i \in I_k$ and $s(r)$ is semantics of instruction located in center between instructions r_i and r_j in coordinates of metainstruction space. The metric is directly connected to the triangle inequality. For each group of semantics $\{s(r_i), s(r_j), s(r)\}$, it measures how close is triangle based on given semantics to being degenerated. The closer it is, the semantics $s(r)$ is nearer to the middle of segment between $s(r_i)$ and $s(r_j)$. Thus, the degenerated triangle indicates perfect geometricity of that group. The coefficient at the beginning of the equation acts as averaging operator. Codomain of the metric is $[0,1]$, 1 is achieved only if all triangles produced from semantics are degenerated, or in other words, the space is perfectly geometrical, 0 is returned if all semantics are the same (there is only one semantic). Let us use standardized Hamming distance with average of 3 and standard deviation of

$k\backslash d$	2	3	4	5	6	7	8	9	10
3	0.541	0.557	0.600	0.610	0.616	0.620	0.623	0.599*	0.579*
	0.593	0.665	0.698	0.675	0.689	0.689	0.722	0.675	0.660
4	0.531	0.531	0.531	0.532	0.533	0.537	0.534	0.536	0.537
	0.531	0.534	0.543	0.558	0.560	0.572	0.547	0.565	0.568
5	0.532	0.532	0.532	0.533	0.534	0.536	0.536	0.537	0.538
	0.533	0.534	0.538	0.548	0.557	0.548	0.555	0.557	0.559

TABLE 5.6: Geometricity values of instruction spaces depending on depth of metainstruction k and dimensionality d of the metainstruction space. The first value represents geometricity of non-optimized space, the second of optimized space.
*– the metainstruction space is redundant, values left for comparison only.

1 to measure differences between semantics, because of their binary nature.

The Table 5.6 shows measured values of geometricity metric for $k \in \{3,4,5\}$ decision nodes and $d = [2,10] \cap \mathbb{N}$ dimensions of instruction space. First number in each row indicates geometricity of random assignment of instructions and the latter one represents geometricity of the optimized space. Value of the metric clearly increases with number of dimensions, which could be connected with decreasing sizes of them and lowering average distance between instructions, causing the space to be 'more local'. Since locality optimization is, from its name, local operation, it is expected that locally measured geometricity can be higher than for more global scope. The geometricity noticeably decreased for $k = 3$ and $d \in \{9,10\}$. It is obviously caused by far more instructions in these two cases caused by insufficient number of unique instructions with this depth. Value of the metric slowly decreases with rising depth of instructions. Since for $k \geq 4$ the instruction space is sampled, the fall in value could be caused by absent instruction semantics and need future studies. Moreover the growth in geometricity is smaller for higher k, what can be caused by exponentially higher cardinality of the space.

The Figure 5.3 illustrates exemplary visualization of distribution of metainstruction semantics in the space, produced by embedding of 2-dimensional toroidal space containing metainstructions of depth 3. Since total number of instructions is 256 in that case, the space has 8×8 structure with no duplicates inside. The Figure shows pre-optimization state to the left as well as post-optimization one to the right. Semantics vector is wrapped around and drawn as 4×2 matrix, with 1s represented as red squares and 0s as white squares. Each instruction semantics is single tile and intensity of shading between pairs of tiles represents Hamming distance between them. General look at the optimized space clearly exposes brighter and darker areas representing instructions that produces many 1s (darker) and 0s (brighter) output values. Moreover there are visible boundaries between different areas. On the contrary the non-optimized space resembles white noise. The closer look at the optimized space proves general observations by similar appearance of neighboring tiles. On the other hand tiles in the left picture are arranged totally randomly.

In conclusion, the experimental results produced by proposed locality improvement approach for metainstruction space were described in this section. In particular results of local search steepest algorithm were examined for three sets of metainstructions. The sets were built upon OBDDs of depth $3,4$ and 5, and the space was build upon $2-10$ dimensional hyperrectangle. In two cases the space was made redundant, since there is no right space with desired dimensionality and as low as given cardinality. The cardinality of distinct metainstruction space rises rapidly with growing

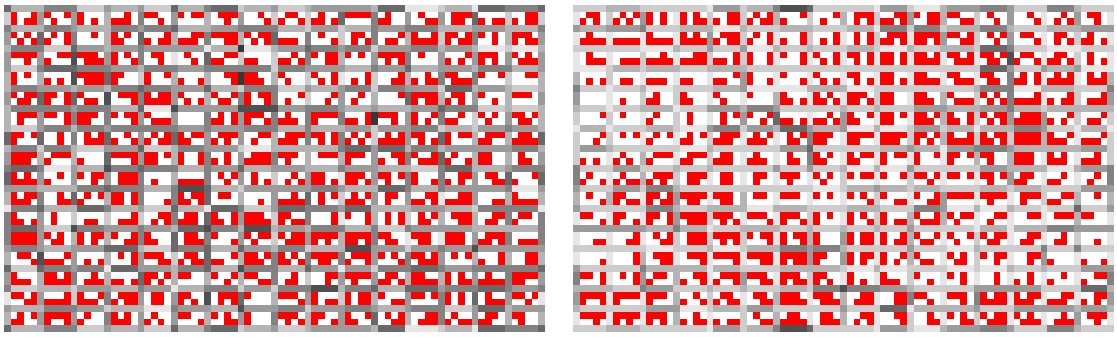

FIGURE 5.3: Visualization of distribution of metainstruction semantics in 2D space built upon set of OBDDs of depth 3. Left image illustrates state of space before optimization and right one depicts optimized space. Each tile represents semantics of one metainstruction, the red color indicates 1 and white 0, vector of semantics is wrapped in half of its length and put in 4×2 matrix. Shading between tiles is the darker the greater the semantic distance.

k	Average distance	Max distance	Average L_0	Average L^*	Average $\frac{L^*}{L_0}$
3	4.000 ± 0.171	8	0.269 ± 0.001	0.406 ± 0.016	1.510 ± 0.057
4	8.000 ± 0.021	16	0.269 ± 0.000	0.575 ± 0.047	2.132 ± 0.175
5	16.000 ± 0.030	31	0.270 ± 0.000	0.513 ± 0.047	1.905 ± 0.174

TABLE 5.7: Summary of locality optimization of metainstruction space. The table shows average and maximum values of semantic distances between metainstructions, as well as average values of locality measure for randomly assigned (L_0) and optimized (L^*) space. The values are divided into groups of equal k metainstruction depth. All averages are provided with 0.05 confidence intervals.

number of decision nodes, therefore there is a need for sampling of the space in order to keep it in computer memory. Table 5.7 shows average Hamming distance between metainstructions in all sampled sets and maximum distance. As you can see, in case of $k = 5$ the maximum distance is 31, while 'true' distance in entire space would be 32. It is clearly caused by lack of instruction samples caused by as low as 0.0008% selection probability. The probability cannot be increased here, due to memory limitations of the test system. The average distances in all cases are very similar to the distances in whole corresponding metainstruction space, whose are respectively $4, 8, 16$. Additionally the confidence intervals are very thin. These two facts clearly confirm that the spaces were uniformly sampled. The Table 5.7 also shows, averaged over the space dimensions, values of locality measure produced by random assignment of instructions and by optimization procedure. The random assignment behaves very similarly in all cases, causing confidence intervals to be very tight. Consequently the shape and size of instruction space have no impact on the random search algorithm. Nevertheless the dimensionality of the space has important meaning to the optimization algorithm, which behaved more effectively, in terms of both computational effort and value of objective function, on low dimensionality than on higher ones. The average locality improvement ratio varies from about one and a half to more than twice of the original value of non-optimized space. The ratio is also better for lower dimensionality of the space, than for higher. The number of decision nodes seems not to have an impact on the locality gain.

5.4 Search using optimized metainstruction space

In this section, a practical application of the optimized metainstruction space is discussed. It is shown, how much is the impact of the space optimization for performance of a program search process, particularly in case of standard Genetic Programming algorithm. The performance benchmark consists of 9-even- and 9-odd-parity problems. In n-even-parity problem the task is to evolve function that returns $true$ if the input vector contains even number of ones, on the other hand in the n-odd-parity problem $true$ is expected if input contains odd number of ones. The binary neural network of homogeneous architecture and configuration $3 \times 3 \times 1$ was used as program representation. The GP algorithm is equipped with toroidal crossover and mutation operators, discussed earlier in Section 4.5, in order to take advantage of coordinates in new (optimized) metainstruction space. Prior to the main run of the algorithm, some preliminary runs have been made with intention to find parameters for new genetic operators, such that they drive the search process to good results. All combinations of occurrence likelihood of crossover α_c and mutation α_m as well as probability of crossing-over β_c and mutation β_m of particular gene have been checked within range $[0,1]$ with step 0.125. Search performance has been analyzed on test set built upon all combinations of 9-bit input vector and proper program output bit. The set consists of 512 test cases. The same test set is employed in target search experiment. The parameters have been set to $\alpha_c = 0.9$, $\alpha_m = 0.1$, $\beta_c = 0.5$, $\beta_m = 0.35$. The number of correctly returned values was used as fitness function. The algorithm driven by that fitness function runs until either ideal solution is found or limit of 100 generations is reached, whatever comes first. The population size is 2000 in each trial.

Comparison of search performance of the GP algorithm working on optimized and non-optimized metainstruction space, as well as canonical problem implementation are illustrated in the Figure 5.4. The space is built upon metainstructions of depth $k = 3$ and it is organized as hyperrectangle. There are three different hyperrectangles involved: 2-dimensional 16×16, 4-dimensional $4 \times 4 \times 4 \times 4$ and 6-dimensional $2 \times 2 \times 2 \times 2 \times 4 \times 4$. The choice of that space is dictated by the maximum OBDD depth studied in Section 5.3, for which the space is complete (unsampled). The canonical implementation took advantage of Koza-I [44] setup, with population size altered to 2000 individuals and number of generations to 100 in order to be comparable with the new method. The figure consists of two charts, one for each variant of 9-odd/even-problem. The charts show mean and maximum fitness in population over the generations for search in optimized and non-optimized metainstruction spaces and for standard implementation. The results are averaged over 30 runs of algorithm with different random seeds.

The Genetic Programming algorithm clearly behaves better on optimized space than on non-optimized one. Since the even-parity and odd-parity problems are symmetric, there is no meaningful differences between their results. Consequently they are analyzed together. The optimization of the metainstruction space increased final program fitness from 9% up to 23% comparing with search in non-optimized space. The fitness gain is dependent on dimensionality of the metainstruction space and is higher for higher dimensionalities. Although the rising dimensionality generally impedes the search process and causes worse results. It obviously may be connected with low locality of high dimensional spaces. In almost all cases both search in optimized and non-optimized space behaves very similarly in first phase of the trial, then after about 20 generations both series

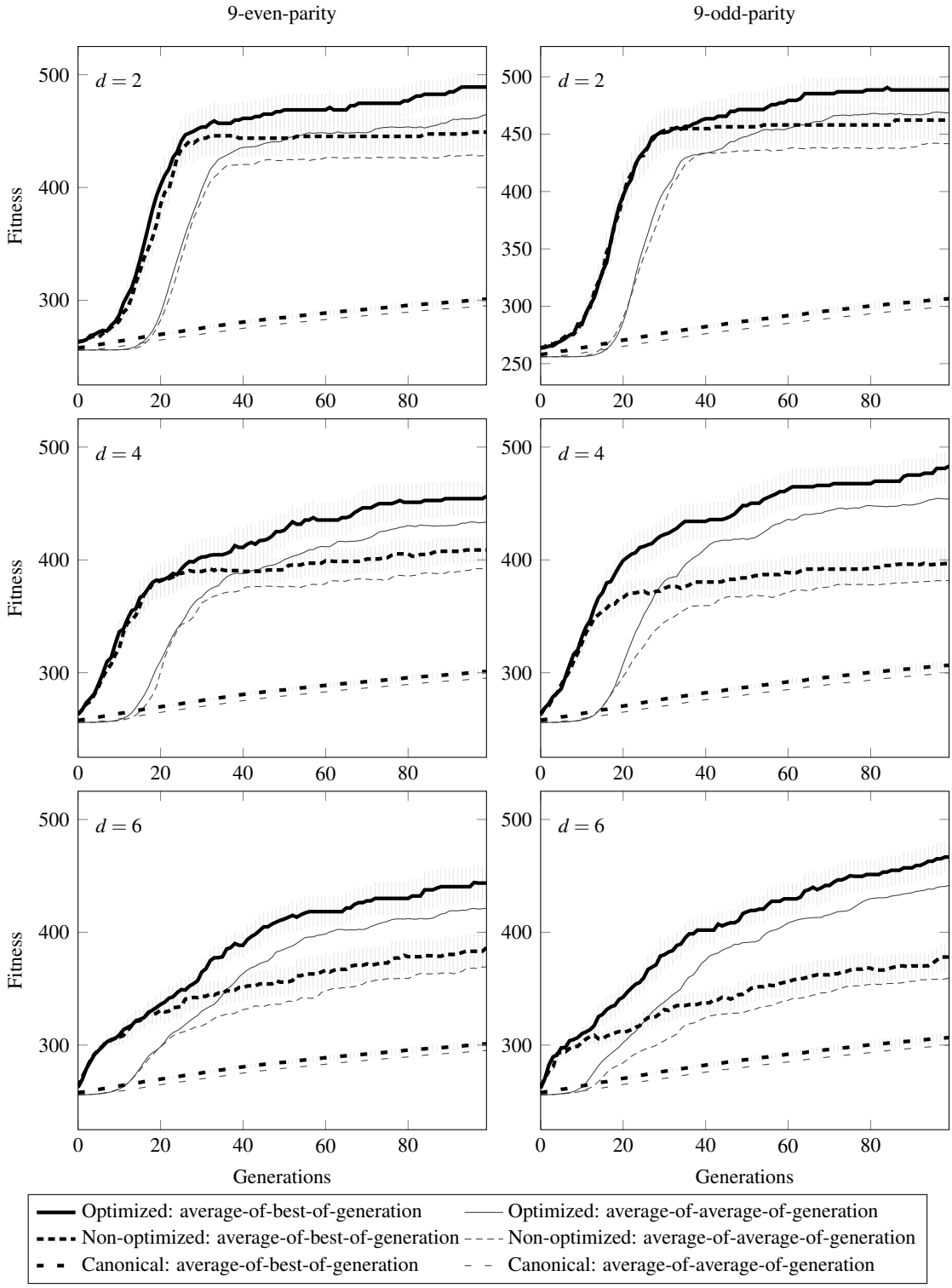

FIGURE 5.4: Comparison of search performance of Genetic Programming algorithm working on optimized and non-optimized d-dimensional metainstruction space and canonical problem implementation. Charts show fitness of both average-of-best-individual-in-population and average-of-average-individual-in-population in each generation. Presented values are averaged over 30 runs of the algorithm, additionally averages-of-best-individual-in-population are provided with 0.05 confidence intervals.

d	Optimized space	Non-optimized space	Canonical implementation
2	172521.5	398290.9	∞
4	306455.8	∞	
6	689041.2	∞	

TABLE 5.8: Comparison of computational effort of search in both optimized and non-optimized space for space dimensionality $d \in \{2,4,6\}$ and canonical implementation solving 9-parity problem.

separate from each other. The search through non-optimized space remains at acquired level, on the other hand search in the optimized space still improves the results but slower than before. The point of time when search in optimized space achieves its best results seems to depend on dimensionality of the space. The convergence is faster for lower dimensionalities. It can be explained by better value of locality measurement achieved for them as well as less complicated metainstruction neighborhood relation. The canonical implementation of parity problem behaves worse than others, but its results are more stable, what is represented by 0.05 confidence intervals in the charts. The difference between best results of canonical method and search in optimized space oscillates in range from 47% up to 62%.

The Table 5.8 presents computational effort of all three compared methods. The effort is computed as total number of fitness function evaluations divided by number of successful runs. The run is considered successful if it found optimal program or in other words it found program, whose fitness is $2^9 = 512$. Computational effort is the smallest for search in the optimized space, what was expected, since this method usually finds optimum very quickly. It is worth noting, that concept of metainstruction itself significantly decreased effort comparing to the canonical implementation, what can be seen in the table.

It is worth noting, that there were analyzed other neural network architectures like some modifications of network with all k-ary combinations of n inputs connected to $\binom{n}{k}$ neurons in the first layer and then aggregated by following layers to a single value. Unfortunately none of these architectures brought significant improvement in performance of the algorithm.

In conclusion, preliminary optimization of metainstruction space with regard to semantic locality increased performance of search process on parity problem by about one fifth. Additionally the optimization decreased program search effort and consequently computation times. Furthermore the concept of metainstruction and binary neural network raised values of program fitness almost one and a half times and noticeably decreased effort required to find the solution.

Finally, it is important to say, that all program search trials were run on single metainstruction space, optimized in previous step of experiment. Therefore the effort of space optimization can be distributed on all these search attempts and consequently it disappears in overall cost of the task. Moreover optimization of huge metainstruction spaces can be done once on high-end machine, such as a computer grid, and then used multiple times by standard computers performing program search tasks.

Chapter 6

Conclusions and Future Work

This master's thesis has described the problem of low fitness-distance correlation in Genetic Programming, and generally in program spaces. A locality metric to measure the strength of the correlation was proposed. Furthermore, the concept of learnable embeddings of program spaces was described, employing given locality measurement in the space optimization process. The optimization affects coordinates of programs in that space, so that semantically similar programs are placed together and different ones are separated from each other. This process smoothes program fitness landscape and in consequence, emphasizes its global convexity. The noticeable convexity of fitness landscape definitely increases performance of many non-random search algorithms, because the general way of improvement can be seen from multiple points of the landscape .

Because the space of constructible programs is in general unlimited, we proposed a notion of metainstruction – a program building block, limited in size and functionality. Therefore, the task of space optimization was shifted to the space of metainstructions, with hope that searches involving that space indirectly improve performance of the entire program search process. The experimental results showed that the locality of the [pre]space can be improved, in acceptable time, more than two times even by such a simple approach like local optimization algorithm. Moreover, the successive experiments have proven that search process, which makes use of the optimized space, can achieve up to one fifth times better results, with even less computational cost, than the same process working in the non-optimized space. The experiments have shown also that an alternative representation of instructions as ordered binary decision diagrams gives better results on logic problems than the canonical approach.

From the wide range of conceivable instruction space topologies, neighborhood relations and mapping definitions, in this study the hyperrectangle topology, the neighborhood limited to Hamming ball of radius one, and permutation mapping have been chosen. Note that the space of alternative representations is not limited to this arbitrary selection. For instance, the instruction space could be organized as a graph and the neighborhood in such a graph can be restricted by the maximal number of hops from one vertex to another. Moreover, metainstructions were analyzed on logic problems only, thus other applications need to be investigated. Furthermore, the concept of program space optimization is innovatory, therefore there are multiple questions that should be answered. For instance, what is optimal representation of the program space, what is its optimal dimensionality (assuming it is Cartesian), and what are optimal sizes of space dimensions? Is the optimal space topology related to the cardinality of semantically unique instructions? What is

the optimal definition of program neighborhood relation? Does the proposed definition of locality metric provide clear view on the smoothness of fitness landscape, or maybe there are better measures? Finally, is there a minimal size of the program space for which the proposed approach remains profitable, given the extra computational overhead involved? These and similar questions deserve being addressed in future research.

Bibliography

[1] Martha Chase A. D. Hershey. Independent functions of viral protein and nucleic acid in growth of bacteriophage. *The Journal of General Physiology*, 36:39–56, 1952.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *SJCC*, 1967.

[3] David Andre and John R. Koza. Parallel genetic programming on a network of transputers. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 111–120, Tahoe City, California, USA, 9 July 1995.

[4] Peter J. Angeline. An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover. 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 21–29, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[5] 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.

[6] A. O. L. Atkin and Daniel J. Bernstein. Prime sieves using binary quadratic forms. *Math. Comput*, 73(246):1023–1030, 2004.

[7] Douglas A. Augusto and Helio J. C. Barbosa. Symbolic regression via genetic programming. In *VI Brazilian Symposium on Neural Networks (SBRN'00)*, page 173, Rio de Janeiro, RJ, Brazil, January 22-25 2000. VI Simposio Brasileiro de Redes Neurais.

[8] Wolfgang Banzhaf. Genetic programming for pedestrians. March 1993. ICGA,.

[9] Forrest H Bennett III, John R. Koza, Jessen Yu, and William Mydlowec. Automatic synthesis, placement, and routing of an amplifier circuit by means of genetic programming. In Julian Miller, Adrian Thompson, Peter Thomson, and Terence C. Fogarty, editors, *Evolvable Systems: From Biology to Hardware Third International Conference, ICES 2000*, volume 1801 of *LNCS*, pages 1–10, Edinburgh, Scotland, UK, 17-19 April 2000. Springer-Verlag.

[10] Bollig and Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEETC: IEEE Transactions on Computers*, 45, 1996.

[11] George Boole. *An Investigation of the Laws of Thought*. Walton, London, 1854. Reprinted by Dover Books, New York, 1954.

[12] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.

[13] Flor Castillo, Arthur Kordon, Guido Smits, Ben Christenson, and Dee Dickerson. Pareto front genetic programming parameter selection based on design of experiments and industrial data. In Maarten Keijzer, Mike Cattolico, Dirk Arnold, Vladan Babovic, Christian Blum, Peter Bosman, Martin V. Butz, Carlos Coello Coello, Dipankar Dasgupta, Sevan G. Ficici, James Foster, Arturo Hernandez-Aguirre, Greg Hornby, Hod Lipson, Phil McMinn, Jason Moore, Guenther Raidl, Franz Rothlauf, Conor Ryan, and Dirk Thierens, editors, *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 2, pages 1613–1620, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[14] Kumar Chellapilla. Evolving computer programs without subtree crossover. *IEEE Transactions on Evolutionary Computation*, 1(3):209–216, September 1997.

[15] M. Conrad. Bootstrapping on the adaptive landscape. *BioSystems*, 11:167–182, 1979. Another discussion of the evolutionary self-facilitation of evolvability.

[16] 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.

[17] Charles Darwin. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. John Murray, London, 6th ed. edition, 1872.

[18] Patrik D'haeseleer. Context preserving crossover in genetic programming. In *International Conference on Evolutionary Computation*, pages 256–261, 1994.

[19] Edsger Wybe Dijkstra. On the cruelty of really teaching computing science. Circulated privately, December 1988.

[20] J. D. Watson F. H. C. Crick. The complementary structure of deoxyribonucleic acid. 1953.

[21] J. D. Watson F. H. C. Crick. A structure for deoxyribose nucleic acid. *Nature*, 171:737–738, 1953.

[22] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley, New York, USA, 1966.

[23] Guillaume Rochart Hadrien Cambazard Charles Prud'homme Arnaud Malapert Julien Menana Francois Laburthe, Narendra Jussien. CHOCO Solver. `http://www.emn.fr/z-info/choco-solver/`.

[24] Edgar Galvan-Lopez, James McDermott, Michael O'Neill, and Anthony Brabazon. Defining locality in genetic programming to predict performance. In *2010 IEEE World Congress on Computational Intelligence*, pages 1828–1835, Barcelona, Spain, 18-23 July 2010. IEEE Computational Intelligence Society, IEEE Press.

[25] Edgar Galvan-Lopez, James McDermott, Michael O'Neill, and Anthony Brabazon. Towards an understanding of locality in genetic programming. In Juergen Branke, Martin Pelikan, Enrique Alba, Dirk V. Arnold, Josh Bongard, Anthony Brabazon, Juergen Branke, Martin V. Butz, Jeff Clune, Myra Cohen, Kalyanmoy Deb, Andries P Engelbrecht, Natalio Krasnogor, Julian F. Miller, Michael O'Neill, Kumara Sastry, Dirk Thierens, Jano van Hemert, Leonardo Vanneschi, and Carsten Witt, editors, *GECCO '10: Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 901–908, Portland, Oregon, USA, 7-11 July 2010. ACM.

[26] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 69–93, San Mateo, 1991. Morgan Kaufmann.

[27] D.E. Goldberg and K. Grygiel. *Algorytmy genetyczne i ich zastosowania*. Wydawnictwa Naukowo-Techniczne, 1995.

[28] Kim Harries and Peter Smith. Exploring alternative operators and search strategies 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 147–155, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[29] Nguyen Xuan Hoai, R. I. (Bob) McKay, and Daryl Essam. Representation and structural difficulty in genetic programming. *IEEE Transactions on Evolutionary Computation*, 10(2):157–166, April 2006.

[30] J.H. Holland and J.H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, 1975.

[31] 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.

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

[33] Terry Jones. Crossover, macromutation, and population-based search. In Larry Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 73–80, San Francisco, CA, 1995. Morgan Kaufmann.

[34] M. A. Kaboudan. Genetic programming prediction of stock prices. 2000.

[35] Tatiana Kalganova. An extrinsic function-level evolvable hardware approach. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 60–75, Edinburgh, 15-16 April 2000. Springer-Verlag.

[36] M. Karnaugh. The map method for synthesis of combinational logic circuits. *AIEE Transactions, Part I Communication and Electronics*, 72:593–599, November 1953.

[37] Ahmad Kattan and Riccardo Poli. Evolutionary lossless compression with GP-ZIP. In Jun Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, pages 2468–2472, Hong Kong, 1-6 June 2008. IEEE Computational Intelligence Society, IEEE Press.

[38] Maarten Keijzer. Scaled symbolic regression. *Genetic Programming and Evolvable Machines*, 5(3):259–269, September 2004.

[39] Kenneth E. Kinnear, Jr. Evolving a sort: Lessons in genetic programming. In *Proceedings of 1993 IEEE International Conference on Neural Networks (Joint FUZZ-IEEE'93 and ICNN'93 [IJCNN93])*, volume II, pages 881–888, San Francisco, California, March-April 1993. IEEE/INNS. SunSoft.

[40] Kenneth E. Kinnear, Jr. Fitness landscapes and difficulty in genetic programming. In *Proceedings of the 1994 IEEE World Conference on Computational Intelligence*, volume 1, pages 142–147, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[41] Miha Kovacic and Joze Balic. Evolutionary programming of a CNC cutting machine. *International journal for advanced manufacturing technology*, 22(1-2):118–124, September 2003.

[42] John R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In *IJCAI*, pages 768–774, 1989.

[43] John R. Koza. Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Technical Report STAN-CS-90-1314, Department of Computer Science, Stanford University, Stanford, California 94305, June 1990.

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

[45] John R. Koza, David Andre, Forrest H Bennett III, and Martin A. Keane. Use of automatically defined functions and architecture-altering operations in automated circuit synthesis using 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 132–149, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

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

[47] Krzysztof Krawiec. Learnable embeddings of program spaces. In Sara Silva, James A. Foster, Miguel Nicolau, Mario Giacobini, and Penousal Machado, editors, *Proceedings of the 14th European Conference on Genetic Programming, EuroGP 2011*, volume 6621 of *LNCS*, pages 167–178, Turin, Italy, 27-29 April 2011. Springer Verlag.

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

[49] Vladik Kreinovich and R. Kearfott. Beyond convex? global optimization is feasible only for convex objective functions: A theorem. *Journal of Global Optimization*, 33:617–624, 2005. 10.1007/s10898-004-2120-1.

[50] 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.

[51] W. B. Langdon. Size fair and homologous tree crossovers for tree genetic programming, 2000.

[52] William B. Langdon and Peter Nordin. Evolving hand-eye coordination for a humanoid robot with machine code genetic programming. *Lecture Notes in Computer Science*, 2038:313–??, 2001.

[53] William B. Langdon and Riccardo Poli. On turing complete T7 and MISC F–4 program fitnes landscapes. In Dirk V. Arnold, Thomas Jansen, Michael D. Vose, and Jonathan E. Rowe, editors, *Theory of Evolutionary Algorithms*, number 06061 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[54] 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.

[55] Ricky D. Ledwith and Julian F. Miller. Introducing flexibility in digital circuit evolution: Exploiting undefined values in binary truth tables. In Gianluca Tempesti, Andy M. Tyrrell, and Julian F. Miller, editors, *Proceedings of the 9th International Conference Evolvable Systems: From Biology to Hardware, ICES 2010*, volume 6274 of *Lecture Notes in Computer Science*, pages 25–36, York, September 6-8 2010. Springer.

[56] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, July 1959.

[57] Joanna Lis. Genetic algorithm with the dynamic probability of mutation in the classification problem. *Pattern Recognition Letters*, 16:1311–1320, 1995.

[58] Jason D. Lohn, Gregory Hornby, and Derek S. Linden. Human-competitive evolved antennas. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 22(3):235–247, 2008.

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

[60] 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.

[61] Steven Manos, Maryanne C. J. Large, and Leon Poladian. Evolutionary design of single-mode microstructured polymer optical fibres using an artificial embryogeny representation. In Peter A. N. Bosman, editor, *Late breaking paper at Genetic and Evolutionary Computation Conference (GECCO'2007)*, pages 2549–2556, London, United Kingdom, 7-11 July 2007. ACM Press.

[62] Nicholas Freitag McPhee and Justin Darwin Miller. Accurate replication in genetic programming. In Larry Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 303–309, San Francisco, CA, 1995. Morgan Kaufmann.

[63] Nicholas Freitag McPhee, Brian Ohs, and Tyler Hutchison. Semantic building blocks in genetic programming. Working Paper Series Volume 3 Number 2, University of Minnesota Morris, 600 East 4th Street, Morris, MN 56267, USA, 12 December 2007.

[64] Microsoft. Windows PowerShell. `http://www.microsoft.com/powershell`.

[65] Julian F. Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach, 1999.

[66] Julian F. Miller and Stephen L. Smith. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Trans. Evolutionary Computation*, 10(2):167–174, 2006.

[67] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[68] Peter Nordin and Wolfgang Banzhaf. Programmatic compression of images and sound. 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 345–350, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[69] 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.

[70] Alan Piszcz and Terence Soule. Genetic programming: Analysis of optimal mutation rates in a problem with varying difficulty. In Geoff C. J. Sutcliffe and Randy G. Goebel, editors, *Proceedings of the Nineteenth International Florida Artificial Intelligence Research Society Conference*, pages 451–456, Melbourne Beach, Florida, USA, May 11-13 2006. American Association for Artificial Intelligence.

[71] R. Poli, W. B. Langdon, and Stephen Dignum. On the limiting distribution of program sizes in tree-based genetic programming. Technical Report CSM-464, Department of Computer Science, University of Essex, December 2006.

[72] Riccardo Poli. Discovery of symbolic, neuro-symbolic and neural networks with parallel distributed genetic programming. In George D. Smith, Nigel C. Steele, and Rudolf F. Albrecht, editors, *Artificial Neural Nets and Genetic Algorithms: Proceedings of the International Conference, ICANNGA97*, University of East Anglia, Norwich, UK, 1997. Springer-Verlag. published in 1998.

[73] Riccardo Poli. Parallel distributed genetic programming. In David Corne, Marco Dorigo, and Fred Glover, editors, *New Ideas in Optimization*, Advanced Topics in Computer Science, chapter 27, pages 403–431. McGraw-Hill, Maidenhead, Berkshire, England, 1999.

[74] Riccardo Poli and W. B. Langdon. A new schema theory for genetic programming with one-point crossover and point mutation. 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 278–285, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[75] Riccardo Poli and William B. Langdon. On the search properties of different crossover operators in genetic programming. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 293–301, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[76] 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.

[77] 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).

[78] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Reihe Problemata. Frommann-Holzboog, 1973.

[79] B. Roy. The outranking approach and the foundations of ELECTRE methods. In C.A Bana e Costa, editor, *Readings in Multiple Criteria Decision Aid*, pages 155–183. Springer-Verlag, Berlin, 1990.

[80] Rafal Salustowicz and Jürgen Schmidhuber. Probabilistic incremental program evolution, 1997.

[81] Rafal Salustowicz, Marco Wiering, and Jürgen Schmidhuber. Learning team strategies: Soccer case studies. *Machine Learning*, 33(2-3):263–282, 1998.

[82] Gabriel Balan Sean Paus Zbigniew Skolicki Elena Popovici Keith Sullivan Joseph Harrison Jeff Bassett Robert Hubley Alexander Chircop Jack Compton William Haddon Stephen Donnelly Beenish Jamil Joseph Zelibor Eric Kangas Faisal Abidi Houston Mooers Sean Luke, Liviu Panait and James O'Beirne. Evolutionary Computation in Java. `http://cs.gmu.edu/~eclab/projects/ecj/`.

[83] Detlef Sieling. The nonapproximability of OBDD minimization. *Inf. Comput*, 172(2):103–138, 2002.

[84] Y. Siskos, E. Grigoroudis, and N.F. Matsatsinis. Uta methods. In J. Figueira, S. Greco, and M. Ehrgott, editors, *Multiple Criteria Decision Analysis: State of the Art Surveys*, pages 297–344. Springer Verlag, Boston, Dordrecht, London, 2005.

[85] 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.

[86] Rainer Storn. On the usage of differential evolution for function optimization. In *NAFIPS'96*, pages 519–523. IEEE, 1996.

[87] Rainer Storn and Kenneth Price. Differential evolution: A simple and efficient heuristic for global optimization over continuous spaces. *J. of Global Optimization*, 11:341–359, December 1997.

[88] Walter Alden Tackett. Genetic generation of "dendritic" trees for image classification. In *Proceedings of WCNN93*, pages IV 646–649. IEEE Press, July 1993.

[89] Astro Teller. Turing completeness in the language of genetic programming with indexed memory. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 136–141, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[90] F. Towhidi, A.H. Lashkari, and R.S. Hosseini. Binary decision diagram (bdd). In *Future Computer and Communication, 2009. ICFCC 2009. International Conference on*, pages 496 –499, april 2009.

[91] Nguyen Quang Uy, Nguyen Xuan Hoai, Michael O'Neill, and Bob McKay. The role of syntactic and semantic locality of crossover 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 533–542, Krakow, Poland, 11-15 September 2010. Springer.

[92] 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*. Online first.

[93] Mark Voorneveld. Characterization of pareto dominance. *Oper. Res. Lett*, 31(1):7–11, 2003.

[94] Klaus Weinert and Marc Stautner. A new view on symbolic regression. In James A. Foster, Evelyne Lutton, Julian Miller, Conor Ryan, and Andrea G. B. Tettamanzi, editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 113–122, Kinsale, Ireland, 3-5 April 2002. Springer-Verlag.

[95] S. Wright. The roles of mutation, inbreeding, crossbreeding, and selection in evolution. In *Proceedings of the Sixth Congress on Genetics*, volume 1, page 365, 1932.

[96] Taro Yabuki and Hitoshi Iba. Turing-complete data structure for genetic programming. In *IEEE International Conference on Systems, Man and Cybernetics*, volume 4, pages 3577–3582, Washington, D.C., USA, 5-8 October 2003. IEEE Press.

Poznan University of Technology
Faculty of Computing Science
Institute of Computing Science

Typeset using LaTeX in Computer Modern.

BibTeX:

```
@mastersthesis{ key,
   author = "Tomasz Pawlak",
   title = "{Automated Design of Semantically Smooth Instruction Spaces
for Genetic Programming}",
   school = "Poznan University of Technology",
   address = "Pozna{\'n}, Poland",
   year = "2011",
}
```