Derivation Tree Based Genetic Programming

Summary of the Ph.D. Thesis

by

Róbert Ványi

supervisor:

apl. Prof. Dr.-Ing. Gabriella Kókai

Doctoral School of Computer Science

Faculty of Science and Informatics

University of Szeged

Szeged, 2012

1 Introduction

This booklet summarizes the results presented in the Ph.D. thesis entitled *Derivation Tree Based Genetic Programming*. The thesis describes a method with the same name.

The structure of this summary follows the outline of the thesis: after introducing the problem domain in this section, the relevant subjects from the field of evolutionary algorithms and formal grammars are summarized. Section 4 describes the DTGP method, whereas Section 5 provides a summary of some extensions of the base method, like introducing semantic constraints. Section 6 gives an overview of the results regarding several applications of DTGP, and Section 7 lists the major findings of the thesis.

1.1 Problem domain

In computer science a very common task is to solve optimization problems. This includes minimizing, maximizing or, in general, finding the *best* solution. One can distinguish between *deterministic* and *stochastic* methods. While deterministic methods are usually easier to analyze, their time complexity often makes them impractical. Stochastic algorithms, on the other hand, come with some uncertainty due to their randomized nature, and may yield a less optimal solution, but in many cases they provide better results within a reasonable time frame.

Evolutionary algorithms are metaheuristic optimization algorithms that use the evolution in nature and the *survival of the fittest* as the schema for creating, examining and selecting candidate solutions. [3] Since their introduction, various types of evolutionary algorithms, such as *evolution strategies*, *genetic algorithms* and *genetic programming* have been defined and used to solve diverse problems from engineering design optimization to automated construction of computer programs.

The *black box paradigm* means that the optimization algorithm has little to no information regarding the structure of the candidate solutions or the solution space itself. This also means that often it cannot be guaranteed that the new candidate is a valid solution of the problem, which has a negative impact on the quality of the search. The invalid solutions decrease the effective population size and introduce extra computational effort for detection, evaluation and sometimes correction.

One method for regulating the evolutionary search is to use *formal grammars* to guide the evolutionary process. If the set of the candidate solutions or their representations is a *formal language*, that is the set can be described by a grammar, one can restrict the search to the set of solutions by adhering to the rules of the grammar. Several *grammar guided genetic programming* (GGGP) approaches have been defined, many of them use *context-free grammars* for guidance, and representations of the *derivations* as genotypes. Operating on derivations instead of solutions guarantees the validity of the generated candidates. [6]

1.2 Proposed method

The method presented in the thesis uses *derivation trees* as representations and defines the evolutionary operators in a way that the results are always valid derivation trees. The improvement over existing GGGP methods is the extensive use of parameters. These parameters are stored at each node of the tree and can be used for various purposes, such as guaranteeing the balanced behavior of the random node selection, reducing the evaluation time from linear to logarithmic and introducing search bias or semantic constraints.

1.3 Publications

The results presented in the thesis are based on the five publications listed below in chronological order:

[11] R. Ványi and Sz. Zvada *Avoiding syntactically incorrect individuals via parameterized operators applied on derivation trees.* In R. Sharker, et al., editors, Proceedings of the 2003 Congress on Evolutionary Computation CEC2003, volume 4, pages 2791–2798, Canberra, 8-12 Dec 2003. IEEE Press.

[16] Sz. Zvada and R. Ványi *Improving grammar-based evolutionary algorithms via attributed derivation trees.* In M. Keijzer, et al., editors, Genetic Programming 7th European Conference, EuroGP 2004, Proceedings, volume 3003 of LNCS, pages 208–219, Coimbra, Portugal, 5-7 Apr 2004. Springer-Verlag.

[12] R. Ványi and Sz. Zvada *Syntactically correct genetic programming.* In R. Poli et al., editors, GECCO 2004 Workshop Proceedings, Seattle, Washington, USA, 26-30 Jun 2004.

[15] Sz. Zvada, G. Kókai, R. Ványi, and H.H. Frühauf *EvolFIR: Evolving redundancy-free FIR structures.* In Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007), pages 439–446. IEEE Computer Society, 5-8 Aug 2007.

[10] R. Ványi *Enforcing semantic constraints with derivation tree based genetic programming.* Abstract accepted to oral presentation at Veszprém Optimization Conference: Advanced Algorithms (VOCAL 2012), 11-14 Dec 2012.

In [11] the foundations of the DTGP method were introduced with the basic tree operators and random tree generation. In [16] the *balanced random node selector* was constructed and some possibilities for parameter usage were outlined. In [12] DTGP was compared with other GGGP methods, *pool crossover* was introduced and the time complexity of the operators was evaluated. In [15] a real-world application, developed in cooperation with the Fraunhofer Institute for Integrated Circuits, was introduced, and the first attempts to apply *semantic constraints* were made. The first formalization of *semantically constrained derivation* including *distribution sets*, *distribution functions* and *forced synthesized attributes* will be presented in [10].

1.4 Nomenclature

Throughout the thesis the following notions are used. Function f to be optimized is called *target function*. It is defined over a set S, called *solution space*. The search algorithm might work on a superset of the solution space, called *hypothesis space*. The elements of hypothesis space H are called *hypotheses*, *candidates* or *candidate solutions*. The elements of the solution space are called *solutions*.

Grammars in the thesis mean *context-free grammars*, and derivation trees are *complete derivation trees* over a given context-free grammar. *Parameters* are values stored at nodes of the derivation trees, usually defined in a *bottom-up* fashion, meaning they are calculated based on the values in the subtrees.

2 Evolutionary algorithms

Evolutionary algorithms (or EAs for short) use a special way to find an optimal solution. They are based upon the evolution that can be observed in nature. When using evolutionary algorithms, the search method is *parallel*, this means several hypotheses are examined at the same time. These hypotheses are called *individuals*, and they make up a *population*. The initial population is created randomly. Afterwards, each hypothesis is evaluated using a *fitness function*, showing how good a hypothesis is considered. Then, proportionally to the *fitness value*, individuals are selected as *parents* using a *selection* operator. Finally, by applying evolutionary operators, such as *mutation* or *recombination* to the set of parents, a new population of *offsprings* is generated, and the process is started over with the new population.

During this process, in the population (also called *generation* at the end of a given step) better and better individuals appear. The process is stopped when the so called *halting criterion* is satisfied. Usual halting criteria are the number of steps, small or no change in the best fitness, or reaching close approximation of the optimum.

There are many types of evolutionary algorithms. [3] The two major types are *evolution strategies* (ES) and *genetic algorithms* (GA). A third type is called *genetic programming* (GP), which was derived from genetic algorithms.

2.1 Optimizing complex structures

In the thesis an example was shown for optimizing a complex structure with evolutionary algorithms. The goal was to find a Boolean expression, possibly short, that matches a pre-defined Boolean function. Using genetic algorithms the expressions were represented by strings over the alphabet containing variables, negated variables, operator symbols for conjunction and disjunction, and parentheses.

The test results showed that GA produces many syntactically incorrect individuals, which prevents finding a solution. There are some methods to tweak the algorithm and find correct solutions, but according to the test results the number of invalid individuals still remains high and the success rate remains low.

Genetic programming was designed to evolve abstract syntax trees, especially ones that represent S-expressions of the LISP programming language. Using GP, syntactically correct individuals can be created, but only with some limitations: canonical GP requires the *closure property* to be fulfilled. [5, 9] On one hand this requires *type consistency*, which means that all arguments and return values must have the same type. It is required because the evolutionary operators can replace subtrees arbitrarily. On the other hand the closure property includes *evaluation safety*, which means that every possible expression that can be represented by a subtree can be evaluated. It is necessary so that a fitness value can be assigned to every candidate. Therefore, canonical GP does not provide a generic solution for restricting the optimization process to syntactically correct individuals.

However, GP can be extended to use formal grammars to ensure the syntactical correctness during evolution. Such methods are called grammar guided genetic programming (GGGP) algorithms. [6] The method proposed in the thesis is also a GGGP method that uses context-free grammars, therefore it relies on some important properties of these grammars.

3 Grammars and formal languages

The most commonly used tools to describe how to build languages from the symbols of an alphabet are *formal grammars* as proposed by Chomsky. [2] In the thesis *context-free* grammars are of special interest. A *context-free grammar* G is a quadruple $G = (\mathcal{N}, \Sigma, \mathcal{P}, S)$, where $\mathcal{N}$ is an alphabet of *nonterminal symbols*, Σ is an alphabet of *terminal symbols*, with $\mathcal{N} \cap \Sigma = \emptyset$, $\mathcal{P} \subset \mathcal{N} \times (\mathcal{N} \cup \Sigma)^*$ is a finite set of *rewriting rules*, and $S \in \mathcal{N}$ is the *start symbol*.

Grammars generate languages using derivations. The start symbol is taken, and then it is replaced using an appropriate rule. There may be nonterminal symbols in the new word, so they are also replaced. It is repeated until a word containing only terminal symbols is reached. Since there are usually multiple rules that can be applied, more than one word can be generated using a single grammar. The concept of derivations is essential for most GGGP methods, therefore a formal definition is also given.

Definition 3.1 *(Derivation)*
Direct derivation over grammar $G = (\mathcal{N}, \Sigma, \mathcal{P}, S)$, denoted by $\Rightarrow_G$ is a binary relation over $(\mathcal{N} \cup \Sigma)^*$. For any $\gamma, \delta \in (\mathcal{N} \cup \Sigma)^*$ $\gamma \Rightarrow_G \delta$ if and only if $\exists \varphi, \psi \in (\mathcal{N} \cup \Sigma)^*$ and an $A \to \beta$ rule in $\mathcal{P}$, such that $\gamma = \varphi A \psi$ and $\delta = \varphi \beta \psi$. Derivation is the reflexive transitive closure of the direct derivation relation, that is $\Rightarrow_G^*$.

With the help of direct derivation, one can also define a *derivation sequence*, which is a finite sequence of words $\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_n$, such that $\alpha_i \Rightarrow_G \alpha_{i+1}$. The length of the derivation sequence is the number of direct derivation steps, that is n.

Using the definition of derivation, the language generated by grammar G, denoted by $L(G)$ can be defined as follows: $L(G) = \{u \in \Sigma^* \mid S \Rightarrow_G^* u\}$.

3.1 Derivation trees

Derivations over context-free grammars can easily be visualized. One can consider the letters of the derived word as nodes. When a nonterminal symbol is replaced with a string, that is a sequence of symbols, these symbols can be represented as children of the node representing the original symbol in a tree structure, which is called *derivation tree*. The actual derived word can be found at the *frontier* of the tree, which is the sequence of the leaves in a left-to-right ordering.

It is very important to note that a word can be derived from a symbol if and only if there exists a derivation tree with the symbol as root and the word as frontier. It is especially important for the start symbol and the terminal words as stated by the following specialized theorem.

Theorem 3.2
For any $u \in \Sigma^*$ $S \Rightarrow^* u$ if, and only if there exists $T \in \mathcal{T}(S)$, such that $fr(T) = u$, where $\mathcal{T}(S)$ denotes the set of derivation trees with root S.

Corollary 3.3
For any $u \in \Sigma^*$ $u \in L(G)$ if, and only if there exists $T \in \mathcal{T}(S)$, such that $fr(T) = u$.

This corollary means that if one operates on derivation trees and ensures that the results are also valid derivation trees, then the words found at the frontiers are always valid solutions as well.

3.1.1 Derivation tree sizes

In the thesis derivation tree is used as data type, therefore it is interesting to examine the relation between the size of the derived words, derivations and derivation trees. If the number of terminal symbols on the right-hand side of rule r is denoted by $|r|_\Sigma$, a correlation can be formulated as shown by the following theorem.

Theorem 3.4 *(Correlation of derivation related sizes)*
Given a context-free grammar $G = (\mathcal{N}, \Sigma, \mathcal{P}, S)$. For an arbitrary derivation sequence $\alpha_0, \alpha_1, \ldots, \alpha_k$ with $\alpha_k \in \Sigma^*$, let us denote the size of the assigned derivation tree by s and the length of the derived terminal word α_k by n. In this case $s = \mathcal{O}(k)$. Furthermore, if $|r|_\Sigma$ averaged over all applied rule r_i $(1 \leq i \leq k)$ is at least 1, then $k \leq n$.

3.2 Attribute grammars

Attribute grammars extend the concept of context-free grammars by defining attributes for the symbols and assigning values to these attributes during derivation. [1]

An *attribute grammar* AG is a quadruple $AG = (G, SD, AD, \mathcal{R})$, where G is a context-free grammar, SD is the *semantic domain*, defining the attribute types, functions and relations, AD contains the *attribute descriptions*, and $\mathcal{R} = \{\mathcal{R}(p) \mid p \in \mathcal{P}\}$ is a family of sets defining the *semantic rules* for each rewriting rule of grammar G.

In the thesis the following assumptions are used regarding attributes. An attribute a is called *inherited* if for each rule $X \to Y_1 Y_2 \ldots Y_n$ the calculation schema is the following:

$$Y_i.a = f_a(X.a, Y_1.a, \ldots, Y_{i-1}.a),$$

and *synthesized* if the calculation schema is the following:

$$X.a = g_a(Y_1.a, Y_2.a, \ldots, Y_n.a).$$

More formal definitions are not required for the purposes of the thesis, nevertheless they can be found in the literature. Informally, inherited attributes can also be called *top-down* attributes, whereas synthesized attributes can be called *bottom-up* attributes.

4 Derivation tree based genetic programming

The standard approach to handle invalid individuals is to rely on an evaluation function to filter them out, for example by using very low fitness values. Another approach is not to allow such individuals at all, that is to modify the representation or the evolutionary operators so that only valid solutions are generated. This can be done in three different ways.

Allow any hypothesis The simplest way is to define the problem domain so that any hypothesis is a solution. This requires no change in the evolutionary algorithm. An example for this approach is the canonical GP as proposed by Koza. [5]

Restrict operators The straightforward approach is to modify the operators. This, however, might not be a simple task. This is how *strongly typed genetic programming* works. [7]

Redefine representations The third possibility is to redefine the set of representations to be closed under the evolutionary operators, which might also require some small changes in the operators. This approach is followed by DTGP.

4.1 Grammar guided genetic programming

Grammar guided genetic programming (*GGGP*) [6] is an approach recently gaining popularity in the GP arena. GGGP methods assume that the set of valid individuals is a context-free language, and use context-free grammars to guide the evolutionary process. There are two approaches depending on the data type.

4.1.1 Tree based GGGP

One of the first tree based GGGP experiments were done by Gruau [4], who used derivation trees to check the validity of the individuals. However, after the verification the trees were deleted, and were re-created again for the offsprings. Whigham [13] represented the individuals in the population as derivation trees and applied the operators directly to them. The limitation of his approach is that the operators cannot be parameterized, they can be influenced only by global parameters.

4.1.2 Linear GGGP

Currently the most researched linear GGGP method is *grammatical evolution* (*GE*). [8] It uses bitvectors to represent the individuals, which makes it possible to use all the methodologies and knowledge from the field of genetic algorithms.

The hypotheses are assumed to be words over an alphabet Σ and the set of solutions is described by a grammar G. The valid representations, that is the words of language $L(G)$ can be described by left derivations, which in turn are represented as index vectors in binary format. GE uses these index vectors as individuals.

However, this approach has some disadvantages. First of all, as only the sequence of the rule indices is stored, the derivations have to be carried out and the derivation trees, or at least their frontier must be computed to get the actual solution. Furthermore, there is no one-to-one mapping between bitstrings and derivations. This might lead to incomplete or infinite derivations.

4.2 Basics of DTGP

Derivation tree based genetic programming (DTGP) is a tree based grammar guided GP method. The representations are derivation trees over a pre-defined context-free grammar G. The solutions, that is the words of the language $L(G)$, can be found at the frontiers of these trees. The evolutionary operators are defined so that they only produce valid derivation trees. As a consequence, only valid hypotheses can be generated by the evolutionary process. The basics of this method are very similar to the one proposed by Whigham [13], however, the data type is extended with parameters that are used to improve the algorithm.

4.2.1 Derivation tree data type

The derivation tree is basically a standard tree data type, usually using a pointer based implementation, storing the label at each node. However, DTGP not only stores the labels at each node, but other information as well. Thanks to these parameters this approach has some advantages over other tree-based GGGP methods, as it is described in the thesis.

The notations used throughout the thesis are summarized in Table 1. Note that sometimes node and tree are used interchangeably. For example the size of a tree can be denoted by $T.size$, but it is usually stored as a parameter of its root node N, thus denoted by $N.size$.

symbol	**meaning**	**remark**
$T, T_1, T_2, \ldots$	tree, subtree	$T_1, T_2, \ldots$ denote the subtrees of T
$N, N_1, N_2, \ldots$	node	$N_1, N_2, \ldots$ are the children of N
$N[T_1, \ldots, T_n]$	tree	root node N with subtrees $T_1, \ldots, T_n$
$N.label$	label	label of node N
$T.label$	label	label of the root node of tree T
$N.param$	parameter	parameter of node N
$T.param$	parameter	parameter of the root node of tree T

Table 1: Notations for components of the derivation tree data type

4.2.2 Parameterized derivation trees

During the evolutionary process several properties of the derivation trees might be used by the evolutionary operators. However, recalculating this data every time could be a huge effort. Therefore, DTGP stores these properties as parameters in the root nodes of the subtrees.

The operations change subtrees, and we want these changes to have an effect only on properties of a limited set of nodes, therefore we require that the properties of a tree depend only on the properties of its subtrees. That means these properties should be *bottom-up*, that is for each property p and each tree $T = N[T_1, \ldots, T_n]$ the calculation schema is $T.p = f_p(N, T_1, \ldots, T_n)$.

A disadvantage of storing parameters in each node is that they have to be maintained during the evolutionary process. However, as we defined the parameters to be *bottom-up*, it is easy to see that changing a subtree within a derivation tree does not have an effect on nodes other than the ones on the path from the subtree's root node to the root of the derivation tree. Therefore, the parameters can be updated by a simple algorithm, that only requires logarithmic time with respect to the size of the tree. It starts at the root of the recently changed subtree and updates the predecessors until it reaches the root.

4.3 Evolutionary operators

The evolutionary operators are based on the usual tree operators, however some modifications are needed, to ensure that only valid derivation trees are generated. For DTGP several mutation and crossover operators are defined. All these operators use two basic operations: *random tree generation* (*RTG*) and *random node selection* (*RNS*). The operators were designed not only to follow the restrictions imposed by the context-free grammar, but also in a way that does not significantly increase the complexity of the evolutionary algorithm.

4.3.1 Random tree generation

Random tree generator is used for creating the initial population and also by the mutation operator. It is an essential part of DTGP, because it is the only component that has any knowledge of the problem domain in form of a context-free grammar. The basic idea is to start with a given nonterminal symbol, take the applicable rules, randomly select a rule, apply it and then proceed to the nonterminal symbols at the frontier of the current tree. The algorithm continues as long as there are nonterminal symbols at the frontier.

To limit the size of the tree, a constant called min_p is assigned to each rule p to show the minimal size a subtree can have, if it is started by applying the given rule. Calculating min_p takes some time, but it has to be done only once, before starting the evolutionary process.

4.3.2 Random node selection

Each operator has to select one or more nodes of the derivation tree. However, due to the complex data structure, it is not a trivial task, especially since not every node of the tree is a possible candidate for selection.

Randomly selecting an element of a vector is straightforward, but it is not feasible to move all the nodes of a tree into a vector just to select a node. Therefore, the random node selector of DTGP is designed based on the selector method of the search tree, which needs logarithmic time. It means that a path from the root down towards the leaves is traversed by randomly selecting one of the children or the node itself at each step. If the node itself is selected, the search is finished.

However, if all the children have the same selection weight, RNS will be unbalanced, because nodes in smaller subtrees have higher probabilities. Therefore, at each node X the size of the subtree, denoted by $X.size$ is stored, and used as selection weight. That is, the probability of making a step $X \rightsquigarrow X'$ is defined as follows:

$$P(X \rightsquigarrow X') = \begin{cases} \frac{1}{X.size} & \text{if } X' = X, \\ \frac{X'.size}{X.size} & \text{if } X' \text{ is a child of } X. \end{cases}$$

It is can be seen that with this definition, each node in the tree has the same probability to be selected. Furthermore, since the selection path proceeds through the children, it always goes down, thus the maximal length of this path is equal to the height of the tree, which in turn is logarithmic with respect to the number of nodes. Since the value $X.size$ depends only on $X'.size$, where X' is a child of X, $X.size$ can be considered as bottom-up property, and as such, it can be stored as parameter in the nodes, making it available at no cost during random node selection. This method can also be generalized to allow different selection weights at each node.

4.3.3 Derivation tree mutation

Using random tree generation and random node selection as described in the previous subsections, a simple mutation can easily be defined. First a node is selected randomly, then its subtree is replaced by a randomly generated subtree.

The generated subtree may or may not be restricted by the actual attributes of the selected node. For instance, sometimes it makes sense to replace a subtree having a certain depth with another having exactly the same depth, however, in general no such restrictions are applied, other than the global limit on the new subtree size.

It is important to note that the mutation operator introduces only local changes and can be controlled by several parameters. The cost of a mutation is mainly composed of the costs of the random tree generation and the random node selection. Some additional computation has to be done to update the attributes after a new subtree is inserted, but as discussed in Section 4.2.2, it only requires logarithmic time.

4.3.4 Derivation tree crossover

A tree crossover for derivation trees can be defined easier than a mutation, since no subtrees have to be generated. Only two nodes have to be selected and the subtrees under these nodes have to be swapped. However, there is one issue. The selected nodes must have the same label to ensure valid derivation trees. If this is not the case, one can skip crossover or one can retry selection. It is also possible to select a node in the first parent, and then apply a special selection to select another one in the second parent having the same label, but this can be very inefficient.

There is another possibility: to redefine crossover to operate on the whole population instead of pairs of parents. This approach makes it possible to swap subtrees even if the simple random node selection is used, and the labels of the selected nodes are not specified in advance.

4.3.5 Pool crossover

The importance of crossover is that good candidates can contribute their building blocks into the next generation. This can not only be achieved by taking two parents and creating two offsprings. Crossover can also be defined to work on the whole population, just like global ES recombination. Based on this idea, a new operator for derivation trees called *pool crossover* is defined.

This operator works as follows. In the first step subtrees are selected, removed from the parents and inserted into a *contribution pool*, categorized by the label of the selected node. In the second step, these subtrees are inserted back into the parents, but in a random order.

4.3.6 Operator costs

The first look at the genetic operators might suggest that their costs are significantly higher than the ones for bitvectors, because of the tree operations such as removing or inserting subtrees. However, using a pointer based implementation, subtree removal and insertion can be solved in constant time, similarly to linked lists.

The average case operator costs were analyzed in the thesis, and they are summarized in Table 2. The analyzed operators were mutation (MUT)

with range r, and four types of crossover: standard (XO), with retry (XO-R), with weighted node selector (XO-W), and pool crossover (POOL). Columns RNS and OP show the time needed for random node selection and for applying the operator itself. The size of the solution, that is the length of the word is denoted by n and based on Theorem 3.4, the size of the derivation tree is assumed to be $\mathcal{O}(n)$.

	RNS	OP	update	success	size
MUT	$\mathcal{O}(\log n)$	$\mathcal{O}(\lvert\mathcal{P}\rvert r)$	$\mathcal{O}(\log n)$	100%	$\mathcal{O}(1)$
XO	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$1/\lvert\mathcal{N}\rvert^a$	$\mathcal{O}(1)$
XO-R	$\mathcal{O}(\lvert\mathcal{N}\rvert \log n)^a$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	100%	$\mathcal{O}(1)$
XO-W	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\lvert\mathcal{N}\rvert \log n)$	100%	$\mathcal{O}(\lvert\mathcal{N}\rvert)$
POOL	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\approx 100\%^b$	$\mathcal{O}(1)$

[a] depending on the frequency of the given nonterminal
[b] if the population is large compared to $\mathcal{N}$

Table 2: Summary of DTGP operator costs

This analysis shows a slight disadvantage of DTGP (and other tree-based GGGP methods) over string based GGGP methods, because the latter often require only constant time to apply the operators, and the success rate is 100%. However, it also demonstrates the advantages of DTGP over other tree based methods. Using parameters in the nodes, DTGP can decrease the time complexity from linear to logarithmic by avoiding the need for visiting each node during random node selection.

When discussing operator costs, one must also consider the *evaluation costs*. First the genotype, in this case a derivation tree, must be mapped to a phenotype by reading the frontier of the tree, and then the phenotype must be evaluated using a fitness function. It is important to note that the phenotype is always a syntactically correct hypothesis, thus no syntax check or correction has to be carried out. Reading the frontier normally requires full traversal of the tree, which is comparable to applying a derivation required for linear GGGP. Thus the cost of genotype-phenotype mapping for both approaches is $\mathcal{O}(n)$. However, for certain problems DTGP can use parameters to construct the phenotype or calculate the fitness function, thus reducing the evaluation costs to $\mathcal{O}(1)$. Such an improvement is not possible for algorithms that do not store the derivations, such

as linear GGGP methods.

4.4 DTGP example

To demonstrate how DTGP works and to examine some of its properties, an example was introduced and analyzed. The example was a five dimensional Boolean regression problem, the same one that was used as an example for genetic algorithms. The context-free grammar that generates the language of valid logical expressions for this example is very straightforward and has been presented in many textbooks.

4.4.1 Results

The standard setup was to use a population size of 1000, make 100 steps and apply standard mutation and pool crossover. For the initial population, the height did not exceed 15. For random node selection, each node had the same weight, except, of course, the leaves. During mutation, the randomly generated subtrees had a height limit of 10.

To calculate the fitness, the expression has been calculated for all the $2^5 = 32$ possible variable evaluations, and compared to the target function, represented as 32-bit numbers. Each match has been awarded with 1000 points, then the size of the expression has been subtracted, making the theoretical maximum of the fitness value 32000-1.

During a single test run, the algorithm was able to find an expression that exactly describes the function by generation 40, and then it was able to further optimize the size of the derivation tree from 532 down to 247. The result is comparable to the one reached by the GA algorithm when safe mutation is used. However, the GA process used a population of 10000 individuals, many of which were invalid. For the DTGP algorithm 1000 individuals were enough. The best solution found by the algorithm was a 93 symbol long expression.

Using the standard settings 100 independent runs were carried out. Of those, 38 runs were able to reach fitness values above 31000, that is all 32 bits were matched, meaning the run was successful. However, the median was below 31000, and in the worst case the fitness was slightly below 27000 meaning only a 27 bit match.

4.4.2 Parameter settings

A DTGP algorithm has many parameters that can be set. The most important parameters are the operator application rates, the number of steps and the population size. For a DTGP algorithm, one can also set node selection parameters and bounds for random tree generation.

DTGP has been tested with many settings. The results of these tests are summarized below.

Operator application rate Confirming the results presented in [14], the tests with various application rates show that mutation is the most important operator for DTGP, although a high application rate of crossover also improves the algorithm. Furthermore, it has also been verified that if the population is large enough, the chance that the pool crossover fails is negligible.

Number of steps Since the average fitness of the best solutions found by the algorithm gradually increases with each step, one can try to run the algorithm longer. However, the rate of improvement decreases, as it was also shown by the tests. It is mostly caused by the decreasing diversity in the population, thus it has been concluded in the thesis that making two independent runs can lead to better results than having a single run with twice as many steps. For example, with the setup used in the thesis, making 100 steps twice yields better results than making 200 steps.

Population size The proper population size is important for evolutionary algorithms, because a certain number of individuals is needed to maintain diversity. Furthermore, a large population covers a larger portion of the search space. However, increasing the population size improves results only up to a certain limit. Therefore, just like in the case of the number of steps, sometimes it is better to have more independent runs instead of increasing the population size. In the tests presented in the thesis, the best results were achieved with a population size of 500.

The search bias of the algorithm has also been examined in the thesis. The method shows a slight tendency towards generating larger trees, which is a common phenomenon in genetic programming, known as *bloat*. [9]

5 Improving DTGP

The tree structure used by DTGP is not expected to be asymptotically larger than the structures used by linear GGGP, but it is still a bigger and more complex data type. Fortunately, this data type also provides opportunities for improving the algorithm. The improvements presented in the thesis can be divided into three categories: harnessing the potentials of the parameters, implementing semantic constraints and improving randomization during tree generation.

5.1 Parameter utilization

Parameters can be used to store various subtree information at the nodes. In the thesis this information has been used for the following purposes:

Run-time frontier recovery The frontier of a tree can be stored as parameter in its root node. This is a bottom-up parameter, thus it can easily be handled by DTGP, and it makes the frontier of the tree available in constant time. Note, however, that this parameter needs a total of $\mathcal{O}(n \log n)$ space for a derivation tree, instead of $\mathcal{O}(n)$.

Run-time hypothesis evaluation To evaluate an individual, constructing the frontier itself is actually irrelevant, as long as the evaluation of the represented hypothesis can be done. If the hypothesis can be represented in a compact way, then it can be used as a parameter, so it can be available during fitness calculation in constant time.

Run-time fitness calculation In certain cases it is also possible to partially or even completely calculate and store the portion of the fitness value associated with the subtree. For example the cost associated with the hypothesis represented by the subtree.

Operator biasing For the random node selection, any selection weight defined as bottom-up property can be stored as parameter and used for biasing the random node selection. Three examples were shown in the thesis: selection with a threshold based on height, a weight based on breadth-height ratio, and an exponentially decreasing weight based on height.

5.2 Semantic constraints

The bottom-up parameters used in DTGP can be considered as synthesized attributes of an underlying attribute grammar, thus they can be referred to as *synthesized attributes*, or just *attributes*. The set of all possible values of attribute a is denoted by V_a. Unless noted otherwise, attributes are interpreted in a context of a single rule, because even if they represent the same information at each node, their definition might be different for each rule. Attributes are calculated based on the attribute values of the child nodes, that is $T.a = f_a(T_1.a, \ldots, T_n.a)$.

Having semantic constraints means that pre-defined semantic information has to be passed down to the subtrees, and might need to be updated along the way. This process can be described with the help of *distribution sets* and *distribution functions* that define how the semantic information is distributed among the subtrees.

Definition 5.1 *(Distribution set)*
Given a synthesized attribute a defined by function

$$T.a = f_a(T_1.a, \ldots, T_n.a)$$

The distribution set for attribute value $v_0 \in V_a$ and nonnegative integer $n \in \mathbb{N}$ is a set of vectors $D_a(v_0, n)$ defined as follows

$$D_a(v_0, n) = \{(v_1, \ldots, v_n) \in (V_a)^n \mid f_a(v_1, \ldots, v_n) = v_0\}$$

In words the distribution set is the set of all value vectors that synthesize the predefined value.

Definition 5.2 *(Distribution function)*
Given a synthesized attribute a. Let us define $\mathcal{D} \subseteq V_a \times \mathbb{Z}^+$ as the largest set, such that for each $(v_0, n) \in \mathcal{D}$ distribution set $D_a(v_0, n)$ is not empty. A distribution function is a randomized function $\hat{f}_a$ defined over $Dom(\hat{f}_a) = \mathcal{D}$ such that

$$\hat{f}_a(v_0, n) = \vec{v}, \text{ where } \vec{v} \in D_a(v_0, n)$$

That is $\hat{f}_a(v_0, n)$ is a randomly selected element of $D_a(v_0, n)$.

With the help of distribution sets and distribution functions, it is possible to define attributes that put semantic restrictions on the derivation trees. These are called *forced synthesized attributes*, and unlike synthesized attributes, these are not calculated based on subtree attributes. Instead, their values are distributed from parents to children nodes using the distribution functions.

Note that there is a similarity between inherited and forced synthesized attributes. In both cases the semantic information is propagated in a *top-down* fashion, however, the definition itself is given as *bottom-up* for a forced synthesized attribute.

Definition 5.3 *(Semantically constrained derivation)*
For a given grammar $G = (\mathcal{N}, \Sigma, \mathcal{P}, S)$, derivation sequence $\alpha_0, \alpha_1, \ldots, \alpha_n$ with $\alpha_0 = S$ is semantically constrained by forced synthesized attribute a, if for each step i with $\alpha_{i-1} = \varphi A \psi$, $\alpha_i = \varphi \beta \psi$ and rule $p : A \to \beta$ the following holds:

$$D_{a,p}(A.a) \neq \emptyset,$$

and if there are $k > 0$ nonterminals $B_1, B_2, \ldots B_k$ in β, then

$$(B_1.a, B_2.a, \ldots, B_k.a) \doteq \hat{f}_{a,p}(A.a),$$

where $\doteq$ means equals to one of the possible values.

During a semantically constrained derivation in each step, when rule $A \to \beta$ is applied, the value v_0 of the forced synthesized attribute a at the current node labeled with A is taken. First $D_{a,p}(v_0)$ is checked to see if the value can be distributed. If $D_{a,p}(v_0)$ is empty, then the given rule cannot be applied. If no applicable rules are found, the derivation is terminated. If an applicable rule is found, then the distribution function $\hat{f}_{a,p}$ is used to get the values for the new nonterminal symbols in β.

5.2.1 Random tree generation

During random tree generation, subtrees must be generated in a way that their attributes synthesize the proper value for the parent tree. Therefore, when their roots are created by applying the appropriate rule, the distribution function is used to distribute the semantic constraints among the subtree roots.

One issue with semantically constrained derivation is that it can lead to a dead-end, that is to a partial derivation tree having some nonterminal nodes with empty distribution sets. This is a problem-specific issue, and has to be considered when the distribution functions are defined.

5.2.2 Crossover

During standard crossover two trees are taken and nodes selected in both, such that they have the same label. However, if forced synthesized attributes are used, the values of these attributes must match as well, otherwise a conflict may occur.

To solve this issue, the crossover operator has to be modified, so that not only the labels but also the forced synthesized attributes match when two subtrees are swapped. Standard crossover can have difficulties even with matching labels, but pool crossover can be updated to handle forced synthesized attributes. To achieve this, pools are labeled not only with nonterminals, but also with the values of the forced synthesized attributes.

There is a chance that even pool crossover does not work. It happens when the size of a pool is exactly one. The probability of this to happen increases with the number of possible pools, and decreases with the size of the population. Therefore, pool crossover cannot be applied when there are too many nonterminal–attribute value combinations, unless the population size is increased.

5.2.3 Limitations of semantic constraints

When semantic constraints are defined, one has to consider the failed derivations as well. If the constraint is very restrictive, it becomes very difficult to generate correct solutions. This is the case if the target function is used as semantic constraint with the Boolean regression problem. To avoid failed derivations, the maximum height of the derivation trees has to be increased, but that exponentially increases the size of the trees. Thus, introducing such semantic constraint is useful only up to a certain limit. Furthermore, it is also easy to see that pool crossover becomes less and less effective as the constraint size is increased, since the number of pools is also increasing exponentially.

5.3 Randomization

Derivation tree based genetic programming, just like any other stochastic algorithm depends greatly on the quality of randomization, since it can introduce a bias in the search, and determines which parts of the search space have higher probabilities to be discovered. In the thesis the following aspects have been examined:

Selection Regarding the selection operator, DTGP is not different from any other EA. There are many different selection types that are used with evolutionary algorithms, and any of these can be used with DTGP as well.

Random node selection operates based on the node weight, stored as parameter. The standard definition of weight is 0 for the leaves and 1 for the other nodes. In the thesis two further examples were tested with the Boolean regression problem. Of the three tested node weights, the standard one provided the best results.

Random rule selection To optimize random rule selection, weights can be assigned to the rules based on several properties. In the thesis three possibilities were tested: the *minimum value of the rule* stored in constant min_p as mentioned earlier, the *number of nonterminals* on the right-hand side and the *number of follow-up rules*. The results showed that using the minimum value provides the best outcome.

Subtree limit distribution After a rule is selected and applied, that is new nodes are inserted into the derivation tree, the limits for the subtrees have to be calculated and passed on to the random tree generator for each nonterminal node. There are various strategies how to do this. The following were mentioned in the thesis: the standard method using *random* distribution, *weighted* by min_p and *sequential*. Nevertheless, the tests showed no significant improvement for the Boolean regression problem or the integer generation, therefore it is enough to use the standard limit distribution strategy.

6 DTGP applications

In the thesis the Boolean regression problem was used as an example to introduce DTGP. However, some additional examples were discussed briefly as well. The first example was the *6-Multiplexer* problem, which is a standard example for GP. The second example was the *traveling salesman problem*, which is often used to test optimization methods. The third example was a practical application for optimizing *finite input response filters*.

6.1 Multiplexer

The multiplexer is a logical gate with n address bits and 2^n data bits that returns the data bit selected by the address bits. In the thesis three possibilities were presented for solving the 6-Multiplexer problem (2 address bits and 4 data bits) using derivation tree based genetic programming.

The results showed that DTGP is capable of solving the 6-Multiplexer problem with 88% success rate, and when semantic constraints were used, the results were even better, showing a clear advantage of DTGP compared to other tree based GGGP methods.

6.2 Traveling salesman problem

The results of the various TSP tests showed that it is possible to construct a DTGP algorithm for optimizing a TSP route, although the found solution is not always optimal. Since the solutions for a TSP are paths or circles in graphs, they have an internal structure very different from words of context-free languages. Therefore, applying a syntactically constrained optimization algorithm is not the best choice, and one can only find very few examples in the literature of canonical or grammar guided genetic programming being applied to solve the TSP.

6.3 Finite input response filters

DTGP has been used in a joint project with the Fraunhofer Institute for Integrated Circuits to optimize *finite input response filters*. [15] In the thesis a summary of this real-world application has also been given.

7 Conclusions

In the thesis a new *grammar guided genetic programming* method, called *derivation tree based genetic programming* (DTGP) was defined and evaluated. It uses derivation trees over a pre-defined context-free grammar to represent individuals, and applies genetic programming on these trees. Thus, it can be categorized as *tree-based GGGP*, and like other grammar guided methods, it is able to guarantee that the produced individuals are always *syntactically correct* with respect to the given grammar.

Compared with linear GGGP approaches, the data type used by DTGP is larger, although usually not asymptotically, but nevertheless more complex. However, in the thesis it was presented how the evolutionary operators can be defined correctly and efficiently, such that the results are not only correct derivation trees, but the time complexity remains logarithmic most of the time.

How to use parameters to improve the algorithm was also shown. One important use is for random node selection. Furthermore, as presented in the thesis, bottom-up parameters can also be used to store information related to fitness calculation. In some cases the phenotype, or even the fitness value can be calculated as a parameter, making the evaluation a constant time operator with additional logarithmic time work for parameter updates.

The values of bottom-up parameters can also be defined in advance, so that they represent *semantic constraints*. By using *distribution sets* and *distribution functions*, as defined in the thesis, these values can be passed to the subtrees in a top-down fashion during random tree generation. This process is called *semantically constrained derivation*. Introducing semantic constraints is a significant improvement over standard GGGP approaches, because previously these kinds of constraints were only incorporated in the fitness function. That is, individuals not fulfilling the semantic criteria were created, but later excluded based on the fitness values.

In the thesis DTGP was analyzed in detail using a Boolean regression problem. Furthermore, the method was tested with the 6-Multiplexer problem, and it was also presented how DTGP can be applied to the traveling salesman problem. A real-world practical application optimizing *FIR (finite input response) filters* using an extended version of DTGP was also outlined.

The findings of the thesis can be summarized as follows:

Thesis I. DTGP, as defined in the thesis, is a specialized evolutionary algorithm that can be used to optimize various problems using a black-box principle, while still guaranteeing the syntactical correctness of the generated candidates.

- **a**. With the properly defined random tree generator, DTGP generates valid derivation trees while maintaining the required size limitations.
- **b**. Before applying an operator, DTGP can select a random node in the derivation tree in logarithmic time, while ensuring that the selection probability is the same for each node. Furthermore, the set of selectable nodes can be limited and, if required, a non-uniform selection weight can also be applied.
- **c**. The poorly performing standard crossover can be replaced by pool crossover, which has the same time complexity, but usually runs with practically a 100% success rate.

Thesis II. By making use of the extensive data structure of the derivation trees, and applying properly defined parameters, the behavior of DTGP can be adjusted and the algorithm can be improved.

- **a**. By storing the appropriate information in the nodes, in certain cases the fitness evaluation can be done in constant time. This needs additional work to update the parameters after the operators are applied, but that does not increase their overall time complexity.
- **b**. Using parameters, the random node selection and therefore the evolutionary operators can be biased, and the fitness evaluation can also be influenced.
- **c**. With the help of forced synthesized attributes, the DTGP algorithm can enforce semantic constraints.

Thesis III. DTGP can be applied to various problems, especially when the solutions have a structure that can be directly represented by a context-free grammar.

References

[1] H. Alblas. Introduction to attribute grammars. In *Proceedings of the International Summer School on Attribute Grammars, Applications and Systems (SAGA'91)*, volume 545 of *LNCS*, pages 1–16. Springer Verlag, 1991.

[2] N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2:137–167, 1959.

[3] A. E. Eiben and J. Smith. *Introduction to Evolutionary Computing*. Springer-Verlag, 2003.

[4] F. Gruau. On using syntactic constraints with genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 19, pages 377–394. MIT Press, Cambridge, MA, USA, 1996.

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

[6] R. I. McKay, N. X. Hoai, P. A. Whigham, Y. Shan, and M. O'Neill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3/4):365–396, Sept. 2010. Tenth Anniversary Issue: Progress in Genetic Programming and Evolvable Machines.

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

[8] M. O'Neill and C. Ryan. *Grammatical Evolution - Evolving programs in an arbitrary language.*, volume 4 of *Genetic Programming*. Kluwer Academic Publishers, 2003.

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

[10] R. Ványi. Enforcing semantic constraints with derivation tree based genetic programming. Abstract accepted to oral presentation at Veszprém Optimization Conference: Advanced Algorithms (VOCAL 2012), 11-14 dec 2012.

[11] R. Ványi and S. Zvada. Avoiding syntactically incorrect individuals via parameterized operators applied on derivation trees. In R. Sarker, R. Reynolds, H. Abbass, K. C. Tan, B. McKay, D. Essam, and T. Gedeon, editors, *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, volume 4, pages 2791–2798, Canberra, 8-12 dec 2003. IEEE Press.

[12] R. Ványi and S. Zvada. Syntactically correct genetic programming. In R. Poli et al., editors, *GECCO 2004 Workshop Proceedings*, Seattle, Washington, USA, 26-30 jun 2004.

[13] P. A. Whigham. Grammatically-based genetic programming. In J. P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, Tahoe City, California, USA, 9July 1995.

[14] S. Zvada. *Attribute Grammar Based Genetic Programming.* Cuvillier Verlag, 2010.

[15] S. Zvada, G. Kókai, R. Ványi, and H. H. Frühauf. EvolFIR: Evolving redundancy-free fir structures. In *Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*, pages 439–446. IEEE Computer Society, 5-8 aug 2007.

[16] S. Zvada and R. Ványi. Improving grammar-based evolutionary algorithms via attributed derivation trees. In M. Keijzer, U.-M. O'Reilly, S. M. Lucas, E. Costa, and T. Soule, editors, *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, pages 208–219, Coimbra, Portugal, 5-7 apr 2004. Springer-Verlag.