Derivation Tree Based Genetic Programming

Ph.D. Thesis

by

Róbert Ványi

Supervisor:

apl. Prof. Dr.-Ing. Gabriella Kókai

A Thesis Submitted in Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Doctoral School of Computer Science
Faculty of Science and Informatics
University of Szeged

Szeged, 2012

Acknowledgments

Writing a thesis is a challenging journey and it begins long before the first word is written. Fortunately, I was lucky enough to have had great help during this endeavor.

First of all, I would like to express my gratitude to my supervisor, apl. Prof. Dr.-Ing. Gabriella Kókai, who introduced me to the field of evolutionary computation. Without her support and guidance this thesis would have never been completed. Her early departure was a great loss not only for her family and friends, but also to the scientific community.

I would like to thank Dr.-Ing. Szilvia Zvada, with whom I worked together on the foundations of the DTGP method. I also received a lot of assistance from many professors and colleagues at the Institute of Informatics of the University of Szeged and the Department of Programming Languages at the University of Erlangen.

Last, but not least I would like to thank my wife, Nelli, for her constant support and motivation, and my family and friends for not asking about my progress on my thesis too many times.

Contents

1 Introduction

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.

Stochastic methods are often combined with a *heuristic* approach. These kind of algorithms, also called *metaheuristics* or *black box optimization* methods, iteratively generate new candidate solutions and examine them to decide how good they are. [26] An important task, when designing such algorithms, is to find a strategy for discovering the search space.

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. [14] The various types of evolutionary algorithms, such as *evolution strategies*, *genetic algorithms* and *genetic programming* have been 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. The evolutionary algorithms usually use some computer representations, also called *genotypes* to describe the solution candidates, which are called *phenotypes*. This duality makes the setup more complex, meaning that the outcome of changing a candidate to create a new one is often hard to predict, and sometimes it cannot be guaranteed that the new candidate is a valid solution of the problem.

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 were 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. [28]

In this thesis *derivation tree based genetic programming* (DTGP) is presented. It 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 to introduce search bias or semantic constraints.

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

[45] 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.

[52] 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.

[46] 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.

[51] 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.

[44] 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 [45] the foundations of a derivation tree based GP method were introduced including basic tree operators and random tree generation. In [52] the *balanced random node selector* was constructed and some possibilities for parameter usage were outlined. In [46] DTGP was compared with other GGGP methods, *pool crossover* was introduced and the time complexity of the operators was checked. In [51] a real-world DTGP application, developed 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 [44].

We will proceed as follows. In Chapter 2 and Chapter 3 introductions to evolutionary algorithms and formal grammars are given. In Chapter 4 the motivation behind grammar guided genetic programming is discussed and the details of derivation tree based genetic programming are defined. The chapter is closed with a detailed example. In Chapter 5 some improvements of DTGP are discussed, including a method to apply semantic constraints. In Chapter 6 three further examples are shown and finally the results of the thesis are concluded in Chapter 7 and Chapter 8 in English and in Hungarian respectively.

2 Evolutionary algorithms

Evolutionary algorithms are optimization methods simulating the biological evolution by searching for optima using the concept of the *survival of the fittest*. The idea was already hinted by Alan Turing in a technical report in 1948 [41, 42], and the field of *evolutionary computation* saw a significant development after the 1970's.

The goal of optimization methods is to find a solution for a problem so that the solution is optimal with respect to some metric. The *search method* defines how to find those solutions. A straightforward method is to check each solution, however, this method is often impractical, even if the search space is finite. For some problems a fast deterministic algorithm can be given, but for other problems sometimes the whole search space have to be examined. In this case one possibility is to find some way to exclude some parts of the search space. One may also have a conjecture, where to search for the optimal solution. These "guesses" are called *heuristics*. When an exact solution is not required, approximation algorithms might also be used.

Throughout this thesis we will use the following notions. 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*.

Evolutionary algorithms (EA) use a special way to find an optimal solution. They are based on the evolution that can be observed in nature. [10] 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 the 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.

Usually the hypotheses are complex or abstract constructions that are represented by simple data structures. The evolutionary operators are applied to the representations, which in turn are interpreted as hypotheses and evaluated using the fitness function. In the context of evolutionary algorithms the elements of the hypothesis space, that is the hypotheses are called *phenotypes*, whereas the representations, that is the elements of the population are called *genotypes*.

There are many types of evolutionary algorithms. They can differ in data type, selection method, and, of course, in the operators. There may be other, slighter differences as well. The two major types are *evolution strategies* (ES) [35, 37] and *genetic algorithms* (GA). [19, 17] A third type is the so called *genetic programming* (GP) [22] which was derived from genetic algorithms. However, these algorithms are often combined, thus in practice the applied methods sometimes cannot be classified as pure ESs or GAs. The layout of a basic evolutionary algorithm is represented by Algorithm 2.1. The implementation of a more general version is described in [43].

```
EVOLUTIONARY-ALGORITHM(size)
 1  generation_number = 0
 2  population := ∅
 3  while population.size < size
 4  do population+=RANDOM-SOLUTION
 5  while HALTING-CRITERIA ≠ true
 6  do for  each individual in population
 7      do EVALUATE-FITNESS(individual)
 8      parents := SELECT-INDIVIDUALS(population)
 9      offsprings := RECOMBINE(parents)
10      MUTATE(offsprings)
11      while offsprings.size < size
12      do offsprings+=RANDOM-SOLUTION
13      population := offsprings
14      generation_number++
```

Algorithm 2.1: General evolutionary algorithm

In the following section the basic types of evolutionary algorithms are discussed briefly. First the data structure is defined for each method together with the operators. Then the selection method is given, and finally the specialties are mentioned, if there is any.

2.1 Evolution strategies

Evolution strategies were developed by Ingo Rechenberg [35] and Hans-Paul Schwefel [37]. The first applications were engineering problems, such as optimizing the shape of a pipe, or the structure of a nozzle. These experiments were not even carried out using computers, but real models were built, tested and changed.

2.1.1 Data structure

Since the solution is usually a set of physical parameters, the individual is represented by a vector of real or integer values. For an advanced ES an extended vector is used containing additional, instance-specific parameters that influence the evolutionary operators. In general the format of an *ES individual* is the following:

$$\mathbf{g} = (\mathbf{p}, \mathbf{s}),$$

where $\mathbf{p} = (p_1, p_2, \ldots, p_l)$ is vector of the *object parameters* and $\mathbf{s} = (s_1, s_2, \ldots, s_l)$ is the vector of the *strategy parameters*. The strategy parameters are optional. If given, they control how the object parameters are changed.

2.1.2 Mutation

In the first ES experiments only the mutation operator was used. This operation simply changes the elements of the vector by adding random values. These random values must have zero mean value, and a normal distribution with a pre-defined deviation. When defined, strategy parameters are used as deviation values, and they can also be changed by mutation. In general, *ES mutation* is carried out as follows.

Definition 2.1 *(ES mutation)*
Given an ES individual $\mathbf{g} = (\mathbf{p}, \mathbf{s})$. The mutation of $\mathbf{g}$ yields a new individual $\mathbf{g}'$ as follows.

$$\mathbf{g}' = (\mathbf{p}', \mathbf{s}'), \text{ where}$$

$$\begin{aligned} \mathbf{p}' &= (p_1 + \xi(s_1), p_2 + \xi(s_2), \ldots, p_l + \xi(s_l)) \\ \mathbf{s}' &= (m(s_1), m(s_2), \ldots, m(s_l)) \end{aligned}$$

and $\xi(s_i)$ is a random value having Gaussian distribution with deviation s_i. Function $m : \mathbb{R} \to \mathbb{R}$ is called *mutative stepsize adaption* (MSA). When the strategy parameters are not present in the individual, a pre-defined constant s with $s_1 = s_2 = \ldots = s_l = s$ is used as deviation.

An example for ES mutation is shown in Figure 2.1. There are 3 object parameters, and therefore 3 strategy parameters. The MSA function is $m(x) = x$, so the strategy parameters do not change. As it can be seen in this example, if the strategy parameters are higher, the object parameters can change in a wider region. If the strategy parameters are the same for each individual, then the strategy parameters do not have to be stored with the individuals.

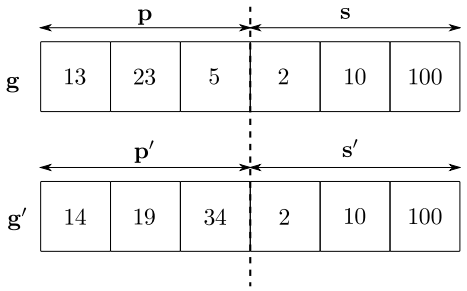


Figure 2.1: ES mutation

2.1.3 Recombination

For recombination two or more *parents* are required. The operator works by applying a recombination function $rec_k : \mathbb{R}^k \to \mathbb{R}$ at each position in the parent vectors. Each time this function is applied, k parents are selected. Depending on whether these parents are selected in advance or separately for each position, one can distinguish global and local recombinations.

Definition 2.2 *(Global ES recombination)*
Given a population of n individuals $\mathbf{g}^1, \mathbf{g}^2, \ldots, \mathbf{g}^n$ with length l and a parameter $2 \leq k \leq n$, $k \in \mathbb{N}$. Vector $\mathbf{g}'$ obtained by global ES recombination with parameter k is the following:

$$\mathbf{g}' = (\mathbf{p}', \mathbf{s}') = (p'_1, p'_2, \ldots, p'_l, s'_1, s'_2, \ldots, s'_l), \text{ where}$$

$$\begin{aligned} p'_j &= rec_k(p_j^{i_{j,1}}, p_j^{i_{j,2}}, \ldots, p_j^{i_{j,k}}) \\ s'_j &= rec_k(s_j^{i_{j,1}}, s_j^{i_{j,2}}, \ldots, s_j^{i_{j,k}}) \end{aligned}$$

and $\{i_{j,1}, i_{j,2}, \ldots i_{j,k}\} \subseteq \{1, 2, \ldots, n\}$ is randomly selected for each j.

Definition 2.3 *(Local ES recombination)*
Given a population of n individuals $\mathbf{g}^1, \mathbf{g}^2, \ldots, \mathbf{g}^n$ and a parameter $2 \leq k \leq n$, $k \in \mathbb{N}$. Vector $\mathbf{g}'$ obtained by local ES recombination with parameter k is the following:

$$\mathbf{g}' = (\mathbf{p}', \mathbf{s}') = (p'_1, p'_2, \ldots, p'_l, s'_1, s'_2, \ldots, s'_l), \text{ where}$$
$$p'_j = rec_k(p_j^{i_1}, p_j^{i_2}, \ldots, p_j^{i_k})$$
$$s'_j = rec_k(s_j^{i_1}, s_j^{i_2}, \ldots, s_j^{i_k})$$

and $\{i_1, i_2, \ldots i_k\} \subseteq \{1, 2, \ldots, n\}$ is randomly selected before applying the operator.

Note that rec_k has not been defined yet. This is usually done in one of the following two ways, which are called *discrete* and *intermediate* recombinations respectively.

Definition 2.4 *(Discrete ES recombination)*
An ES recombination is discrete if rec_k is the following:

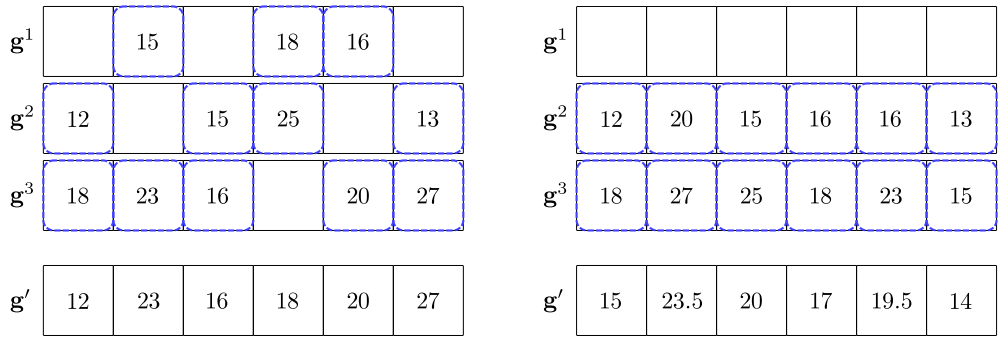
$$rec_k(p_1, p_2, \ldots, p_k) = p_i,$$

where $i \in \{1, 2, \ldots, k\}$ is a random number with uniform distribution.

Definition 2.5 *(Intermediate ES recombination)*
An ES recombination is intermediate if rec_k is the following:

$$rec_k(p_1, p_2, \ldots, p_k) = \frac{1}{k} \sum_{i=1}^{k} p_i$$

It is also possible to define other operators, for example using median value instead of average. An example for discrete global ES recombination can be seen in Figure 2.2a, and another example for intermediate local ES recombination in Figure 2.2b. In both cases $n = 3$, $k = 2$ and strategy parameters are not used.



(a) discrete global (b) intermediate local

Figure 2.2: ES recombinations

2.1.4 Selection

In ES the selection operator always selects the best μ individuals as parents, and puts these parents into the *mating pool*. Then using the mutation and recombination operators λ new individuals are created from the mating pool. The new population can be created either as the union of the mating pool and the set of the new individuals, or as the set of the new individuals only. When parents survive, it is called $(\mu+\lambda)$-ES, when not, we speak about (μ, λ)-ES. Special cases are $(1, 1)$-ES, which is random search, and (1+1)-ES, which is a simple hill-climbing algorithm.

2.1.5 ES algorithm

Algorithm 2.2 describes the general ES algorithm. It can be seen that it is very similar to the general evolutionary algorithm. However, while the general algorithm does not specify how to apply the operators exactly, the ES algorithm does. As shown in lines 12–18, first recombination is applied to the mating pool, and then mutation is applied to the result.

```
EVOLUTION-STRATEGY(μ, λ, strategy)
 1  if strategy = plus
 2     then size := μ + λ
 3     else size := λ
 4  generation_number := 0
 5  population := ∅
 6  while population.size < size
 7  do population+=RANDOM-VECTOR
 8  while HALTING-CRITERIA ≠ true
 9  do for each individual in population
10     do EVALUATE-FITNESS(individual)
11     mating_pool := SELECT-INDIVIDUALS(population, μ)
12     if strategy = plus
13        then offsprings := mating_pool
14        else offsprings := ∅
15     for λ steps
16     do child := RECOMBINE(mating_pool)
17        MUTATE(child)
18        INSERT(offsprings, child)
19     population := offsprings
20     generation_number++
```

Algorithm 2.2: General ES algorithm

There is an extended version of ES, where evolution is applied not only to the individuals but also to the population itself. It is called *meta-ES*. In meta-ES multiple populations are used, and the evolution process is run independently in each population. After a certain number of steps the average fitness is calculated for each population, and based on this fitness value populations are selected and recombined to create new populations. It has been observed, that meta-ES provides better solutions for several problems within the same time. [21]

2.1.6 ES example

To show how ES works, a sample problem from the field of evolutionary design is presented together with its solution by ES. The problem is the following: Given a fluid storage in the shape of a cylinder. There are 11 rings in equal distances, and they are covered with some kind of rubber. The size of the rings can be changed, so the storage will be a union of truncated cones. The volume of the storage has to be at least a given V. A sample deformation for four rings can be seen in Figure 2.3. Our task is to minimize the surface area of the storage.

In this case an individual is a vector of the ring radii: $g = (r_0, r_1, \ldots, r_{10})$. Strategy parameters are not used, and the fitness value is simply the surface of the storage. However, it must also be ensured that the volume is at least V, thus if the actual volume is smaller, a fitness penalty is given: a large constant value, for example 50000 is added to the fitness value, thus, the individual will not be good enough to survive.

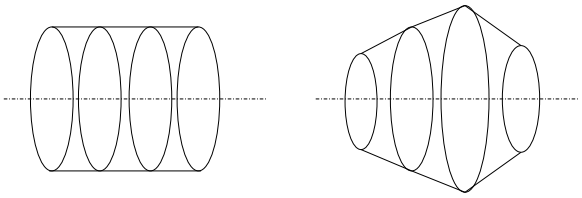
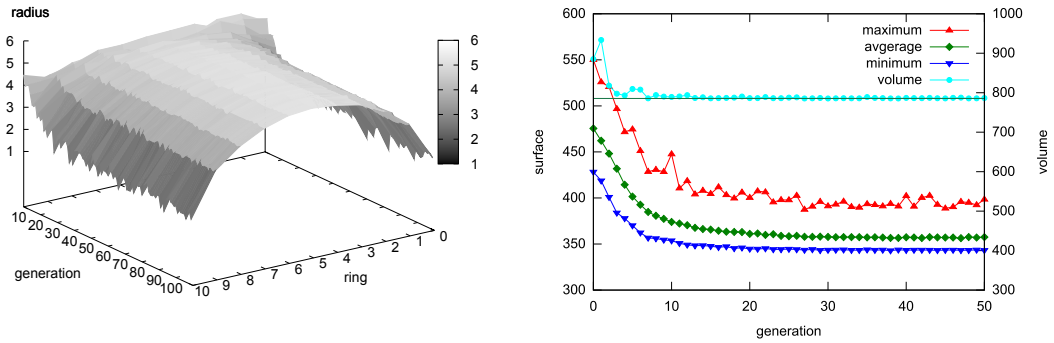


Figure 2.3: Sample deformation for four rings

To optimize the ring sizes (λ, μ)-ES was used with $\lambda = 1000$ and $\mu = 100$. It was run for 100 iteration steps. The initial radius of the rings was set to 5, and the minimal volume was the volume of the initial cylinder, that is $10 \cdot 5^2\pi$. Standard mutation and intermediate local recombination were used, the latter with $k = 2$, that is using two parents.

In Figure 2.4a the evolved ring radii are shown in the consecutive generations. In Figure 2.4b the minimal, maximal and average surface is displayed, along with the volume of the best individual for the first 50 generations, excluding individuals representing too low volumes. One can see that the best and average fitness gradually converges to the anticipated optimum. It can also be noticed that the volume sometimes grows, but due to the fitness penalty it never falls below the minimal value which is approximately 785.4. The found configuration has the following radii rounded to two decimals: $1.74, 3.52, 4.35, 4.88, 5.10, 5.22, 5.12, 4.79, 4.23, 3.40, 1.44$; the surface area is 342.52, the volume is 786.30. The shape of the storage tried to approximate a sphere, which is known to have the smallest surface area among objects with the same volume.



(a) ring sizes

(b) surface and volume

Figure 2.4: ES test results

2.2 Genetic algorithms

Genetic algorithms [19, 17] are the most widely used evolutionary optimization methods. With respect to the data structure GAs model the biological evolution closer than ESs. The most common data structure is *bitvector*, but sometimes words over other finite alphabets are used as well. Due to the similarity to the DNA, the individuals are called *chromosomes*. Because of this data type, usually the solutions have to be encoded, for example for real values a binary representation has to be used. [29] The other consequence of this representation is that genetic algorithms always make a strict distinction between the solution, that is the *phenotype* and the representation, that is the *genotype*. In each step the genotype is mapped onto a phenotype before fitness evaluation.

2.2.1 Data structure

A *GA individual* is a word in a given length over a finite alphabet:

$$c = b_1 b_2 \dots b_l \in \Sigma^l, \text{ where}$$

Σ is a finite alphabet, often $\Sigma = \{0, 1\}$. Word c is called *chromosome*, a position i is called *locus*, a value b_i at this position is called *gene*. The values that b_i can take, that is the elements of Σ are called *alleles*. The standard version of GA requires length l to be constant, but *variable length GA* allows it to be different for each individual. [6, 16]

2.2.2 Mutation

GA mutation simply changes a bit b_i to its complement $\bar{b}_i$, or replaces a character with another character. The probability of mutation is usually low.

Definition 2.6 *(GA mutation for bitvectors)*
Given a chromosome $c = b_1 b_2 \dots b_l \in \{0, 1\}^l$ and mutation probability P_{mut}. The result of mutation is a chromosome $c' = b'_1 b'_2 \dots b'_l$, where

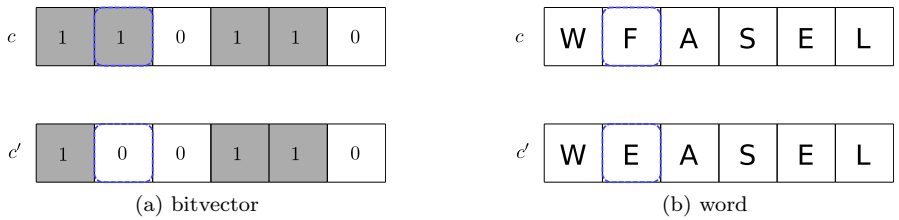
$$b'_i = \begin{cases} \bar{b}_i & \text{with probability } P_{mut} \\ b_i & \text{with probability } 1 - P_{mut} \end{cases}$$

Definition 2.7 *(GA mutation for words)*
Given a chromosome $c = b_1 b_2 \dots b_l \in \Sigma^l$, where Σ is a finite ordered alphabet. Given mutation probability P_{mut} and mutation rate R_{mut}, the result of mutation is a chromosome $c' = b'_1 b'_2 \dots b'_l$, where

$$b'_i = \begin{cases} b_j + r_i \mod |\Sigma| & \text{with probability } P_{mut} \\ b_i & \text{with probability } 1 - P_{mut} \end{cases}$$

and $r_i \in [-R_{mut}, -1] \cup [1, R_{mut}]$ is a random value. Sometimes $R_{mut} = |\Sigma|$, so b'_i can be an arbitrary element of Σ with probability P_{mut}. In this case Σ need not be ordered.

Examples for bitvector and word mutations are shown in Figure 2.5.



(a) bitvector (b) word

Figure 2.5: GA mutation

2.2.3 Crossover

For genetic algorithms the recombination operation is also more like the one in nature, therefore the name *crossover* is used. Several types can be defined such as single-point crossover or multi-point crossover. In contrast to ES recombination, GA crossover always uses two parents.

Definition 2.8 *(Single-point crossover)*
Given two chromosomes $c = x_1x_2 \dots x_ix_{i+1} \dots x_l \in \Sigma^l$, $d = y_1y_2 \dots y_jy_{j+1} \dots y_l \in \Sigma^l$ and given two loci $1 \leq i \leq l-1$ and $1 \leq j \leq l-1$. The results of crossover are the following two chromosomes:

$$c' = x_1x_2 \dots x_iy_{j+1} \dots y_l, \text{ and } d' = y_1y_2 \dots y_jx_{i+1} \dots x_l.$$

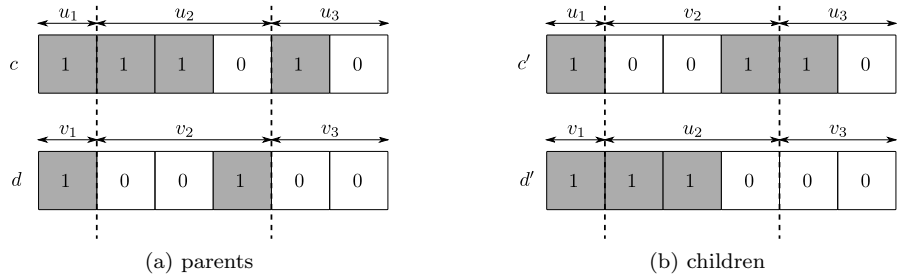
Note that if the individuals have to be of the same length, then $i = j$ is required. Multi-point crossover can be considered as a generalization of the single-point crossover.

Definition 2.9 *(Multi-point crossover)*
Given two chromosomes $c = x_1x_2 \dots x_l \in \Sigma^l$ and $d = y_1y_2 \dots y_l \in \Sigma^l$. Let us take random dissociations of these words to $2 \leq k \leq l$ (possibly empty) subwords: $c = u_1u_2 \dots u_k$, $u_i \in \Sigma^*$ and $d = v_1v_2 \dots v_k$, $v_j \in \Sigma^*$. It's common, but not required to have $|u_i| = |v_i|$. The results of crossover are the following two chromosomes:

$$c' = u'_1u'_2 \dots u'_k, \text{ and } d' = v'_1v'_2 \dots v'_k, \text{ where}$$

$$u'_i = \begin{cases} u_i & \text{if } i \text{ is even} \\ v_i & \text{otherwise} \end{cases} \qquad v'_i = \begin{cases} v_i & \text{if } i \text{ is even} \\ u_i & \text{otherwise} \end{cases}$$

It is easy to see that single-point crossover is a multi-point crossover with $k = 2$. An example for a multi-point crossover with $k = 3$ is shown in Figure 2.6.



(a) parents (b) children

Figure 2.6: GA crossover

2.2.4 Selection

There are numerous selection methods defined for GA. Usually GA does not select the best individual, but the selection is *fitness proportional*. Individuals are selected with a probability proportional to their fitness values and copied into the *parent population*. For higher fitness value, the expected number of copies is larger. Beside fitness proportional selection many other types of selection methods are defined. For example *best selection* can be used, which is the same as the ES selection. Further selection methods are described in the literature. [30]

2.2.5 GA process

An outline of a genetic algorithm is shown in Algorithm 2.3. It can be seen that generating random individuals is very simple, as shown in line 4, but before evaluating an individual, the genotype has to be decoded to a phenotype in line 8. Creating the new population consist of selecting two parents and first applying crossover, which yields two children and then applying mutation to these children. This is repeated until the new population is filled.

```
GENETIC-ALGORITHM(size)
 1  generation_number := 0
 2  population := ∅
 3  while population.size < size
 4  do population+=RANDOM-STRING
 5  while HALTING-CRITERIA ≠ true
 6  do offsprings := ∅
 7      for each individual in population
 8      do phenotype := DECODE-GENOTYPE(individual)
 9          EVALUATE-FITNESS(phenotype)
10      parents := SELECT-INDIVIDUALS(population)
11      while offsprings.size < size
12      do parent_1 := RANDOM-SELECT(parents)
13          parent_2 := RANDOM-SELECT(parents)
14          (child_1, child_2) := CROSSOVER(parent_1, parent_2)
15          MUTATE(child_1)
16          MUTATE(child_2)
17          offsprings+={child_1, child_2}
18      population := offsprings
19      generation_number++
```

Algorithm 2.3: Example genetic algorithm

It must be mentioned that there are several types of GAs, which modify one or more property of the standard GA presented here. For example *real-coded genetic algorithms* (*RCGA*) use real values instead of bitstrings. [15] One can also modify GA to use it for multi-objective optimization. [11] The idea of meta-ES can be transferred to GA as well, such that multiple populations can be used with various interactions. This idea is used by *parallel GA* [33], *distributed GA* [39], *injection island GA* [25], and many others. [2]

2.2.6 GA example

Let us consider Boolean expressions as words. For this example the alphabet contains variables, negated variables, operator symbols for conjunction and disjunction and parentheses.

$$\Sigma_{br} = \{x_0, x_1, x_2, x_3, x_4, \bar{x}_0\, \bar{x}_1, \bar{x}_2, \bar{x}_3, \bar{x}_4, \wedge, \vee, (,)\}$$

Our goal is to find an expression representing the Boolean function that is given by Table 2.1.

$\mathbf{x}$	$f(\mathbf{x})$
00000	1
00001	0
00010	0
00011	0
00100	0
00101	0
00110	1
00111	1

$\mathbf{x}$	$f(\mathbf{x})$
01000	1
01001	0
01010	1
01011	1
01100	1
01101	1
01110	0
01111	0

$\mathbf{x}$	$f(\mathbf{x})$
10000	0
10001	0
10010	1
10011	0
10100	0
10101	0
10110	0
10111	0

$\mathbf{x}$	$f(\mathbf{x})$
11000	0
11001	1
11010	1
11011	1
11100	0
11101	1
11110	0
11111	0

Table 2.1: The searched Boolean function with $\mathbf{x} = (x_4, x_3, x_2, x_1, x_0)$

Since this function has 5 Boolean variables, its domain has $2^5 = 32$ elements, thus the target function can also be considered as a 32-bit number. Using $f(00000)$ as the most significant bit, and $f(11111)$ as the least significant bit, this value is `0x83BC2074` in our case.

To ensure that the initial population contains valid expressions only, several rules must be followed by the random individual generator, as listed below.

- Random words are generated from left to right, adding a new symbol in each step
- Variables and closing parentheses must be followed by operators or closing parentheses
- Operators and opening parentheses must be followed by variables or opening parentheses
- The number of closing parentheses must not exceed the number of opening parentheses in the prefix generated so far
- If the number of opening parentheses is greater than the number of closing parentheses at the end, then extra closing parentheses are added

For crossover and mutation the standard operators are used. It means that invalid individuals may occur, that is the operators might generate strings that cannot be interpreted as Boolean expressions. However, one can also allow a permissive evaluation: if a closing parenthesis is found with no matching opening parenthesis, the individual is considered as valid and the value is the expression represented by the prefix leaving some unused characters at the end of the word. This also occurs in biological evolution, where the unused genes are called *introns* and represent dormant genetic information. In the following charts the proportion of the invalid individuals will be represented by a gray area.

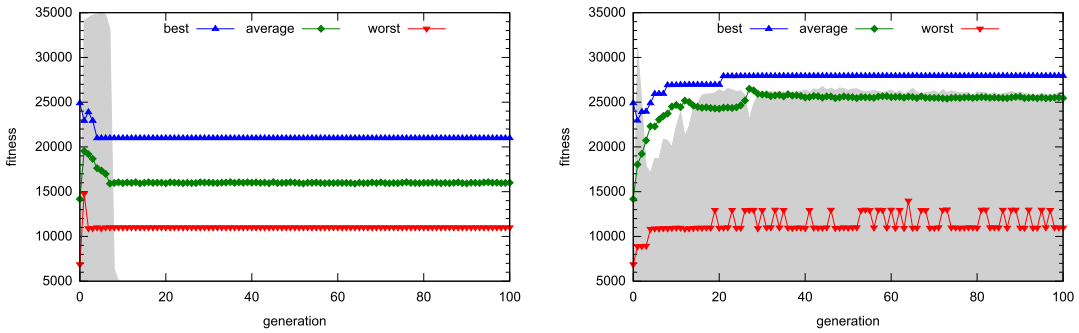
The fitness function is calculated as a combination of correctness and length. For each evaluation that matches the target function, the fitness is increased by 1000, and then the length of the word is subtracted from the fitness value. That is the theoretical maximum of the fitness value is 32000-1.

The result of a test run over a population of 10000 individuals is shown in Figure 2.7. When strict evaluation is used, one can observe a strange phenomenon: at step 5 the best fitness drops to approximately 21000, meaning a 21 bit match, and then it remains constant. A few steps later the number of invalid individuals experiences a dramatic decrease. The reason behind this can be found by checking the individuals in the population, especially those that contain a single symbol. These individuals have relatively high fitness values and they are very likely to produce valid offsprings. For example $\bar{x}_2$ has 19 matches, and x_3 has 21 matches. In this particular test the first of those individuals appeared in generation 4, and by generation 9 the whole population was taken over by these.

This means that using the standard GA operators introduced an unwanted bias: the process preferred those individuals that are likely to produce valid offsprings, instead of those with good matches, although using permissive evaluation this problem could be eliminated.

The other option is to change the operators to produce valid individuals with a higher probability. However, we would like to stay within the simple framework of GA, therefore we want to avoid extensive checks, for example enumerating all the parentheses. Thus only the mutation operator is changed and the following rules are implemented:

- A conjunction operator symbol is replaced by a disjunction operator symbol, and vice versa
- Variable symbols are replaced by another, randomly selected variable symbol regardless of negation
- If an opening parenthesis found then another random locus is selected, and if a closing parenthesis is found then these two are removed

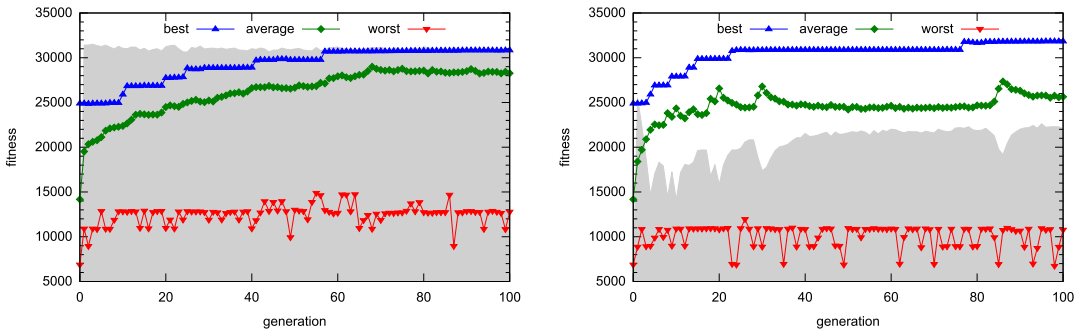


(a) strict evaluation (b) permissive evaluation

Figure 2.7: GA tests with standard mutation

- If a closing parenthesis found then another random locus is selected, and if an opening parenthesis is found then these two are removed
- If a variable or an operator is selected, then with 20% probability another locus is selected randomly, and a pair of parentheses is inserted if possible.

These rules still not guarantee correct individuals, but the probability of creating invalid individuals is low, and the checks can be carried out in constant time. The results of the test runs using the safe mutation operator are shown in Figure 2.8.



(a) strict evaluation (b) permissive evaluation

Figure 2.8: GA tests with safe mutation

The best individuals found in the four different test runs are summarized in Table 2.2, and the found Boolean expressions are listed below.

$$f_{std,strict} = x_3$$

$$f_{std,permissive} = \bar{x}_4 x_3 \bar{x}_1 \vee ((\bar{x}_2 \bar{x}_4 \bar{x}_0 \bar{x}_1) \vee (x_0 x_3 \bar{x}_1)) \vee ((\bar{x}_2 x_3)) x_1$$

$$\begin{aligned} f_{safe,strict} &= \bar{x}_4(\bar{x}_3 x_1(x_1 \vee \bar{x}_4 x_4 x_1(((\bar{x}_0)))))(x_2) \vee x_0(x_3(\bar{x}_1)) \vee (x_1)(((x_1 \vee (\bar{x}_4)))x_3 \bar{x}_2) \\ &\vee ((x_1)\bar{x}_0(((((x_0)(((\bar{x}_1))) \vee \bar{x}_2 \vee \bar{x}_2))))x_4) \\ &\vee ((\bar{x}_4) \vee x_4)x_2 \bar{x}_4 \bar{x}_2 x_3 \bar{x}_2 \vee \bar{x}_4(\bar{x}_1((((\bar{x}_4)))))(\bar{x}_0)(\bar{x}_1 \vee \bar{x}_2)(x_1 \vee x_0 \bar{x}_1 \vee (\bar{x}_2) \vee x_3)((\bar{x}_1))) \end{aligned}$$

$$
\begin{aligned}
f_{safe,permissive} &= x_1((((\bar{x}_3 \vee (\bar{x}_1))))x_3\bar{x}_2) \\
&\vee \quad (((\bar{x}_1(\bar{x}_1)) \vee (\bar{x}_2)))x_3(x_1 \vee x_0\bar{x}_1x_0)(((\bar{x}_4x_1)) \vee x_2 \vee x_4) \\
&\vee \quad (((x_2(\bar{x}_1)) \vee (\bar{x}_2))\bar{x}_0((x_1 \vee \bar{x}_3x_3) \vee x_0)(((x_4)) \vee x_2 \vee x_3) \\
&\vee \quad \bar{x}_1\bar{x}_4\bar{x}_4\bar{x}_0x_3x_3 \vee \bar{x}_1\bar{x}_4\bar{x}_0\bar{x}_4\bar{x}_0\bar{x}_2((\bar{x}_4) \vee x_2) \vee x_2\bar{x}_4x_2\bar{x}_3(x_1))
\end{aligned}
$$

mutation	evaluation	result	matches	length	introns
standard	strict	`0x00FF00FF`	21	1	0
standard	permissive	`0x80FC0074`	28	35	4
safe	strict	`0x83FC2074`	31	157	0
safe	permissive	`0x83BC2074`	32	147	10

Table 2.2: The best individuals found by GA

These results demonstrate various properties of genetic algorithms. On the positive side, GAs can find good approximations for complex problems even without using background information, as shown by the first results using standard GA operators. However, Figure 2.7 also shows that in certain cases GA can easily get stuck in local optima. It is also obvious that GAs cannot guarantee the syntactical correctness of the phenotype and the attempts to handle invalid individuals might introduce an undesired bias. To overcome some of these issues, *genetic programming* extends the concept of genetic algorithms from strings to complex structures, as it will be presented in the following section. There are also other approaches for syntactically correct evolutionary algorithms, some of them will be discussed in Chapter 4.

2.3 Genetic programming

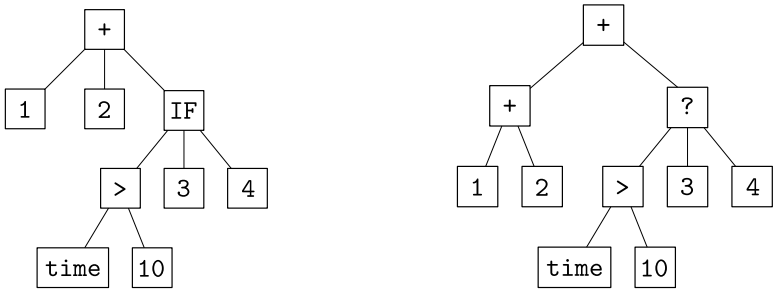
Genetic programming (*GP*) was developed based on genetic algorithms to generate computer programs automatically by means of evolution. [22] First, GP was used to evolve LISP programs, but now GP has versatile applications from circuit design to classification algorithms. The common in these applications is the high-level data structure. In the first application S-expressions and expression trees were used, and it is considered as the standard data type of GP. The necessity of using tree-like structures was shown in the previous section, and can also be illustrated with the following example. Let us take two logical expressions over the alphabet $\{A, B, C, \wedge, \vee, \neg\}$, for example $A \wedge B \vee C$ and $\neg A \vee C \wedge B$. A GA crossover may result in invalid expressions, like $A \wedge BC \wedge B$ and $\neg A \vee \vee C$. However, if trees are used, the obtained expressions can easily be kept valid, as it will be shown in this section. The outline of the algorithm and the selection methods need not be changed, that is in a GP algorithm they are the same as in genetic algorithms.

It must be mentioned that there are several interpretations of the term genetic programming. Sometimes any kind of EA operating on trees or complex structures is referred to as GP. Sometimes researchers, however, restrict the definition to the evolution of S-expressions or other *executable structures*. In this thesis the term genetic programming is used to describe evolutionary algorithms that are operating on syntactical or structural descriptions, such as S-expressions, derivation trees or flow-graphs.

2.3.1 Data structure

As mentioned previously, the most common data type of genetic programming is the abstract syntax tree used to describe expressions of a programming language, for example LISP symbolic expressions, also known as *S-expressions*.

Example 2.10 *(GP Data structure)*
Let us take the S-expression `(+ 1 2(IF(>time 10)3 4))`. This means add 1 and 2 and 3 (if time is greater than 10) or 4 (otherwise). In C-style syntax this is `1+2+((time>10)?3:4)`. The trees for these expressions, that is the GP individuals can be seen in Figure 2.9.



(a) LISP S-expression (b) C-style expression

Figure 2.9: Examples for GP individuals

A very important restriction on GP objects is the *closure property*. [22, 34] 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 and a subtree representing an argument might be replaced by a subtree representing any kind of expression. On the other hand, the closure property includes *evaluation safety*, meaning that every possible expression represented by a subtree must evaluate to a result, so that it is possible to evaluate the solutions and assign fitness values to them. It must be mentioned that there are ways to remove these restrictions, as it will be shown in Section 4.1.1.

2.3.2 Mutation

GP mutation replaces a subtree with a randomly generated subtree. In practical applications this operator is usually parameterized not to allow arbitrary large subtrees to be removed or inserted. This parameter can be considered as the counterpart of GA mutation radius.

Example 2.11 *(GP mutation)*
Let us take an expressions over the alphabet $\{A, B, C, \wedge, \vee, \neg, (,)\}$. For example $(A \vee B) \wedge \neg C$. A possible mutation of this expression is shown in Figure 2.10. The result is the valid expression $(C \vee \neg A) \vee \neg C$. From this example it can also be seen that canonical GP cannot guarantee anything about the semantics. For example the result of this mutation is an always false expression.

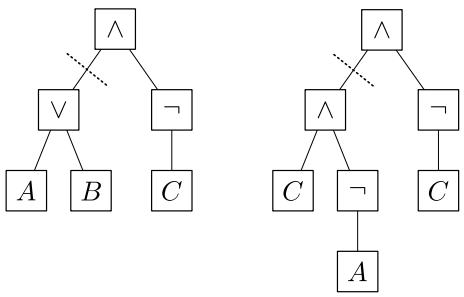


Figure 2.10: Example for GP mutation

2.3.3 Crossover

Crossover works similarly to mutation. Given two trees, a node is selected in both and then the subtrees are swapped.

Example 2.12 *(GP crossover)*
Let us take two logical expressions over the alphabet $\{A, B, C, \wedge, \vee, \neg\}$, $A \wedge B \vee C$ and $\neg A \vee C \wedge B$. An example for a crossover using these expressions are parents is shown in Figure 2.11. The results are expression $\neg A \vee C$ and $A \wedge B \vee C \wedge B$. It is easy to see that any subtree can be selected for crossover, the resulting expressions will be valid. This holds for all logical expression represented in tree-form.

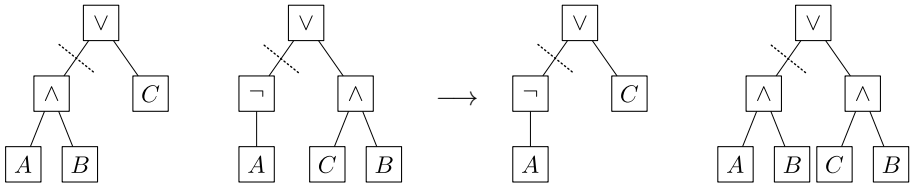


Figure 2.11: Example for GP crossover

The GP operators may have several parameters, for example the height of the changed subtree can be maximized. Other kind of tree structures can also be used as representation, and there is GP approach not using trees, called *linear GP*. [7] Further operators can also be defined for example *permutation*, or *encapsulation*. Another technique is the evolution of so called *automatically defined functions* (*ADF*s) [27], where subtrees can be reused, just like methods or functions in computer programs.

2.3.4 GP example

Let us consider an example for a GP process. The goal is to find the Boolean function defined in Table 2.1 and used for the GA example. In our solution the internal nodes of the GP individual are the Boolean operators $\wedge$ and $\vee$, whereas the leaves are the variables and the negated variables. Even though this means that subtrees cannot be negated, it will not introduce a restriction, because of the fact that $\neg(A \wedge B) = \neg A \vee \neg B$ and $\neg(A \vee B) = \neg A \wedge \neg B$. The fitness function will be the length of the expression. Here the length is defined as the number of the variables and the binary operators. The length of the negated variables is 1. To ensure that the expression stays equivalent to the target function, an evaluation is done for each individual, and if the expression is no longer equivalent, a fitness penalty is given.

To be able to write expressions in a short form, negated variables are denoted by $\bar{x}_1 \dots \bar{x}_5$, and the $\wedge$ operation is not written out explicitly. So for example $x_1\bar{x}_2$ means $x_1 \wedge (\neg x_2)$. The results of a GP optimization run are the following.

The length of the starting expression is 127:

$$\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4\bar{x}_5 \vee \bar{x}_1\bar{x}_2x_3x_4\bar{x}_5 \vee \bar{x}_1\bar{x}_2x_3x_4x_5 \vee$$
$$\bar{x}_1x_2\bar{x}_3\bar{x}_4\bar{x}_5 \vee \bar{x}_1x_2\bar{x}_3x_4\bar{x}_5 \vee \bar{x}_1x_2\bar{x}_3x_4x_5 \vee \bar{x}_1x_2x_3\bar{x}_4\bar{x}_5 \vee \bar{x}_1x_2x_3\bar{x}_4x_5 \vee$$
$$x_1\bar{x}_2\bar{x}_3x_4\bar{x}_5 \vee$$
$$x_1x_2\bar{x}_3\bar{x}_4x_5 \vee x_1x_2\bar{x}_3x_4\bar{x}_5 \vee x_1x_2\bar{x}_3x_4x_5 \vee x_1x_2x_3\bar{x}_4x_5$$

The population contained 1000 individuals and the process was run for 1000 steps. It took about 8 minutes on a Pentium III system. The length of the Boolean expressions during the

evolution process are shown in Figure 2.12. The length of the shortest expression after 1000 steps was 71:

$$\bar{x}_0\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4 \vee \bar{x}_0\bar{x}_1x_2x_3 \vee \bar{x}_0x_1\bar{x}_2\bar{x}_4 \vee \bar{x}_0x_1\bar{x}_2x_3 \vee x_2\bar{x}_0x_1x_2\bar{x}_3 \vee x_0x_3\bar{x}_2\bar{x}_4 \vee x_0x_1\bar{x}_2(x_4 \vee x_4) \vee x_0x_1x_2\bar{x}_3x_4$$

From this example it can be seen again that a simple GP process does not consider the semantics of the evolved objects. When we want to further minimize the last expression manually, it is easy to point out that $(x_4 \vee x_4)$ could be replaced by x_4. This is because GP (like all EA processes) makes the decision based only on the fitness value.

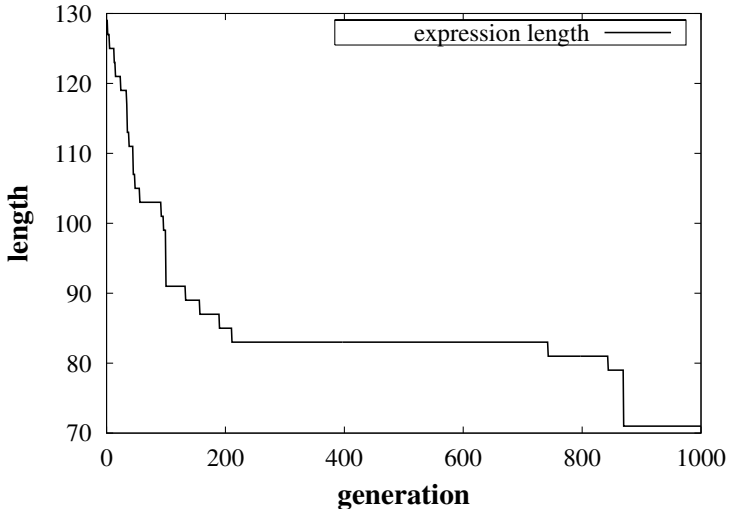


Figure 2.12: Results of the GP test problem

Note that it is common to run evolutionary processes several times, because due to their stochastic nature, independent runs can come up with different solutions, some of which might be better than the results of the first run. For example after 1000 steps of another run with exactly the same parameters, a shorter expression appeared with a length of 57:

$$\bar{x}_0\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4 \vee \bar{x}_0\bar{x}_1x_2x_3 \vee \bar{x}_2\bar{x}_4\bar{x}_0x_1 \vee x_3\bar{x}_2x_4x_1 \vee \bar{x}_0x_1x_2\bar{x}_3 \vee x_0\bar{x}_2x_3\bar{x}_4 \vee x_4x_1\bar{x}_3x_0$$

This example also illustrates the shortcoming of GP that it ignores the semantics of the evolved individuals. To find an appropriate expression, we used the total disjunctive normal form for the initial population, instead of randomly generated individuals as it is usual with evolutionary algorithms. Unfortunately if we decided to randomly generate the initial population, most of it would contain logical expressions describing a different Boolean function. Even using our way to initialize the population, during the evolutionary process many individuals are created that are syntactically correct, but semantically different. These are excluded from the parent population using fitness penalty, which might be effective but very inefficient, because it narrows down the search area and slows down the convergence. This issue will be revisited later in this thesis and a solution is proposed in Section 5.2.

3 Grammars and formal languages

In this chapter the basics of grammars and formal languages are summarized. For a detailed description the interested reader might check [20], or any textbook on this topic.

An *alphabet* is a finite, non-empty set of symbols, or *letters*, usually denoted by Σ. An associative operation, called *concatenation* is defined, denoted by $a_1 \cdot a_2$ for $a_1, a_2 \in \Sigma$. The result of concatenating zero or more symbols is called *word*, and is usually denoted as a sequence of letters without the concatenation symbol. A special word is the *empty word*, which is denoted by λ.[1] The set of all words generated from the symbols in Σ using concatenation is denoted by Σ^*. In other words $(\Sigma^*, \cdot)$ is the free monoid over Σ, with identity element λ. The subsets of Σ^* are called *formal languages*.

3.1 Generative grammars

The most commonly used tools to describe how to build languages from the symbols of an alphabet are the *formal grammars* as proposed by Chomsky. [8]

Definition 3.1 *(Generative grammar)*
A *generative grammar* G is a 4-tuple $G = (\mathcal{N}, \Sigma, \mathcal{P}, S)$, where

(i) $\mathcal{N}$ is an alphabet of *nonterminal symbols*,

(ii) Σ is an alphabet of *terminal symbols*, with $\mathcal{N} \cap \Sigma = \emptyset$,

(iii) $\mathcal{P} \subset (\mathcal{N} \cup \Sigma)^+ \times (\mathcal{N} \cup \Sigma)^*$ is a nonempty finite set of *rewriting rules*, and

(iv) $S \in \mathcal{N}$ is the *start symbol*.

The rewriting rules have the form (α, β), which is written as $\alpha \to \beta$, where α is called the left-hand side and β is called the right-hand side of the rule. Note that α has to contain at least one nonterminal. Several rules can be written together if their left-hand side is the same, for example $S \to abc$ and $S \to bS$ can be written as $S \to abc \mid bS$. In this thesis the following conventions are used: nonterminals are denoted by capital letters $(A, B, C, \ldots)$, terminals are denoted by small letters $(a, b, c, \ldots)$. Words containing only terminals are also denoted by small letters, but from the end of the English alphabet, starting with u $(u, v, w, \ldots)$. Arbitrary words are denoted by the letters of the Greek alphabet $(\alpha, \beta, \gamma, \ldots)$, except λ, which denotes the empty word.

[1] Another usual notation for the empty word is ε.

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.

Definition 3.2 *(Direct 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 $\alpha \to \beta$ rule in $\mathcal{P}$ such that $\gamma = \varphi\alpha\psi$ and $\delta = \varphi\beta\psi$.

Definition 3.3 *(Derivation)*
Derivation is the reflexive transitive closure of the direct derivation relation, that is $\Rightarrow_G^*$. If it is unambiguous, the grammar is not indicated, and the derivation relation is denoted by $\Rightarrow^*$.

Remark 3.4
Derivation is reflexive, thus $\gamma \Rightarrow^* \gamma$. Derivation is also transitive, therefore derivations can be concatenated, that is if $X \Rightarrow^* \alpha Y \gamma$ and $Y \Rightarrow^* \beta$ then $X \Rightarrow^* \alpha\beta\gamma$.

Definition 3.5 *(Derivation sequence)*
Derivation sequence for a given grammar G 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.

Note that $\gamma \Rightarrow_G^* \delta$ if and only if there exists a derivation sequence such that $\alpha_1 = \gamma$ and $\alpha_n = \delta$. Often a derivation sequence is also called derivation. With the help of derivation the generated language can be defined as follows.

Definition 3.6 *(Language generated by a grammar)*
A language generated by grammar $G = (\mathcal{N}, \Sigma, \mathcal{P}, S)$ is

$$L(G) = \{u \in \Sigma^* \mid S \Rightarrow_G^* u\}.$$

Sometimes, however, one does not need the whole generating power of the grammars. Therefore the allowed form of the rewriting rules can be restricted or changed, so the class of the possibly generated languages also changes. Chomsky introduced several types of grammars, they are described by the following definition.

Definition 3.7 *(Grammar types)*
A grammar $G = (\mathcal{N}, \Sigma, \mathcal{P}, S)$ is

- *type 0* or *phrase structure*, if there are no restrictions.
- *type 1* or *context sensitive*, if rules have the form $\alpha A \beta \to \alpha\delta\beta$, where $\alpha, \beta, \delta \in (\mathcal{N} \cup \Sigma)^*$ and $\delta \neq \lambda$. The only exception is rule $S \to \lambda$ (that is $\alpha = \delta = \beta = \lambda$), but in this case S cannot be on the right-hand side of any other rule.
- *type 2* or *context-free*, if rules have the form $A \to \alpha$, where $A \in \mathcal{N}$ and $\alpha \in (\mathcal{N} \cup \Sigma)^*$.
- *type 3*, *regular* or *right linear*, if rules have the form $A \to u$ or $A \to uB$, where $A, B \in \mathcal{N}$ and $u \in \Sigma^*$.

Definition 3.8 *(Language types)*
A language $L \subseteq \Sigma^*$ has type i for $0 \leq i \leq 3$, if there exists a grammar G of type i such that $L = L(G)$. The class of languages of type i is denoted by $\mathcal{L}_i$.

Remark 3.9
A regular grammar is context-free. Furthermore, it can be shown that $\mathcal{L}_3 \subsetneq \mathcal{L}_2 \subsetneq \mathcal{L}_1 \subsetneq \mathcal{L}_0$.

3.2 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. Since the derivation starts with a single symbol, the structure representing the full derivation will be a single connected acyclic graph, that is a tree.

Definition 3.10 *(Derivation tree)*
Given a context-free grammar $G = (\mathcal{N}, \Sigma, \mathcal{P}, S)$. For a terminal symbol $a \in \Sigma$ the set of derivation trees rooted in a is $\mathcal{T}(a) = \{a\}$. For a nonterminal symbol $A \in \mathcal{N}$ the set of derivation trees rooted in A is the smallest set $\mathcal{T}(A)$, where

(i) The tree having exactly one node (its root) labeled with A is in $\mathcal{T}(A)$. This tree is denoted by A.

(ii) If $A \to \lambda \in \mathcal{P}$, then the tree having a root node labeled with A and exactly one successor node of the root labeled with λ is in $\mathcal{T}(A)$. This tree is denoted by $A[\lambda]$.

(iii) If $A \to X_1 \dots X_n \in \mathcal{P}$ and $T_i \in \mathcal{T}(X_i)$ for $i = 1, \dots, n$, then the tree having a root node labeled with A, n successors labeled with $X_1, \dots, X_n$ in this order from left to right and $T_1, \dots, T_n$ subtrees rooted in the nodes denoted by $X_1, \dots, X_n$ is also an element of $\mathcal{T}(A)$. This tree is denoted by $A[T_1, \dots, T_n]$.

Definition 3.11 *(Frontier of a derivation tree)*
Let T be a derivation tree rooted in X. The frontier of T is denoted by $fr(T)$, and defined as follows.

(i) If $T = X$ then $fr(T) = X$.

(ii) If $T = X[\lambda]$ then $fr(T) = \lambda$.

(iii) If $T = X[T_1, \dots, T_n]$ then $fr(T) = fr(T_1) \cdot \ldots \cdot fr(T_n)$.

One can also use any property defined for common rooted trees, for example height or size. In this thesis some of these properties will be used with the following remark: the *breadth* of a derivation tree is the number of leaves, whereas the *width* is the length of the frontier, that is the total number of symbols in the leaves excluding λ. By definition of the derivation tree, the difference is exactly the number of λ-nodes, but a compact implementation might merge neighboring nodes with terminal labels, significantly reducing the number of leaves, that is the breadth of the tree.

Example 3.12
An example for a derivation tree is shown in Figure 3.1. The tree is $S[aS[ab]bS[ab]]$ for grammar $G = (\{S\}, \{a, b\}, \{S \to aSbS \mid ab\}, S)$. The height of this tree is 2, the frontier is *aabbab*, the width is 6 and the breadth is 6 for the standard tree and 4 for the compact tree.

(a) standard (b) compact

Figure 3.1: Example for derivation tree

Theorem 3.13 *(Relation of derivations and derivation trees)*
For any $X \in (\mathcal{N} \cup \Sigma)$ and $\alpha \in (\mathcal{N} \cup \Sigma)^*$

$$X \Rightarrow^* \alpha \text{ if, and only if } \exists T \in \mathcal{T}(X) : fr(T) = \alpha$$

Proof 3.14
A constructive proof is given by induction.

(i) $X \Rightarrow^* \alpha \rightarrow \exists T \in \mathcal{T}(X) : fr(T) = \alpha$ is shown by induction on the length of the derivation.

- For a trivial derivation $X \Rightarrow^* X$ tree X is in $\mathcal{T}(X)$ by item (i) of Definition 3.10 and $fr(X) = X$ by item (i) of Definition 3.11.
- Now let us assume that the statement is true for derivations of length n. Let us take a derivation of length $n+1$: $\alpha_0, \alpha_1, \ldots, \alpha_n, \alpha_{n+1}$, where $\alpha_0 = X$ and $\alpha_{n+1} = \alpha$.
  By the definition of the derivation sequence, $\alpha_n \Rightarrow \alpha_{n+1}$, thus $\exists \varphi, \psi, \beta \in (\mathcal{N} \cup \Sigma)^*$ and $\exists Y \in \mathcal{N}$ such that $\alpha_n = \varphi Y \psi$, $\alpha_{n+1} = \varphi \beta \psi$ and $p : Y \rightarrow \beta \in \mathcal{P}$.
  By the induction assumption $\exists T \in \mathcal{T}(X)$ such that $fr(T) = \alpha_n$. It means that

  $$fr(T) = \alpha_n = \varphi Y \psi = \varphi \cdot fr(Y) \cdot \psi$$

  According to item (i) of Definition 3.10, for each letter Z_i in word β, tree Z_i is in $\mathcal{T}(Z_i)$. Therefore by item (iii) tree $T' = Y[Z_1, \ldots, Z_k]$ is in $\mathcal{T}(Y)$, which in turn means that taking tree T and replacing leaf Y with tree T', the resulting tree T'' is in $\mathcal{T}(Y)$. The frontier of the resulting tree by applying items (iii) then (i) of Definition 3.11 is:

  $$fr(T'') = \varphi \cdot fr(T') \cdot \psi = \varphi \cdot fr(Z_1) \cdot \ldots \cdot fr(Z_k) \cdot \psi = \varphi Z_1 \ldots Z_k \psi = \varphi \beta \psi = \alpha_{n+1}$$

(ii) $\exists T \in \mathcal{T}(X) : fr(T) = \alpha \rightarrow X \Rightarrow^* \alpha$ is shown by induction on the definition.

- For tree $T = X$ with $fr(T) = X$ we can use the trivial derivation $X \Rightarrow^* X$.
- For tree $T = X[\lambda]$ with $fr(T) = \lambda$ there exists a rule $p : X \rightarrow \lambda$, thus $X \Rightarrow \lambda$.
- For tree $T = X[T_1, \ldots, T_n]$ with T_i having a root node labeled with X_i there must be a rule $p : X \rightarrow X_1 \ldots X_n \in \mathcal{P}$. Furthermore $fr(T) = fr(T_1) \cdot \ldots \cdot fr(T_n)$. The induction assumption is that $X_i \Rightarrow^* fr(T_i) = \alpha_i$. By applying Remark 3.4 to concatenate derivations, we get the following:

$$\begin{aligned} X &\Rightarrow^1 X_1 X_2 X_3 \ldots X_{n-1} X_n \\ &\Rightarrow^* \alpha_1 X_2 X_3 \ldots X_{n-1} X_n \\ &\Rightarrow^* \alpha_1 \alpha_2 X_3 \ldots X_{n-1} X_n \\ &\ldots \\ &\Rightarrow^* \alpha_1 \alpha_2 \alpha_3 \ldots \alpha_{n-1} X_n \\ &\Rightarrow^* \alpha_1 \alpha_2 \alpha_3 \ldots \alpha_{n-1} \alpha_n = \alpha \end{aligned}$$

Remark 3.15 *(Left derivation)*
In the above construction a derivation sequence is described, where in each step the leftmost nonterminal is replaced using an appropriate rule. These derivation sequences are called *left derivations*. Usually derivation trees describe more than one possible derivation sequences, but there is a one-to-one mapping between left derivations and derivation trees. Figure 3.2 shows an example for a left derivation and the (partial) derivation trees associated with each step.

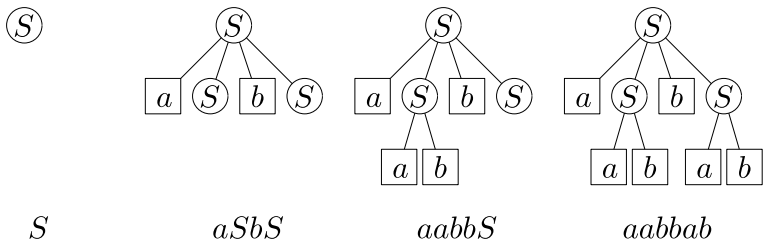


Figure 3.2: Derivation sequence with derivation trees

Corollary 3.16
Given a context-free grammar $G = (\mathcal{N}, \Sigma, \mathcal{P}, S)$. For any $u \in \Sigma^*$ $u \in L(G)$ if and only if there exists $T \in \mathcal{T}(S)$ such that $fr(T) = u$.

Remark 3.17
According to the definition of derivation trees, the frontier can be an arbitrary string. Due to Corollary 3.16, however, derivation trees with terminal frontier are of particular interest. These are referred to as *complete derivation trees*, whereas trees having at least one nonterminal symbol in the frontier are called *partial derivation trees*. The set of complete derivation trees rooted at the start symbol is denoted by $\mathcal{T}(G)$, that is

$$\mathcal{T}(G) = \{T \in \mathcal{T}(S) \mid fr(T) \in \Sigma^*\}$$

Using this definition Corollary 3.16 can be written as follows.

$$fr(\mathcal{T}(G)) = \bigcup_{T \in \mathcal{T}(G)} \{fr(T)\} = L(G)$$

From the derivation tree representation one can infer an interesting property of the context-free grammars. As both alphabets are finite, if a derivation tree is sufficiently large, there is a path between the root and a leaf that contains the same nonterminal symbol at least twice. Taking these two occurrences the path between them can be removed from the tree, or it can be inserted multiple times causing two parts of the frontier to be multiplied. This property is stated by the following lemma.

Lemma 3.18 *(Bar-Hillel Lemma)*
For every context-free language L there exists an integer $p \geq 1$ such that any word $u \in L$ with $|u| \geq p$ can be written as $u = vwxyz$ such that $|wxy| \leq p$, $|wy| \geq 1$ and $\forall n \in \mathbb{N}$ $vw^nxy^nz \in L$.

3.2.1 Regular grammars and derivation trees

As mentioned earlier regular grammars are also context-free grammars, therefore the concept of derivation trees can be applied to them as well. Regular grammars only allow a single nonterminal on the right-hand side of the rules, and it can only be on the last position. This means that for every node of a derivation tree all but the last successor nodes must be leaves.

3.2.2 Derivation tree sizes

In this thesis derivation trees will be used as data types, therefore it is interesting to examine the relation between the size of the words, derivations and derivation trees. For examining this relation, an important property is the number of terminal symbols on the right-hand side of the rules. Therefore, in this thesis we will use the following definition.

Definition 3.19 *(Size of a rule)*
Given a context-free grammar $G = (N, A, P, S)$. For a rule $r : A \to \beta$ the size of r is the number of symbols in β denoted by $|r|$, that is $|r| = |\beta|$. The terminal size of r is the number of terminal symbols in β, and it is denoted by $|r|_\Sigma$. The size of the longest rule is denoted by $|\hat{r}|$, that is

$$|\hat{r}| = \max_{r \in \mathcal{P}} |r|$$

As mentioned before, the length of a derivation sequence $\alpha_0, \alpha_1, \ldots, \alpha_k$ is the number of derivation steps, that is k. The size of a tree T is the number of nodes, which can be calculated recursively, as one plus the sum of the sizes of the subtrees. The length of the derived word is calculated as usual. Between these values a correlation can be found as shown by the following theorem.

Theorem 3.20 *(Correlation of derivation related sizes)*
Given a context-free grammar $G = (N, S, P, S)$. 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

$$s \quad = \quad \mathcal{O}(k) \tag{3.1}$$

Furthermore, assume that

$$\frac{\sum_{i=1}^{k} |r_i|_\Sigma}{k} \quad \geq \quad 1, \tag{3.2}$$

where $r_i = A_i \to \beta_i$ is the rule applied in derivation step i. In this case the following also holds

$$k \quad \leq \quad n \tag{3.3}$$

Remark 3.21
The assumption described by Equation 3.2 means that on average at each step at least one nonterminal symbol is added to the word. Note that if the grammar is in Greibach normal form, this condition is always true.

Proof 3.22
The size of the derivation tree is the number of nodes introduced by each rule, plus one for the start symbol. The number of nodes introduced by a rule is the number of symbols in the right-hand side. That is

$$s = 1 + \sum_{i=1}^{k} |r_i| \leq 1 + \sum_{i=1}^{k} |\hat{r}| = 1 + k \cdot |\hat{r}| = \mathcal{O}(k)$$

The length of the derived word is the sum of the nonterminal symbols added by the rules, that is if Equation 3.2 holds

$$k \leq \sum_{i=1}^{k} |r_i|_\Sigma = n$$

□

This theorem means that using a derivation tree is asymptotically not worse than using a list of the applied rules. Furthermore, if Equation 3.2 holds, the derivation tree is asymptotically not larger than the word itself.

3.3 Attribute grammars

It is possible to increase the generating power of context-free grammars by defining attributes for the symbols and assigning values to these attributes during derivation. [3] For example, without using attributes, language $\{a^n b^n c^n\}$ cannot be generated by any context-free grammar[2], but there is an attribute grammar that generates this language.

In this thesis we will not use the additional generating power of attribute grammars, but the concept of assigning attributes to symbols is very useful, thus a short definition is given here.

Definition 3.23 *(Attribute grammar)*
An *attribute grammar* AG is a quadruple $AG = (G, SD, AD, \mathcal{R})$, where

(i) G is a context-free grammar,

(ii) SD is the *semantic domain*, defining the attribute types, functions and relations

(iii) AD contains the *attribute descriptions* defining the set of synthesized and inherited attributes $Attr_S$ and $Attr_I$ respectively, the types of these attributes and the assignment of attributes to symbols of G. Attributes assigned to a symbol are denoted by attaching the attribute to the symbol using a dot, for example $S.a$

(iv) $\mathcal{R} = \{\mathcal{R}(p) \mid p \in \mathcal{P}\}$ is a family of sets defining the *semantic rules* for each rewriting rule of grammar G

Remark 3.24
It happens often that one nonterminal symbol appears multiple times in a given rule. In this case, to avoid confusion when defining semantic rules, all occurrences are indexed. For example an attribute *length* for the left-hand side symbol of a rule $S \to aSbS$ can be defined as follows:

$$S_0 \to aS_1bS_2 \qquad S_0.length := S_1.length + S_2.length + 2$$

When a derivation tree is created using an attribute grammar, the semantic rules are evaluated for each rule that is applied. If a rule defines a calculation schema for an attribute, then the calculated value is assigned to the given *occurrence* of the attribute in the derivation tree. Such a derivation tree is called *decorated derivation tree*. Sometimes the calculation schema defines a value based on the values found in the parents. These attributes are called *inherited attributes*. In other cases the values in the parents are calculated based on the values in the subtrees. These are called *synthesized attributes*, and they can be calculated after the subtrees have been constructed.

In this thesis the following assumptions will be used regarding attributes. An attribute a is called *inherited* if for each rule $X \to Y_1Y_2 \dots Y_n$ the calculation schema is the following:

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

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

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

More formal definitions are not required for the purposes of this thesis, nevertheless they can be found in the literature. [12] Informally inherited attributes can also be called *top-down* attributes, whereas synthesized attributes can be called *bottom-up* attributes.

The semantic rules not only include rules defining values, but also rules that define *conditions*. These conditions determine when a rewriting rule can be applied based on the values of the attributes.

[2]It is the classical example of a language that cannot be generated by context-free grammars. This fact can easily be proven using the Bar-Hillel lemma.

Example 3.25
The following attribute grammar generates language $\{a^n b^n c^n \mid n \in \mathbb{Z}^+\}$.

- $G = (\{A, B, C, S\}, \{a, b, c\}, \{S \to ABC, A \to a \mid aA, B \to b \mid bB, C \to c \mid cC\}, S)$
- The semantic domain is $(\mathbb{Z}^+, \{+\}, \{=\})$, that is the set of positive integers with the addition operator and equality relation.
- We define one attribute: *count*, it is assigned to nonterminals A, B and C.
- The semantic rules are defined as follows

$$\begin{array}{rl} S \to ABC & A.count = B.count = C.count \\ A \to a & A.count := 1 \\ A_0 \to aA_1 & A_0.count := A_1.count + 1 \\ B \to b & B.count := 1 \\ B_0 \to bB_1 & B_0.count := B_1.count + 1 \\ C \to c & C.count := 1 \\ C_0 \to cC_1 & C_0.count := C_1.count + 1 \end{array}$$

There is one semantic rule defining a condition allowing only derivations where the three subtrees for a^*, b^* and c^* have the same number of letters. The rest of the semantic rules are used to define the value for the *count* attribute, counting the number of letters.

As it is shown by the example above, a straightforward way to define the semantic domain SD is to use the set of integer or real numbers and the usual arithmetic operators and relations.

4

Derivation tree based genetic programming

Evolutionary algorithms can be efficiently used to solve various problems without using any information on the problem domain. Only a representation form and an evaluation function is required. This property is called *black box principle*, and this simplicity is one of the most appreciated characteristics of these algorithms.

However, this setup has a disadvantage; sometimes invalid solution candidates are produced. Changing the evaluation function to check, replace and correct these candidates requires processing time. Modifying the algorithm to avoid such individuals, or changing the evaluation function to be able to handle them might even introduce an unexpected bias. *Derivation tree based genetic programming* (*DTGP*) gives a solution for this problem. It is a modification of canonical genetic programming to use derivation trees over a context-free grammar as individuals to introduce syntactical restrictions without significantly increasing the complexity of the algorithm.

In this chapter the issues with the canonical evolutionary search are presented, then existing grammar based approaches are discussed. In the main part DTGP is defined, the data type is introduced and the operators are constructed. The chapter is concluded with an example to illustrate how DTGP works. In later chapters it will also be shown that DTGP not only solves the problem of invalid individuals, but also provides further improvements over canonical GP.

4.1 Canonical evolutionary search

A usual setup for evolutionary optimization is shown in Figure 4.1. The set of *hypotheses* H for a given problem is described by a set of *representations* R. In GA terms they are called *phenotypes* and *genotypes* respectively. The elements of R are also called *individuals* when they are part of a population. In most cases only a proper subset of H contains the *solutions* of the problem, let us denote this set by S. The hypotheses outside of this set are referred to as *incorrect hypotheses*. The set of representations describing the solutions is denoted by R_v. The elements of R_v can be evaluated using the fitness function, therefore they are called *valid representations*. The goal of the process is to find an element of R_v that represents an element of S which is optimal with respect to the selected fitness function.

In a common EA the operators work on a population containing elements of set R and nothing guarantees that an operator applied to an element of R_v results an individual within R_v. In other words operators may lead out of R_v. Usually R_v is not even defined explicitly. Let R_{ea} denote the smallest subset of R that is closed under the evolutionary operators and contains R_v. The elements of R_{ea} describe the hypotheses in set H_{ea}.

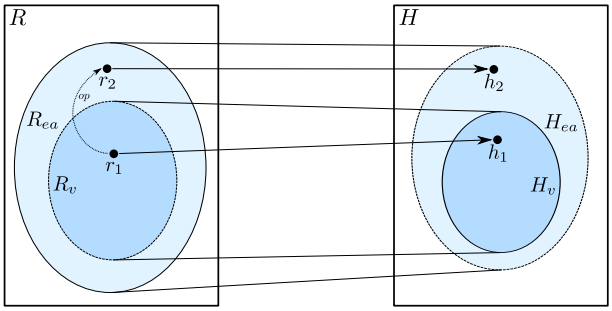


Figure 4.1: Common setup for optimization with EA

4.1.1 Invalid individuals

If the smallest closed subset of the representations is larger than the set of valid representations, that is $R_v \subsetneq R_{ea}$, the algorithm may produce an individual $r' \in R_{ea} \setminus R_v$ such that r' evaluates to $h' \in H_{ea} \setminus S$. This means that an invalid hypothesis is generated. There are several approaches how to handle this issue.

The invalid individual can be thrown away, and a new individual can be generated. This needs extra time and in extreme cases no valid individuals can be generated from certain parents or pairs of parents.

It is also possible to correct the invalid individuals, but this also takes some time and might cause the evolutionary algorithm to be unbalanced. For example if the valid representations are nonnegative integers, and negative numbers are replaced by 0, then 0 will occur with a very high probability. When the absolute value is taken, 0 will only have half of the probability of the positive numbers. This can be more difficult for complex representations like trees.

The invalid individuals can be left in the population and penalized with bad fitness values, but this increases the search space and decreases the useful size of the population, just like in the case of the GA example presented in Section 2.2.6.

All the approaches mentioned above allow invalid individuals to occur and rely on an evaluation, which itself might need additional time to handle them. The other possible approach is not to allow such individuals at all, that is to modify the representation or the evolutionary operators so that $R_{ea} = R_v$. This can be done in three different ways.

Allow any hypothesis The simplest way is to define the problem and the sets H and R 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. [22]

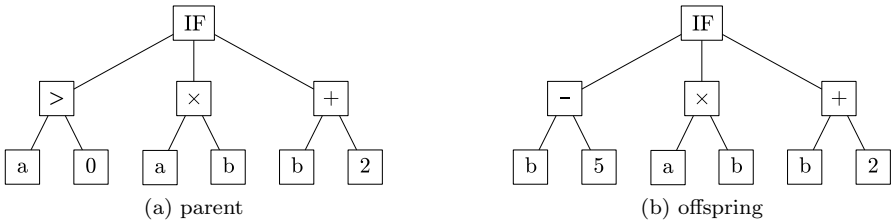
Restrict operators The straightforward approach is to modify the operators. This, however, might not be a simple task. This is how *strongly typed genetic programming* works. [31]

Redefine representations The third possibility is to redefine the set R, so that R_v is closed under the evolutionary operators. This might also require some small changes in the operators. This approach is followed by *grammar guided genetic programming* techniques, such as DTGP.

All the examples mentioned above are special types of genetic programming. This is because syntactical restrictions are easier to implement using high-level data structures, for example expression trees. However, it is also possible to use bitvectors and GA based methods, for which an example will be shown later in this chapter.

4.2 Strongly typed genetic programming

The original GP algorithm uses LISP S-expressions, but the operators are not restricted. This might cause problems as shown in Figure 4.2. The parent tree contains an *if-then-else* branch with three subtrees: a condition and two expressions. However, after applying an evolutionary operator, the condition might be replaced with an arithmetic expression.



(a) parent (b) offspring

Figure 4.2: Syntactical error caused by GP operators

To handle this problem, canonical GP defines how to interpret arithmetical expressions as conditions, and in general it defines implicit conversion for each type. In GP terms it is the closure property that was described in Section 2.3.1. Although it works for S-expressions, it is not a generic solution, and with other data types this issue may still occur.

Strongly typed genetic programming (*STGP*) [31] was directly designed to solve this problem. It defines a set of types, and it assigns types for each function and each parameter. Furthermore, the functions have pre-defined arities as well, that is the signature for each function is defined. Formally it means the following. Let $\mathcal{T}$ be a set of types and $\mathcal{F}$ be a set of function symbols. A signature σ is a triplet, that is an element of $\mathcal{T} \times \mathcal{F} \times \mathcal{T}^*$. Signature $\sigma = (t, f, t_1 t_2 \dots t_n)$ can be written as $t\ f(t_1, t_2, \dots, t_n)$, and it means that function f takes n parameters of type t_1, t_2, $\dots$ t_n respectively, and it returns a value of type t.

An example for a set of signatures can be seen in Figure 4.3. These signatures describe a set of functions built from the usual arithmetical operators $(+, -, *, /)$, logical operators $(\neg, |, \&)$ and the ternary operator if, which takes a logical expression and two real expressions and depending on the value of the logical expression returns the value of the first or the second expression.

real	$+$	(real,real)
real	$-$	(real,real)
real	$*$	(real,real)
real	$/$	(real,real)

a) real functions

bool	$\neg$	(bool)
bool	$\|$	(bool,bool)
bool	&	(bool,bool)
real	if	(bool,real,real)

b) Boolean functions

real	0	()
real	1	()
bool	$false$	()
bool	$true$	()

c) constants

Figure 4.3: Example set of signatures

STGP uses parse trees to represent the solutions, just like canonical GP. However, the operators are restricted. Mutation and crossover are only allowed if the return type of the expression represented by both the old and new subtrees are the same. Furthermore, when a random subtree is generated, it is done based on the signatures, which means the number of the subtrees is the same as the arity of the function represented by the node, and the return types of the subtrees match the parameter types of the given function.

STGP can be used to evolve syntactically correct expressions or even programs. The notion of signatures can be generalized even further to define each possible branching in the evolved trees, although it might be difficult to set it up for certain problems. A disadvantage of STGP is that the operators must follow the signatures, which also increases the time complexity.

4.3 Grammar guided genetic programming

Another approach recently gaining popularity in the GP arena is *grammar guided genetic programming* (*GGGP*) [28], which assumes that the set of valid individuals is a context-free language, and uses context-free grammars to guide the evolutionary process. There are two approaches depending on the data type. *Tree based GGGP* uses derivation trees as representation, and the evolutionary operators work on these trees. DTGP uses this approach. The second approach is *linear GGGP*, where the individuals are binary or integer vectors that represent the derivation sequences.

Tree based GGGP

One of the first experiments were done by Gruau [18], who used context-free grammars and derivation trees to check the validity of the individuals. However, after the verification the derivation trees have been deleted and had to be created again for new individuals.

Whigham [47, 48] represented the individuals in the population as derivation trees and applied the operators directly to them. The limitation of his approach is that the trees cannot be parameterized, thus the operators can be influenced only by global parameters. For instance, in order to keep the trees of the populations smaller than a global MAX_DEPTH parameter (set by the initial population), the oversized trees produced by the operators are simply thrown away.

Grammatical evolution

The most popular linear GGGP method is *grammatical evolution* (*GE*) [36, 32]. 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 outline of this approach is shown in Figure 4.4.

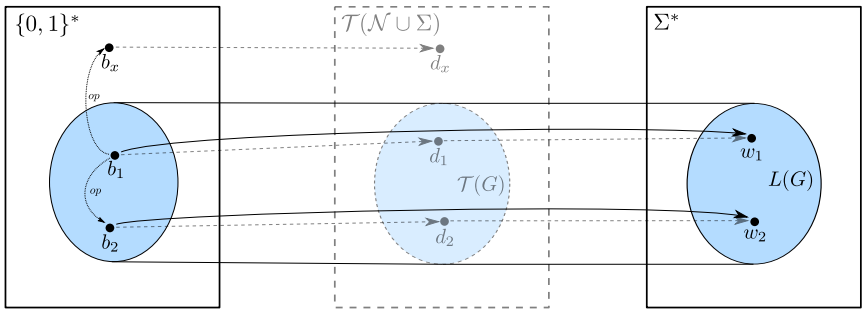


Figure 4.4: Outline of GE

The hypotheses are assumed to be words over an alphabet Σ. The set of solutions are described by a grammar G, thus $R_v = L(G)$. 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.

To recall Remark 3.15, there is a one-to-one mapping between derivation trees and left derivations. Thus it can be assumed that the index vectors represent complete or incomplete derivation trees, that is elements of $\mathcal{T}(\mathcal{N}\cup\Sigma)$, even though these are not constructed explicitly by GE. These trees will be complete derivation trees from $\mathcal{T}(G)$, except in some extreme cases discussed later.

Thanks to this simple representation, GE needs less storage than the tree based approaches, although as shown by Theorem 3.20, there is no asymptotic difference, unless cyclic bitstrings are used. Nevertheless, the evolutionary operators as well as random tree generation are very fast.

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. This might require a significant amount of time. Furthermore, there is no one-to-one mapping between bitstrings and derivations. The problematic cases are the following:

Incomplete derivations might come up if the bitstring does not contain enough rules to finish the derivation. This can be solved by using a cyclic bitstring, that is if the end of the bitstring is reached before the derivation is finished, the process starts again from the beginning of the string.

Infinite derivations might be represented by certain cyclic bitstrings. This issue does not occur if the derivation process is stopped when reaching the end of the string.

Unused suffix might be present if the derivation finishes before reaching the end of the bitstring. These elements are also present in nature; they are called introns and they represent dormant genetic information.

Depending on whether cyclic bitstrings are used or not, the infinite or incomplete derivations have to be detected and either discarded or penalized by bad fitness values. Such an individual is marked by b_x in Figure 4.4. As it can be seen it maps to a tree, but not necessarily a complete finite derivation tree, thus it cannot be mapped to a hypothesis in Σ^*.

4.4 Basics of DTGP

Derivation tree based genetic programming is a tree based grammar guided GP method. Its outline can be seen in Figure 4.5. The representations are derivation trees over a pre-defined context-free

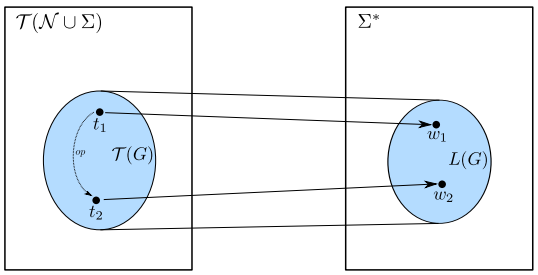


Figure 4.5: Outline of DTGP

grammar G. The solutions, that is the words of the language $L(G)$, can be found at the frontier 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 [47], but DTGP uses parameterized derivation trees to improve the algorithm.

4.4.1 Derivation tree data type

The exact method of storing the derivation trees is not important for the theoretical foundations. However, we will have to refer to certain parts, components or properties of a tree. These are shown in Figure 4.6. A derivation tree is a directed and acyclic graph, where each node has at most one direct predecessor and zero or more direct successors. The direct predecessor is called

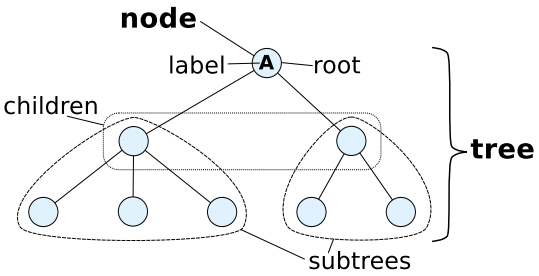


Figure 4.6: Derivation tree data type

parent node, and the direct successors are called *children*. A *subtree* is an arbitrary node with all of its successors. In any subtree there is exactly one node that has no predecessors, it is called the *root*. Given a tree and one of its subtrees. The subtree is called direct subtree, if the root of the tree is the parent of the root of the subtree. In Figure 4.6 all the indicated subtrees are direct subtrees. Each subtree also represents a complete derivation for the label of its root node.

In DTGP not only the labels are stored at the nodes, but other information as well, these are called parameters. Thanks to these parameters our approach has some advantages over the existing tree based GGGP methods, as it will be described in the following sections.

The notations used throughout this thesis are summarized in Table 4.1. 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	usually $T_1, T_2, \ldots$ denote the subtrees of T from left to right
$N, N_1, N_2, \ldots$	node	sometimes node and tree are used interchangeably
$N[T_1, \ldots, T_n]$	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 4.1: Notations for components of the derivation tree data type

4.4.2 Parameterized derivation trees

During the evolutionary process several properties of the derivation trees might be used by the evolutionary operators. For example, when a subtree is selected randomly for a mutation operator, usually there is a limit on the size or the height of the subtree. However, recalculating this data every time could be a huge effort. On the other hand, it is possible to store 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 a limited set of properties, therefore we require that the properties of any 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 $T = N[T_1, \ldots, T_n]$:

$$T.p = f_p(N, T_1, \ldots, T_n)$$

Some commonly used properties, such as the *height* and the *size* of the subtree or the *length* of the subtree frontier depend on the same properties of the subtrees, simplifying the above equation:

$$T.p = f_p(T_1.p, \ldots, T_n.p)$$

Parameter maintenance

A disadvantage of storing parameters in each node is that they have to be maintained during the evolutionary process. When a subtree changes, the algorithm has to update all nodes that store parameters dependent on the changed subtree. However, as we defined the parameters to be *bottom-up*, is it 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, as it is shown in Figure 4.7. Therefore, the parameters can be updated by a simple

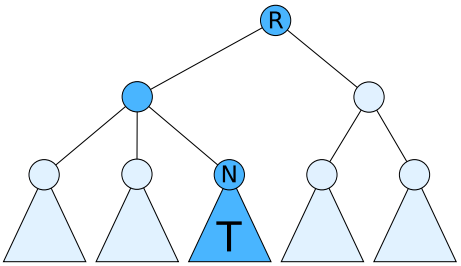


Figure 4.7: Affected nodes after subtree modification

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, as shown in Algorithm 4.1.

```
UPDATE-NODE(node : N)
1  for each p in parameters
2  do p := f_p(N, T_1, ..., T_n)
3  if N.parent ≠ NIL
4     then UPDATE-NODE(N.parent)
```

Algorithm 4.1: Updating the parameters of a node

Parameterized context-free grammars

Parameter calculation can be defined once and used for each node, or it can also be defined for each label. For context-free grammars, however, it is convenient to define calculation functions with each rule. The calculation scheme of standard tree properties, like *size*, is the same for each node, the only difference is whether it is a leaf node or not. However, later other parameters will be introduced which are calculated depending on the applied rule.

In the following chapters parameter calculation definitions will be listed with the rules as shown in the following example for rule $A \to xyABC$ and the *size* property.

$$A_0 \to xyA_1BC \qquad A_0.size = A_1.size + B.size + C.size + 2$$

Note the indices used for nonterminal A. These are added to be able to identify the nodes labeled with the same symbol. The above example defines *size* as the sum of the sizes of the subtrees plus two, due to the two terminal symbols.

One can also consider using attribute grammars. This way the bottom-up parameters in the derivation tree can be handled as synthesized attributes of the underlying attribute grammar. Such an extension of DTGP is discussed in details in [50].

4.5 Evolutionary operators

The evolutionary operators are based on the usual tree operators, however some modifications are needed, so 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*). RTG is also used for creating the initial population. These two operators have to be designed to follow the restrictions imposed by the context-free grammar. As it will be shown in this section, it is possible to construct algorithms for these operators in a way that does not increase the complexity of the evolutionary algorithm significantly.

4.5.1 Random tree generation

The random tree generator 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, select a rule randomly, 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.

The problem with this approach is that the trees can grow unbounded, which is an issue for evolutionary algorithms, where small random changes are preferred. To overcome this issue, a constant called min_p is assigned to each rule $p : A \to x$ to show the minimal size a subtree can have, if it is started by applying rule p. Thus, when selecting rules randomly during random tree generation, rules resulting in too large trees can be ignored. Hereby size can have various definitions depending on the restriction to be applied, as long as this metric is monotone, that is the size of a tree is never smaller than the size of a subtree. Calculating min_p takes some time, but it has to be done only once, before starting the evolutionary process, so the algorithm need not be optimal. The values can even be calculated manually and provided as constant for the algorithm.

To compute min_p, two functions are defined. Function $min_r(p, M)$ is the minimal size a tree can have if started with rule p and only the elements of M can be used as nonterminals in the tree with the exception of the root. Function $min_s(X, M)$ is the minimal size a tree can have if started with symbol X and only the elements of M can be used as nonterminals in the tree including the root. It is obvious that for any grammar $G = (\mathcal{N}, \Sigma, \mathcal{P}, S)$ and $p \in \mathcal{P}$

$$min_p = min_r(p : A \to x, \mathcal{N}) \tag{4.1}$$

To calculate min_r for a given rule, min_s is calculated for each symbol on the right-hand side of the rule. It gives enough information to calculate min_r. If the height of the tree is used, then the following computation rule applies:

$$min_r(p : A \to x, M) = 1 + \max_{X \in x} \{min_s(X, M)\} \tag{4.2}$$

To calculate min_s for a symbol X, this symbol has to be classified first. If X is a nonterminal symbol and it is an element of M, then min_r has to be calculated for each rule that can be applied to the symbol. However, when calculating min_r, symbol X need not be allowed anymore, as a consequence of the Bar-Hillel lemma (Lemma 3.18). This means min_r has to be calculated with nonterminal set $M \setminus \{X\}$. When min_r is known for each rule, the minimum will give the value of min_s for X. If X is a nonterminal symbol, but not an element of M, then min_s is ∞. If X is a terminal symbol, the result is a constant depending on what property is used as size. For height the calculation scheme is the following:

$$min_s(X, M) = \begin{cases} \min_{p:X \to u} \{min_r(p : X \to u, M \setminus \{X\})\} & X \in M \\ 0 & X \in \Sigma \\ \infty & \text{otherwise} \end{cases} \tag{4.3}$$

These equations compiled into a single function can be seen in Algorithm 4.2. Lines 4–13 calculate $min_s(X, M)$ according to Equation 4.3. This part might be moved into a separate function, but it is easier to analyze the function this way. Lines 14–15 store the calculated value if it is larger then the maximum stored so far. At the end the calculated maximum is increased by one and returned as result according to equation 4.2.

```
MinR(p : A → x, M)
 1  result := 0
 2  for each X in x
 3  do switch
 4        case X ∈ M :
 5             ms := ∞
 6             for each r : X → u
 7             do mr := MinR(r : X → u, M \ {X})
 8                if mr < ms
 9                   then ms := mr
10        case X ∈ Σ :
11             ms := 0
12        case default :
13             ms := ∞
14     if result < ms
15        then result := ms
16     return result + 1
```

Algorithm 4.2: Calculating min_r

The only recursive call occurs in line 7. It is easy to see, that each call decreases the size of M, because it removes X, which in turn is known to be in the set due to the condition in line 4. Thus, if the recursion does not end earlier, M will be reduced to $\emptyset$, making the condition in line 4 false, so no more recursive calls will be made. More details about calculating the min_p, min_r and min_s including theoretical foundations can be found in Section 3.3 of [50].

After min_p has been calculated for each rule, the random tree generator can be designed as shown in Algorithm 4.3. This algorithm randomly selects an applicable rule, adds new nodes for each symbol on the right-hand side, distributes the weight limits and continues derivations with the children.

```
RTG(root, limit)
 1  X := root.label
 2  if X ∈ Σ
 3     then return
 4  C := ∅
 5  for each p : X → α in P
 6  do if min[p] ≤ limit
 7        then C := C ∪ {p}
 8  p̂ : X → β := RuleSelect(C)
 9  childlimits := DistributeLimit(limit, β)
10  for each Y in β
11  do child := NewNode(Y)
12     RTG(child, childlimits[Y])
13     AddChild(root, child)
```

Algorithm 4.3: Random tree generation

The algorithm contains the following parts. Lines 2–3 ensure that no tree is created from terminal symbols. Lines 5–7 select applicable rules, that is rules having the root label on the left-hand side, and minimal size not above the required limit. Line 8 selects a rule randomly. Line 9 distributes the limit between the children based on the right-hand side of the rule. Lines 10–13 apply the rule by creating the appropriate node (line 11), creating a random derivation tree underneath the new node (line 12) and adding it to the current node (line 13).

The algorithm uses standard tree operations NewNode and AddChild to create and insert a new node, and two special function calls: RuleSelect and DistributeLimit. The first one is used to randomly select a rule from a set, and the second one is used to distribute the remaining limit between the subtrees. This depends on the actual type of the limit. For example if *limit* is an upper bound for height, then the limit for each subtree will be $limit - 1$. On the other hand, if it is an upper bound for the number of nodes, then it has to be distributed in a way such that $limit = 1 + \sum_{Y \in \beta} limit_Y$. Further details on these two calls are discussed in Section 5.3.

4.5.2 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. Usually not every node of the tree is a candidate for selection. For example, nodes close to the root are not selected as mutation points in order to keep the changes of the tree small. Leaves are never selected, since they are labeled with terminal symbols and there are no rewriting rules in the grammar with a terminal on the left-hand side.

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. Finding a given element in a search tree needs logarithmic time, thus one can try to design a similar algorithm for random node selection. This is similar to the node search, in a sense that a path from the root towards the leaves is traversed. However, this path is found randomly.

A naive algorithm would be to start at the root, and always choose the target of the next step between the node and its children randomly. Selecting the node itself means the search ends, and the given node is the selected node. When a child is selected, the search continues in the subtree under the child. The probability of selecting a child N' of N, that is making a step $N \rightsquigarrow N'$ is

$$P(N \rightsquigarrow N') = \frac{1}{(n+1)},$$

assumed that node N has n children. This means each subtree has the same probability to be selected, independently from its size. This is obviously not what we want, since nodes in smaller subtrees get higher probabilities assigned. However, this outbalanced search can be resolved by using weights according to the sizes of the subtrees rooted at the nodes. The size of a subtree is defined as usual, it is the number of the nodes in the subtree, which can also be given recursively. Namely, the size of a subtree is one plus the sum of the sizes of its subtrees, that is

$$N.size = 1 + \sum_{N' \text{is a child of } N} N'.size$$

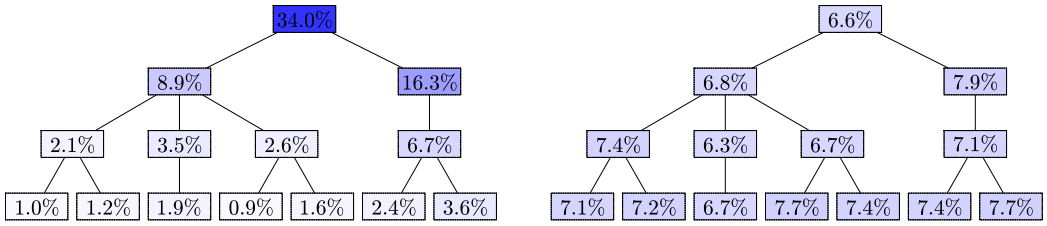
Using this measurement, the correct probability of making a step $N \rightsquigarrow N'$ is defined as follows:

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

With this definition, the probability of selecting a given node N_k from the root T through a path $T = N_0 \rightsquigarrow N_1 \rightsquigarrow N_2 \rightsquigarrow \ldots \rightsquigarrow N_{k-1} \rightsquigarrow N_k \rightsquigarrow N_k$ is the following:

$$\begin{aligned} P(T \rightsquigarrow^+ N_k) &= P(T \rightsquigarrow N_1) \cdot P(N_1 \rightsquigarrow N_2) \cdot \ldots \cdot P(N_{k-1} \rightsquigarrow N_k) \cdot P(N_k \rightsquigarrow N_k) \\ &= \frac{N_1.size}{T.size} \cdot \frac{N_2.size}{N_1.size} \cdot \ldots \cdot \frac{N_k.size}{N_{k-1}.size} \cdot \frac{1}{N_k.size} = \frac{1}{T.size} \ . \end{aligned}$$

Thus each node in the tree has the same probability to be selected. To compare the naive method and the weighted node selection, a simple tree was constructed and 1000 independent selections were carried out. The selection frequency is displayed in Figure 4.8 as percentage and also represented by the colors of the nodes.



(a) naive selection (b) weighted selection

Figure 4.8: Selection frequencies using different random node selection methods

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 $N.size$ depends only on $N'.size$, where N' is a child of N, $N.size$ 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.

Often, however, not every node is a possible candidate. In case of derivation trees, for example, no mutation or crossover may happen at the leaves. When the selection points are restricted, the number of these nodes should be used for determining the selection probability instead of the size. That is if $N.sw$ is the number of the candidate nodes in the subtree of N, and $N.nw$ is 1 when N is a candidate, otherwise 0, then

$$P(N \rightsquigarrow N') = \begin{cases} \frac{N.nw}{N.sw} & \text{if } N' = N, \\ \frac{N'.sw}{N.sw} & \text{if } N' \text{ is a child of } N. \end{cases}$$

It is easy to see that in this case the probability of a node to be selected is either 0 or $1/T.sw$, that is equally distributed on the set of candidates in the tree. Furthermore, if $N.nw$ is a constant or a bottom-up parameter, then $N.sw = N.nw + \sum N'.sw$ can also be stored as parameter, just like $N.nw$. This concept can be generalized even further when condition $N.nw$ is not a binary value, but a nonnegative real number. This means that nodes might have different probabilities to be selected, without changing the computation schema. The node selection function using the node weight property is shown in Algorithm 4.4.

```
SelectNode(node : N)
1  pockets := ()
2  pockets.add(N.node_weight)
3  for each N' in N.children
4  do pockets.add(N'.subtree_weight)
5     i := WeightedSelect(pockets)
6  if i = 0
7    then result := N
8    else result := SelectNode(N.children[i])
9  return result
```

Algorithm 4.4: Random node selection

First $x.nw$, also called *node weight*, is added to the list of pockets. Then for each child N' $N'.sw$, also known as *subtree weight* is added. Then a pocket is randomly selected with a weighted selection based on the pocket sizes. The result of this selection determines the next step.

Sometimes it is necessary to select a node with a specified label. To achieve this, the algorithm can be modified to ignore nodes with incorrect labels. However, the descents into the subtrees are random, thus it might happen that no such nonterminal is found. Furthermore, this search still uses the standard node and subtree weights, which might be different if we only consider a certain nonterminal. It is possible to calculate these weights separately for each nonterminal, but this increases the storage size needed for each node. In some cases, however, it makes sense to store this data, for example when the nonterminal alphabet is small. The updated node selection can be seen in Algorithm 4.5

```
SelectNode(node : N, nonterminal : A)
1  pockets := ()
2  pockets.add(N.node_weight[A])
3  for each N' in N.children
4  do pockets.add(N'.subtree_weight[A])
5       i := WeightedSelect(pockets)
6  if i = 0
7     then result := N
8     else result := SelectNode(N.children[i], A)
9  return result
```

Algorithm 4.5: Random node selection with multiple weights

This second algorithm can also be used if different operators require different node selection strategies. For example one might want to limit the size of derivation trees selected for mutation, but allow tree crossover at any point. This can be done by using a weight array as well.

4.5.3 Derivation tree mutation

Using random tree generation and random node selection as described in the previous sections, a simple mutation can easily be defined. First a node is selected randomly, then its subtree is replaced by a randomly generated subtree as demonstrated in Figure 4.9 and in Algorithm 4.6.

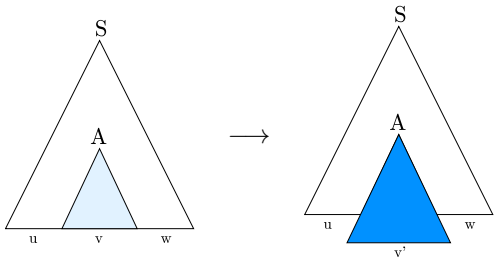


Figure 4.9: Simple mutation for derivation trees

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 height with another one having exactly the same height, however, in general no such restrictions are applied, other than the global limit on the new subtree size.

```
TreeMutation(node : T)
1  N := SelectNode(T)
2  N' := RandomTree(N.label)
3  ReplaceTree(N, N')
```

Algorithm 4.6: Simple mutation for derivation trees

For derivation trees, a two-point mutation can also be defined where instead of a subtree, a middle section of the tree is replaced. This mutation operator may simultaneously change two remote, but syntactically dependent parts of the word as shown in Figure 4.10.

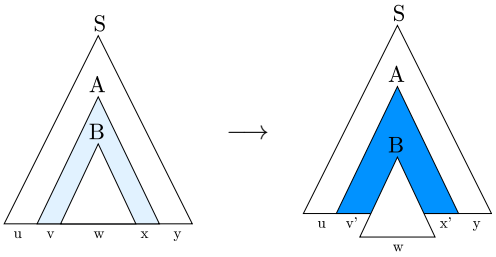


Figure 4.10: Two-point mutation for derivation trees

A special case of this mutation is when the first node is the root of the tree, as shown in Figure 4.11. This can be considered as a reversed one-point mutation: instead of changing the tree under a selected node deep down, change the tree above a node high up. The two-point mutation, including the special case can be very useful, as it allows the process to change parts of the solution that are decided early in the derivation process.

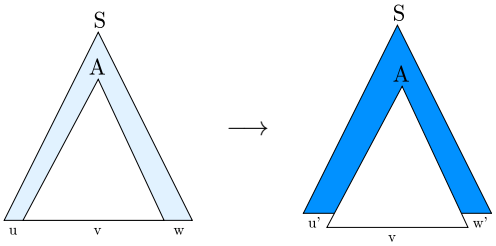


Figure 4.11: Reversed one-point mutation for derivation trees

During a two-point mutation, a middle section of the subtree is replaced. There are several ways to implement it. One possibility is to start as a normal mutation, but then insert part of the removed subtree back into the newly generated subtree. This process is shown in Algorithm 4.7. Note that this version of the mutation requires the special node selection of Algorithm 4.5.

It is important to note that these mutation operators introduce 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 4.4.2, it only requires logarithmic time.

```
TreeMutation2(node : T)
1  N := SelectNode(T)
2  M := SelectNode(N)
3  N' := RandomTree(N.label)
4  M' := SelectNode(N', M.label)
5  ReplaceTree(M', M)
6  ReplaceTree(N, N')
```

Algorithm 4.7: Two-point mutation for derivation trees

4.5.4 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 having the same nonterminal symbol as labels have to be selected and swapped as shown in Figure 4.12. One can also restrict the selection of crossover points based on the parameters stored in the nodes, just like in the case of mutation.

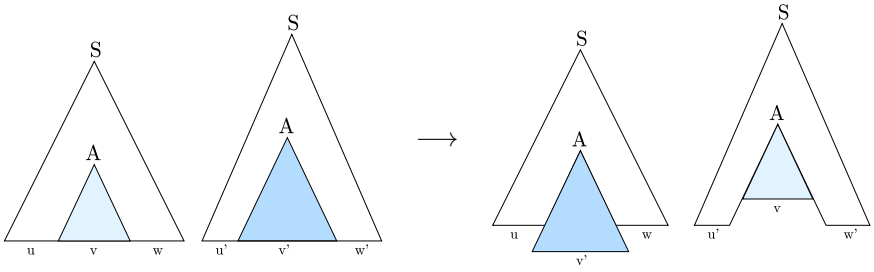


Figure 4.12: One-point crossover for derivation trees

Similarly to the multi-point mutation, a multi-point crossover can also be defined. In this case, subtrees are swapped and then some parts of them are swapped back. A schema for two-point crossover can be seen in Figure 4.13. A reversed crossover can also be implemented as a special

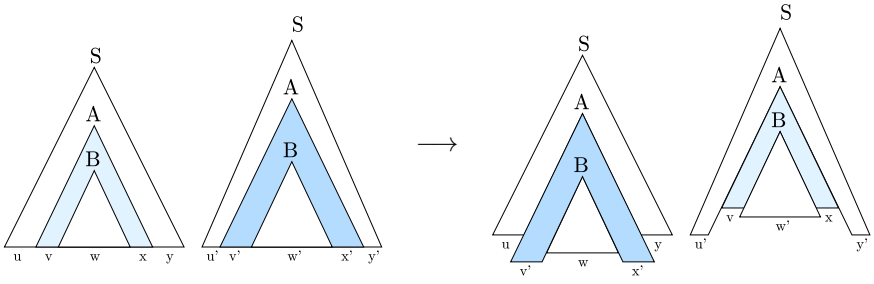


Figure 4.13: Two-point crossover for derivation trees

case of multi-point crossover, where the first node selected is the root node. Since it is irrelevant, whether we swap the subtrees or the crusts for a selected node, the reversed crossover might seem to be the same as the standard crossover. Note, however, that the change occurs at different places. With standard crossover, the change boundaries are close to each other, whereas using reversed crossover, the change boundaries are close to both ends of the string. Nevertheless, in this thesis neither standard crossover, nor the special crossovers will be used due to reasons discussed below.

Although crossover seems to be simple, there is one issue. The selected subtrees must have the same label at their root nodes. 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 Algorithm 4.5 to select another one in the second parent having the same label. This version of crossover is shown in Algorithm 4.8. However, it requires maintaining additional parameters as discussed before.

```
TreeCrossover(node : R_1, node : R_2)
1  N := SelectNode(R_1)
2  M := SelectNode(R_2, N.label)
3  SwapTree(N, M)
```

Algorithm 4.8: Crossover for derivation trees

Sometimes, however, it is not efficient to store the selection weights for all the nonterminals. They not only require more storage, but they also have to be maintained. 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.

4.5.5 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, as shown in Section 2.1.3. Based on this idea, a new operator called *pool crossover* is defined for derivation trees.

This operator works as follows. In the first step subtrees are selected, removed from the parents and inserted into a *contribution pool*. This means that only random node selection has to be applied. When nodes, that is subtrees are selected, the label or some parameter of the root node is checked, and based on this information, the selected subtrees are categorized. In the second step, these subtrees are inserted back into the parents, but in a random order. Insertion needs logarithmic time, because only the parameter updates have to be done. Selecting subtrees from the appropriate categories ensures the correctness of the operator. This modified crossover has the same effect as the original crossover, but it works globally and not locally. The outline of this operator can be seen in Figure 4.14 and the implementation is shown in Algorithm 4.9.

```
PoolCrossover(population : P)
 1  for  each T in P
 2  do N := SelectNode(T)
 3     L := N.label
 4     if pool_L ∉ pools
 5        then pools := pools ∪ {pool_L}
 6     pool_L := pool_L ∪ {N}
 7  for  each pool_i in pools
 8  do for  each N in pool_i
 9     do pool_i := pool_i \ {N}
10        if pool_i ≠ ∅
11           then M := RandomElement(pool[L])
12                SwapTree(N, M)
```

Algorithm 4.9: Pool crossover for derivation trees

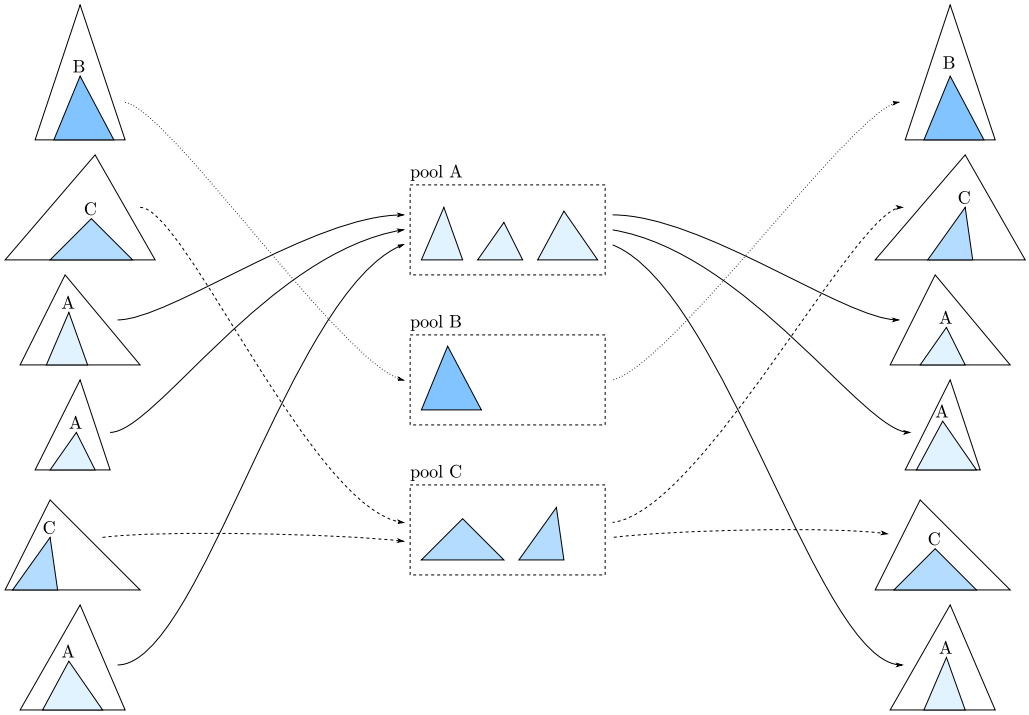


Figure 4.14: Pool crossover

As it can be seen in the figure, this operator is similar to crossover, but subtrees are swapped between multiple trees (A). It may happen, that from a certain type of subtree only one can be found in the contribution pool (B). In this case, this subtree is automatically inserted back to the tree from where it came. In some cases the outcome of the global crossover might be exactly the same as the outcome of the usual crossover (C).

Note, that the algorithm presented here does not actually remove subtrees, but marks the roots. It is similar to inserting references into a set. Furthermore, instead of selecting pairs of trees, one tree is selected, and then a random element of the pool is "swapped in". This means, the number of swaps is equal to the number of parents, although half as many would suffice. This algorithm has been chosen here for demonstration purposes due to its simplicity. For example it does not require special handling of pools with odd number of elements. Of course several other implementations of pool crossover are possible.

4.5.6 Operator costs

In this section we analyze the average case operator costs with respect to the size of the solutions. We assume that Equation 3.2 holds, that is during a derivation at each step at least one terminal symbol is added on average. This implies that if the solution, that is the word of the language has a size of n, then the size of the derivation tree is $\mathcal{O}(n)$.

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. Applying the operators also involves random node selection and parameter update. During mutation, random tree generation is applied as well. The operator costs for the average case are summarized in Table 4.2.

For most of the operators, RNS and parameter update is straightforward, the number of steps is bounded by the height of the tree, and at each node a constant amount of work is required. Thus, if the size of the tree is $\mathcal{O}(n)$, the extra work is $\mathcal{O}(\log n)$. The additional space required at each node is constant, only the selection weights have to be stored.

In case of *mutation*, the operator is not only a simple subtree replacement, but a new subtree has to be generated as well. The size of the new subtree is bounded by the mutation rate r, the work for each node is proportional to the number of rules, that is the size of $\mathcal{P}$.

Crossover can be done in several ways as discussed before, each of which has advantages and disadvantages. To apply the operator, nodes having the same labels have to be selected randomly. The standard RNS might find nodes with different labels, in this case the operator fails. For larger nonterminal alphabets the probability of the failure is higher. If the selection fails, one can retry until an appropriate node is found. In this case the time needed for RNS is increased proportionally to the size of the nonterminal alphabet. The third option is to store multiple selection weights and apply Algorithm 4.8, but it increases the number of parameters, as well as the update time proportionally to the size of the nonterminal alphabet.

Pool crossover avoids the issues of standard crossover; both selection and update remain independent of the size of the nonterminal alphabet. There is a slight chance that pool crossover fails for an individual. It happens when a pool only has a single element. However, usually the population size is much larger than the number of pools, that is the number of nonterminal symbols, therefore a failure is very rare.

operator	RNS	operator	update	success	node size
mutation	$\mathcal{O}(\log n)$	$\mathcal{O}(\lvert\mathcal{P}\rvert \cdot r)$	$\mathcal{O}(\log n)$	100%	$\mathcal{O}(1)$
crossover	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$1/\lvert\mathcal{N}\rvert^a$	$\mathcal{O}(1)$
crossover (retry)	$\mathcal{O}(\lvert\mathcal{N}\rvert \cdot \log n)^a$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	100%	$\mathcal{O}(1)$
crossover (weighted)	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\lvert\mathcal{N}\rvert \cdot \log n)$	100%	$\mathcal{O}(\lvert\mathcal{N}\rvert)$
pool crossover	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\approx 100\%^b$	$\mathcal{O}(1)$

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

Table 4.2: 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, and then the phenotype must be evaluated using a fitness function. The genotype-phenotype mapping is done by reading the frontier of the tree, whereas fitness evaluation is problem specific. However, 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, it must be mentioned in advance that for certain problems DTGP can use parameters to construct the phenotype or calculate the fitness function, thus reducing the evaluation costs to $\mathcal{O}(1)$. The details will be discussed in Section 5.1. Such an improvement is not possible for algorithms that do not store the derivations, such as linear GGGP methods.

4.6 DTGP example

To demonstrate how DTGP works and to examine some of its properties, an example is introduced and analyzed in this section. The test example is a Boolean regression problem, the same one that was used as an example for GA in Section 2.2.6.

The context-free grammar that generates the language of valid logical expressions for this example is $G = \{\mathcal{N}, \Sigma, \mathcal{P}, E\}$, where

$$\begin{aligned} \mathcal{N} &= \{E, T, F, P, N\} \\ \Sigma &= \{\vee, \wedge, (,)\} \cup \Sigma_P \cup \Sigma_N \\ \Sigma_P &= \{x_0 \mid x_1 \mid x_2 \mid x_3 \mid x_4\} \\ \Sigma_N &= \{\bar{x}_0 \mid \bar{x}_1 \mid \bar{x}_2 \mid \bar{x}_3 \mid \bar{x}_4\} \end{aligned}$$

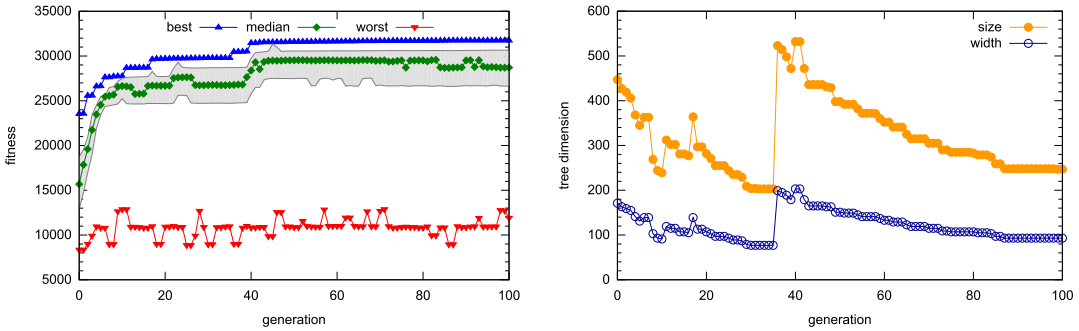
and the set of rules P is the following

$$\begin{aligned} E &\rightarrow E \vee T \mid T \\ T &\rightarrow T \wedge F \mid F \\ F &\rightarrow (E) \mid P \mid N \\ P &\rightarrow x_0 \mid x_1 \mid x_2 \mid x_3 \mid x_4 \\ N &\rightarrow \bar{x}_0 \mid \bar{x}_1 \mid \bar{x}_2 \mid \bar{x}_3 \mid \bar{x}_4 \end{aligned}$$

4.6.1 Single test results

One interesting question is how the algorithm performs during a single run. To test this, a population of 1000 individuals was generated and the DTGP algorithm was run for 100 steps using standard mutation and pool crossover. When the initial population was created, the height limit was set to 15. For random node selection, each node had the same weight, except, of course, the leaves. During mutation, the randomly generated subtrees had random height, independent of the replaced subtrees, but not larger than 10.

The fitness function was defined just as for the GA and GP examples. Each matching evaluation increases the fitness by 1000 and then the fitness is decreased by the size of the solution. For this example, the size is defined as the number of nodes in the tree. The best, worst and median fitness values are shown in Figure 4.15a. The fitness values for individuals above the lower 25% and below the top 25% are shown as a gray band around the median. The tree dimensions were also recorded, the size and the width of the best individual can be observed in Figure 4.15b.



(a) fitness values

(b) best individual dimensions

Figure 4.15: Results for a single run of the DTGP example

As it can be seen from these figures, the algorithm was able to find an expression that exactly describes the function by generation 40, and then it was able to further optimize its size from 532 down to 247. The result is comparable to the one reached by the GA algorithm when safe mutation is used, as shown in Figure 2.8. 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 is the 93 symbol long expression as follows.

$$\begin{aligned} f_{DTGP} &= ((\bar{x}_1 \vee (\bar{x}_3)))(x_3 \vee x_1\bar{x}_4 \vee \bar{x}_2\bar{x}_3\bar{x}_0) \\ &\wedge \ ((\bar{x}_1 \vee x_2)(x_3x_0 \vee \bar{x}_4\bar{x}_3 \vee \bar{x}_1\bar{x}_4)(\bar{x}_0 \vee x_1 \vee x_2\bar{x}_1) \vee (x_0)x_3x_4 \vee ((x_2)\bar{x}_3(x_3)) \vee x_4(x_1x_1)) \\ &\vee \ x_1(\bar{x}_2)x_3\bar{x}_2 \end{aligned}$$

The conjunction symbol ($\wedge$) has been omitted from the expression for better readability, but all the parentheses are shown as found by the algorithm. One can see that there are some unnecessary parentheses, but not as many as in the individuals found by the GA process. Furthermore, repeating parentheses, like $((x_0))$, were almost completely eliminated, because DTGP can handle parentheses as part of the syntax, whereas in GA they are handled specially to avoid invalid individuals due to the simple representation.

The top of the derivation tree representing the best solution found by the algorithm can be seen in Figure 4.16. Each subtree represents a Boolean function that is shown as a 32bit hexadecimal number at each nonterminal. Some parts of the tree can easily be identified in the expression, for example the rightmost subtree of the root node represents the end of the result $x_1(\bar{x}_2)x_3\bar{x}_2$.

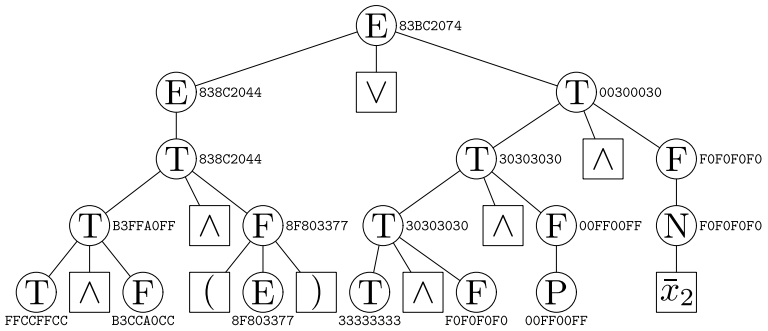
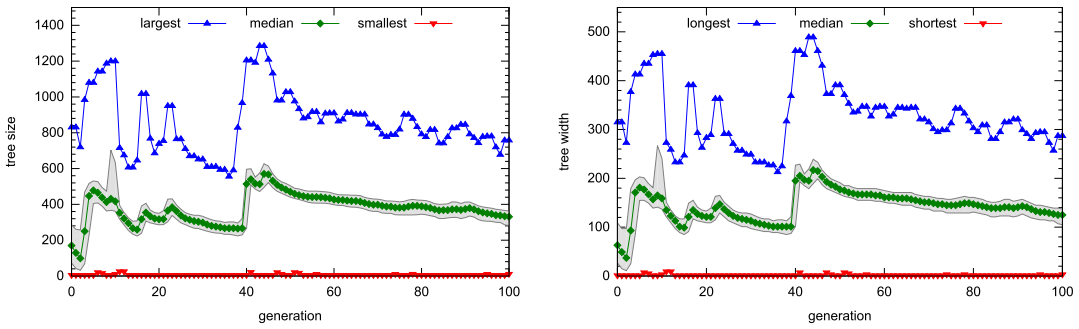


Figure 4.16: Top of the best individual

One disadvantage of GP or DTGP over GA is the size of the individuals. The solution can be found at the frontier of the tree as a string, thus the useful size of the individual is the width of the tree. However, a full derivation tree is stored, making the memory consumption proportional to the size of the tree. In the test run the size of the best tree reached 500, and a single node might require 20–100 times more storage than a character in a GA string. On the other hand, the important question is the asymptotic relation between the size and the width, which is expected to be linear, as described by Theorem 3.20. Figure 4.15b suggests that this theorem holds for the test grammar.

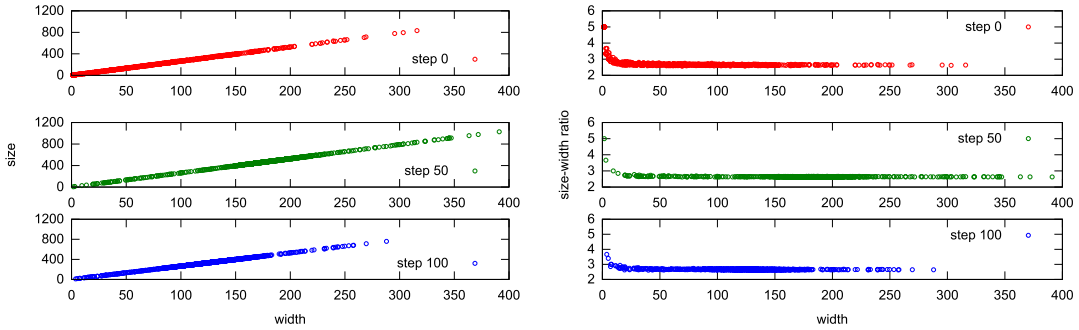
Nevertheless, the fact that the width and the size of the best individual are correlated does not necessarily mean that it is true for any tree. Therefore during the test run the tree dimensions were recorded. This data is visualized in Figure 4.17, showing the extremes, the median and the middle 50% for both size and width. This figure shows a correlation too.

To see not only statistical data regarding tree dimensions, but the exact information as well, all trees of three selected generations are plotted in two-dimensional charts with coordinates showing the size and width-size ratio with respect to width. Figure 4.18a shows the tree sizes for the first and the last generations and a generation at half time, while in Figure 4.18b the width-size ratio is plotted for the same generations.



(a) size (b) width

Figure 4.17: Tree dimensions during the DTGP test run



(a) absolute (b) relative

Figure 4.18: Tree size compared to tree width in selected steps

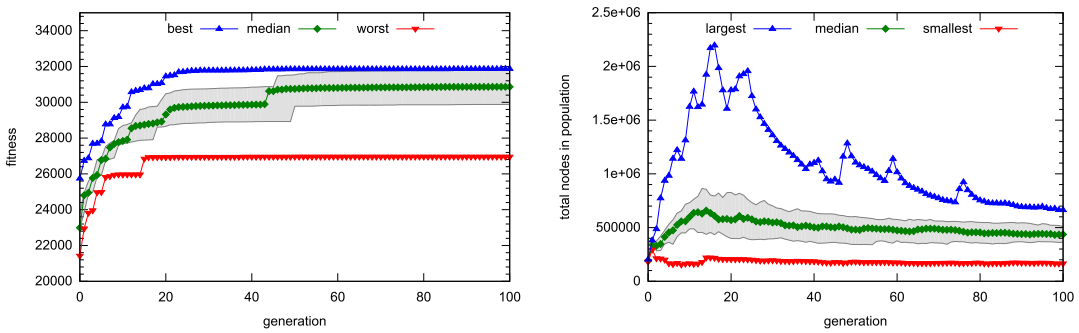
These figures show that considering every tree in the population, the width-size ratio is near constant, especially for larger trees. This also verifies that if the dimension of the problem n is considered to be proportional to the length of the words in the language, then the size of the DTGP individuals as well as the size of the population measured in node count is $\mathcal{O}(n)$.

This realization might be surprising at first, but if one considers simpler trees, the width-size ratio can be exactly calculated. For example, it is a well-known fact that a balanced binary tree with $n = 2^k$ leaves has a height of k and the total number of nodes is $2^{k+1} - 1 = 2n - 1 = \mathcal{O}(n)$. It is possible to define grammars in a way that this assumption does not hold, therefore in examples introduced later in this thesis the correlation between size and with will be checked again.

4.6.2 Independent tests

Analyzing a single run is very useful for determining some properties, but since DTGP, just like any other EA, is a randomized algorithm, each run can yield different results. Therefore 100 independent runs were carried out and the results are shown in Figure 4.19. At each step, the best individual was selected from each population, and statistical data of these selected individuals is plotted in Figure 4.19a. Also, the population size was measured by counting the total number of nodes. This information is shown in Figure 4.19b.

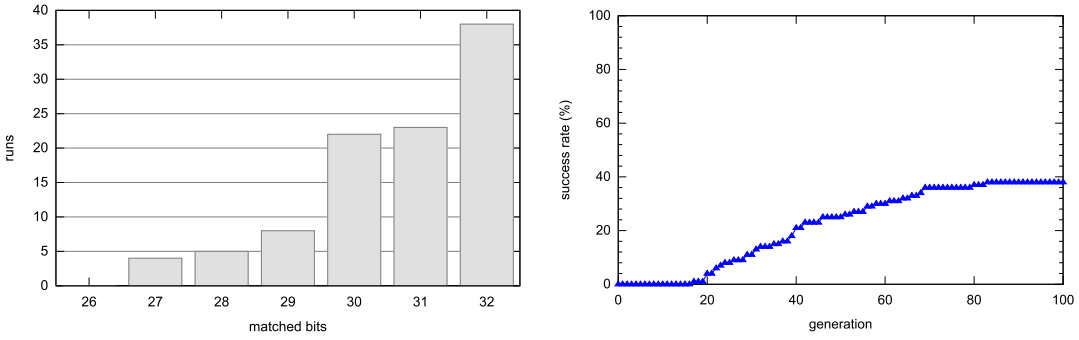
The best runs were able to reach fitness values above 31000, that is all 32 bits were matched, meaning the run was successful. This is also true for the top 25%. However, the median is below 31000, and in the worst case the fitness was slightly below 27000 meaning only a 27 bit match.



(a) fitness value (b) population size

Figure 4.19: DTGP example results accumulated for 100 independent runs

To see what the frequencies of the various end results are, the number of matches was checked for each run in the final step, and the histogram of this data is plotted in Figure 4.20a. It can be seen that from the total 100 runs 38 were able to find a 32-bit match, but 23 found only 31 bits, and 22 found only 30 bits. The worst result was 27 bits, that happened 4 times. Successful runs find total match in different generations. The number of runs that found all 32 bits by a certain generation, that is the *success rate* can be seen in Figure 4.20b.



(a) found matches (b) success rate

Figure 4.20: DTGP example final results for 100 independent runs

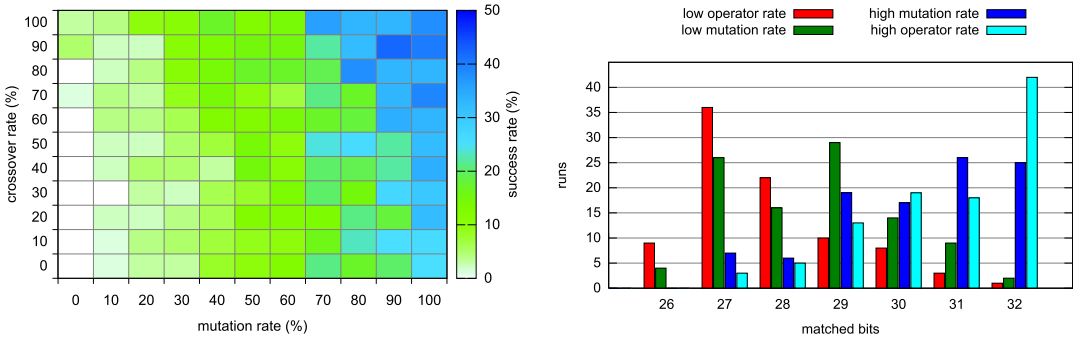
4.6.3 Parameter settings

A DTGP algorithm has many parameters that can be set. It is interesting to see how these parameter settings influence the outcome. The most important parameters are the population size, the operator application rates and the number of steps. For a DTGP algorithm, one can also set node selection parameters and bounds for random tree generation.

Operator application rate

It is common in GP to use crossover on the whole parent pool to create a new population, and then apply mutation to each individual. However, these settings can be changed. To examine the results, the following test was done. Crossover and mutation rates were set to $0\%, 10\%, 20\%, \ldots, 100\%$, and 100 independent runs were carried out to measure the success rate. The results are shown in Figure 4.21a. It can be seen that in order to achieve a higher success

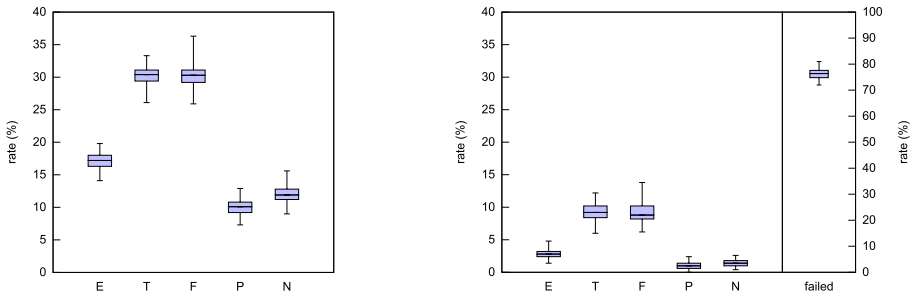
rate for this particular problem, a high mutation rate is necessary. This corresponds to the results in [50]. It is also useful, although not necessary to have a high crossover rate. The histogram of matched bits is also plotted in Figure 4.21b for four of the tests, with 10% and 90% rates for both crossover and mutation. It shows that even though a high crossover rate does not improve success rate by much, it still improves the results significantly. On the other hand, high mutation rate alone can mean not only overall improvement but also an increased success rate.



(a) success rate (b) match histograms

Figure 4.21: Results of various operator settings

Setting crossover probability in this test means that a certain percentage of the new population is created using pool crossover. However, even if it is set to 100%, in some marginal cases pool crossover might fail, meaning that the result is an exact copy of the parent. As mentioned in Section 4.5.6, this happens when a pool contains only a single subtree. The probability of this situation to occur is a complex function of the average tree size, the population size and the frequency of the given nonterminals. Therefore, during a randomly selected Boolean regression test, the pool sizes were recorded at each step for each nonterminal. The pool size shows how many individuals were crossed at a given nonterminal. Statistical information is plotted in Figure 4.22a showing minimum and maximum pool sizes as well as the median and the middle 50%. For classical two-parent crossover similar information can be collected by checking how many trees have been created by applying crossover at a node labeled with the given nonterminal. This information is plotted in Figure 4.22b with an additional column showing the number of failed crossovers.



(a) pool crossover (b) two-parent crossover

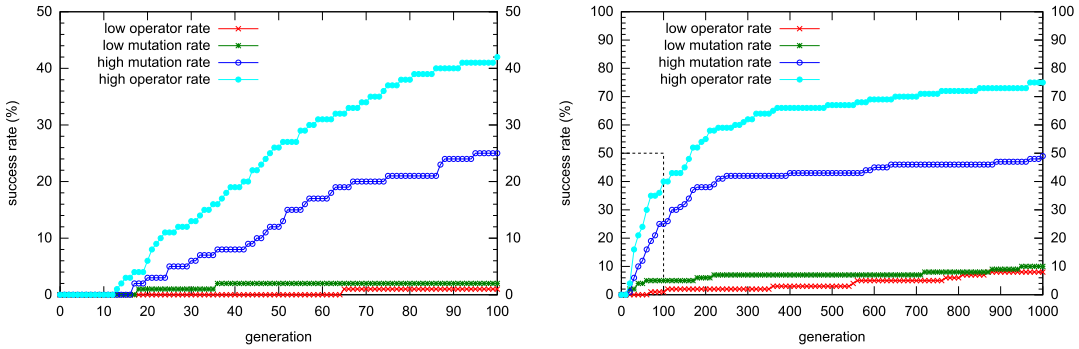
Figure 4.22: Application rates for the crossover operators

The two figures were derived from the same data, that is using the same sequences of node selections. Since both types of crossovers use a single node selection for each parent individual, and the pool sizes as well as the two-parent crossover types depend only on the label of the selected nodes, these results are directly comparable.

Number of steps

In the above mentioned tests, the algorithm was stopped after 100 steps. However, if the process is run longer, it might improve the results. Usually during evolutionary optimization the improvement rate decreases over time, as it can be observed in the previous figures displaying fitness value or success rate over generations. One of the reason for the decreased improvement rate is the fact that the diversity of the population also decreases.

To see how success rate changes through generations with various operator rates, tests were run using the previously mentioned four setups with 10% and 90% operator rates. First only 100 steps were done, then the algorithm was tested with 1000 steps as well. The results are plotted in Figure 4.23.



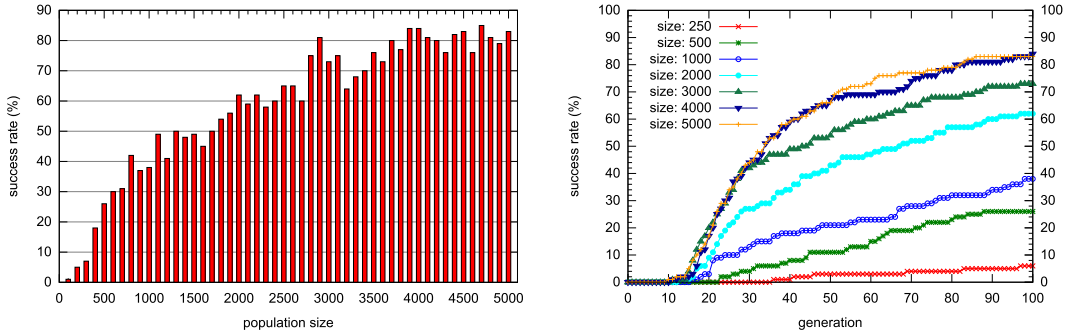
(a) short run (b) long run

Figure 4.23: Success rate development for various operator settings

As the charts show, the success rate steadily increases at first, but after 200 steps it slows down. The success rate for the best setup is around 40% after 100 steps, increases to approximately 55% after 200 steps, but then in the remaining 800 steps the success rate reaches only 75%. Therefore, it is better to run 10 independent tests with 100 steps, rather than one with 1000 steps. With 40% success rate, the probability that one of the 10 independent runs is successful is 99.4%.

Population size

The population size also has an influence on the success rate as shown in Figure 4.24, where the success rate is plotted using 100% operator rate with various population sizes.

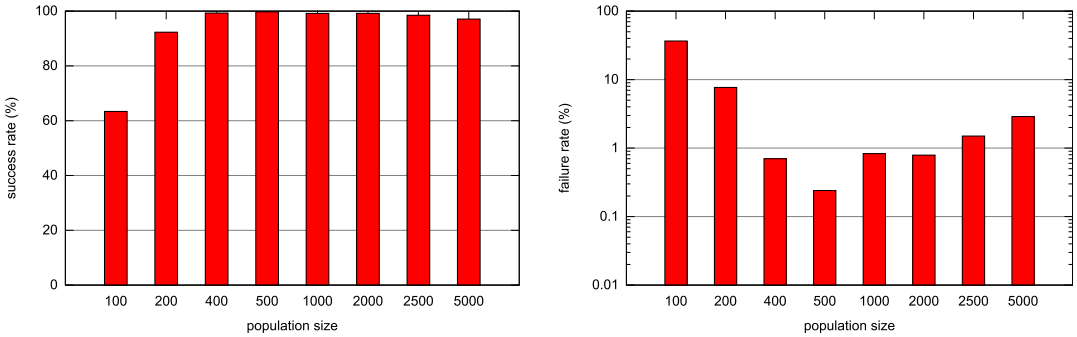


(a) final (b) over generations

Figure 4.24: Success rate development for various population sizes

One can see that population size should be above 300 to have a success rate above 10%. With increasing population size the success rate also increases, for example by doubling the size from 1000 to 2000, the success rate changes from below 40% to above 60%. Around population size 4000 the success rate reaches its maximum at around 80%.

However, it is difficult to compare these results, because higher population means higher memory consumption, and also higher processing time. Using the assumption that the time needed for making 100 steps is proportional to the population size, the success rate can be normalized. For example, using a population size of 10000 as a standard, normalizing the result for a population size of 1000 means considering 10 independent runs, while for size 2000 only 5 independent runs are considered. The normalized success and failure rates are shown in Figure 4.25.



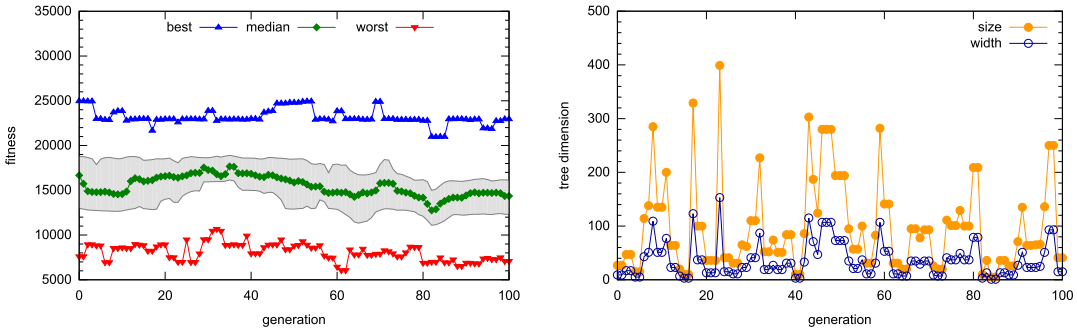
(a) success rate (b) failure rate

Figure 4.25: Normalized rates for various population sizes

The results show that the optimum is around population size 500, which only means 26% success rate, but running it twice gives success rate above 45% success rate. A single run of a test with a population of 1000, on the other hand, gives only 38% success rate, and needs approximately double as much memory and time.

4.6.4 Search bias

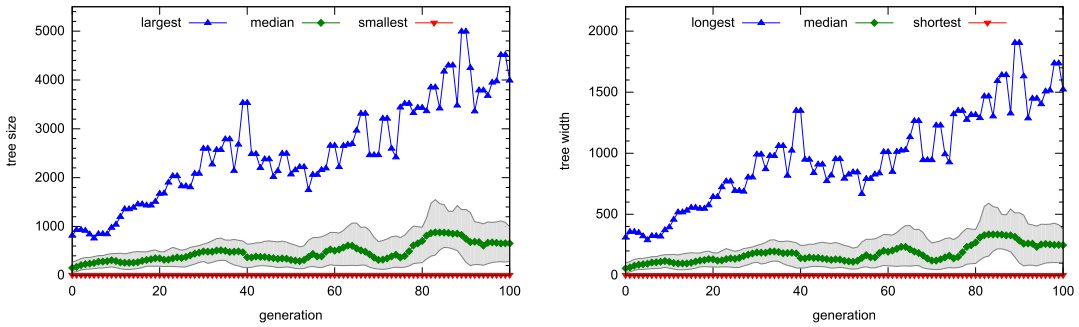
In an evolutionary process the main driving force is the fitness function. However, DTGP uses a complex representation, therefore it is interesting to check if the framework introduces any bias. To test this, the evolutionary process was run using the previously mentioned setup, but the selection was randomized. The results can be seen in Figure 4.26.



(a) fitness values (b) best individual dimensions

Figure 4.26: Results of a randomized run

Without taking the fitness value into consideration, the process creates individuals that have fitness values between 5000 and 25000 with the median being around 15000, and there seems to be no significant change over time. It is also interesting to look at the tree dimensions during the process. They are shown in Figure 4.27.



(a) size (b) width

Figure 4.27: Tree dimensions during a randomized run

It can be observed that the dimensions of the largest tree keep increasing, and it is also true for the individuals in the largest part of the population. This results show that the method itself introduces a slight bias by creating larger trees. This is a known phenomenon, called *bloat* [5]. If needed, this can be addressed by carefully selecting random node selection and mutation parameters, or by using some known methods for avoiding bloat [23, 38]. On the other hand, if the fitness function prefers smaller individuals, just like in the example shown in this section, the issue can simply be ignored.

4.7 Summary

In this chapter a new optimization method called *derivation tree based genetic programming* (DTGP) has been introduced. It is a tree based grammar guided genetic programming (GGGP) method, which provides several improvements over other algorithms of its class. These improvements are the following:

Bounded random tree generator is applied to limit the size of the generated trees.

Parameters are stored in each node to fine-tune the operators.

Logarithmic node selection is implemented using appropriately selected parameters representing selection weights.

Pool crossover is introduced replacing common crossover to increase the crossover success rate up to practically 100% without increasing computational complexity.

The result is a generic algorithm that can be used for various optimization problems, where the set of valid solutions can be described by a context-free grammar. The size of the individuals is comparable to other GP methods, and most of the operators run in logarithmic time. In Chapter 5 this algorithm will be further improved, and in Chapter 6 some example applications will be presented.

5 Improving DTGP

In the previous chapter, the details of derivation tree based genetic programming were described. As we have seen, the tree structure used by the algorithm 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. Also, the complex operation of creating random trees can also be adjusted.

This chapter presents some of the possible improvements. These can be divided into three categories: harnessing the potentials in parameters, implementing semantic constraints and improving randomization during tree generation.

5.1 Parameter utilization

In Section 4.4.2 it has been shown how various tree properties can be stored as parameters at the nodes of the derivation trees. In Section 4.5.2 some of these parameters were used to implement a balanced random node selection. There are, however, further uses of parameters, some of which are presented in this section.

5.1.1 Run-time frontier recovery

One issue with DTGP is that hypotheses are stored at the frontiers of the derivation trees, and this frontier has to be read before each fitness evaluation. Usually this means traversing the whole tree, requiring a lot of extra time.

However, it is possible to store the frontier of a tree as a property called fr in its root node. That is for tree $T = N[T_1, T_2, \ldots, T_n]$

$$T.fr = T_1.fr \cdot T_2.fr \cdot \ldots \cdot T_n.fr$$

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. Unfortunately, this is a parameter that cannot be stored in constant space. The space needed to store the frontier once is $\mathcal{O}(n)$, thus the initial estimate for the extra space is $\mathcal{O}(n^2)$. Fortunately, a better upper bound can also be given.

Theorem 5.1
Given a derivation tree with n leaves. Storing the subtree frontier in each subtree root requires $\mathcal{O}(n \log n)$ space.

Proof 5.2
Let us consider an arbitrary derivation tree T having root node N and n leaves containing the frontier u. For each node M in tree T there is exactly one path from the root N to node M. The *distance* of the two nodes is the length of this path and it is denoted by $|N, M|$. The distance can be used to define *levels* in the tree as sets of nodes as follows

$$\forall i \in \mathbb{N}\ \mathcal{L}_T(i) = \{M \mid M \in T \wedge |M, T| = i\}$$

If the height of tree T is h, then $\forall i > h\ \mathcal{L}_T(i) = \emptyset$.

Given $M_1, M_2 \in \mathcal{L}_T(i)$ such that $M_1 \neq M_2$, and subtrees T_1, T_2 rooted in M_1 and M_2 respectively. The distance between the nodes in a subtree and the root of the tree is always larger than the distance between the root of the subtree and the root of the tree. Therefore

$$M_1 \neq M_2 \iff M_1 \notin T_2 \wedge M_2 \notin T_1$$

Thus any leaf L_k representing symbol u_k of frontier u cannot be in both subtrees T_1 and T_2. Therefore $M_1.fr \cap M_2.fr = \emptyset$, meaning

$$\sum_{M \in \mathcal{L}_T(i)} |M.fr| \leq |u| = \mathcal{O}(n).$$

The total space for storing frontier information is

$$\sum_i \sum_{M \in \mathcal{L}_T(i)} |M.fr| = \sum_{i \leq h} \sum_{M \in \mathcal{L}_T(i)} |M.fr| \leq h|u| = \mathcal{O}(n \log n).$$

□

This means, using a total of $\mathcal{O}(n \log n)$ extra space the frontier of the tree will be available in constant time during fitness evaluation. Note, however, that to keep the parameter update logarithmic, it is assumed that updating the parameters at a single node requires constant time. Unfortunately, it is not the case with the usual string implementations, since concatenating two strings requires moving at least one of them to a different location in the memory. Furthermore, even if the frontier can be recovered in constant time as a string, processing this string is usually linear, therefore this approach has only a limited use.

5.1.2 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. For this purpose, one might take advantage of the derivation tree that provides a syntactical description. If an evaluation property of 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. The extra storage needed for this parameter depends on the exact problem, but in certain cases it might be enough to use constant extra space per node. If it is possible, then storing this property is a better option than storing the frontier, as it does not require extra space and the update algorithm can remain logarithmic.

One example is the domain of n-ary Boolean functions, which can be represented as 2^n bit integers, just like the example used in Section 4.6. Thus the function represented by a subtree can be stored as an integer in the root node of the subtree. An example for the calculation schema for binary functions is shown in Example 5.3. In fact, this schema was also used for the above mentioned DTGP example. Recall Figure 4.16, where the functions represented by each subtree are shown as 32-bit hexadecimal integers. These integers were actually calculated and stored as parameters to speed up the fitness evaluation.

Example 5.3 *(Run-time hypothesis evaluation)*
The set of representations is the set of binary Boolean expressions constructed of conjunction, disjunction, parenthesis, two variables and two negated variables. These are evaluated to construct the Boolean functions, which in turn can be represented as 4 bit integers. These integers will be stored as parameters denoted by p in each node. Using integer operators *binary or* ($|$) and *binary and* ($\&$), the problem can be described by grammar

$$G = (\{E, T, F, V\}, \{\vee, \wedge, (,), x_0, x_1, \bar{x}_0, \bar{x}_1\}, \mathcal{P}, E)$$

where the set of rules $\mathcal{P}$ is defined as follows

$$\begin{aligned}
E \to T \quad & E.p := T.p \\
E_0 \to E_1 \vee T \quad & E_0.p := E_1.p | T.p \\
T \to F \quad & T.p := F.p \\
T_0 \to T \wedge F \quad & T_0.p := T_1.p \& F.p \\
F \to (E) \quad & F.p := E.p \\
F \to V \quad & F.p := V.p \\
V \to x_0 \quad & V.p := \mathtt{0101}_2 \\
V \to x_1 \quad & V.p := \mathtt{0011}_2 \\
V \to \bar{x}_0 \quad & V.p := \mathtt{1010}_2 \\
V \to \bar{x}_1 \quad & V.p := \mathtt{1100}_2
\end{aligned}$$

A derivation tree for expression $x_0 \vee \bar{x}_0 \vee x_0 \wedge x_1 \wedge \bar{x}_1$ can be seen in Figure 5.1. Property p is also shown, which represents the function described by the subtrees. For example the root node for expression $x_0 \wedge x_1$ contains `0001`, meaning the value of the represented function is 0 for $x_0 = x_1 = 0$, for $x_0 = 1, x_1 = 0$ and for $x_0 = 0, x_1 = 1$, and it is 1 for $x_0 = x_1 = 1$.

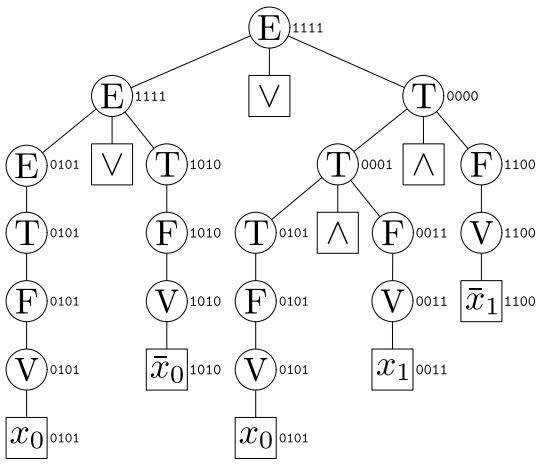


Figure 5.1: Example tree for phenotype mapping

5.1.3 Run-time fitness calculation

In certain cases it is also possible to partially or even completely calculate and store the fitness value and not just the evaluation property. For example if a Boolean expression is searched that describes a pre-defined function, just like the Boolean regression problem used in previous examples, the difference of the described function and the target function can be stored as a parameter. Another application is calculating the cost associated with the hypothesis, as shown by the following example.

Example 5.4 *(Run-time partial fitness calculation)*
The goal is to find a circuit composed of shifters, adders and multipliers. All these components have different costs: 1, 2 and 5 respectively. When the fitness value is calculated, the cost has a negative effect, that is individuals with higher costs have lower fitness values. If the circuits are not allowed to have loops, then they can be described as functions, thus they can be generated by a grammar. Furthermore, the cost can be calculated as a bottom-up parameter as shown below.

$$\begin{aligned}
C_0 &\to shift(C_1, N) & C_0.cost &= 1 + C_1.cost \\
C_0 &\to add(C_1, C_2) & C_0.cost &= 2 + C_1.cost + C_2.cost \\
C_0 &\to mult(C_1, C_2) & C_0.cost &= 5 + C_1.cost + C_2.cost \\
C &\to x & C.cost &= 0 \\
C &\to N & C.cost &= 0 \\
N &\to 1|2|3|4
\end{aligned}$$

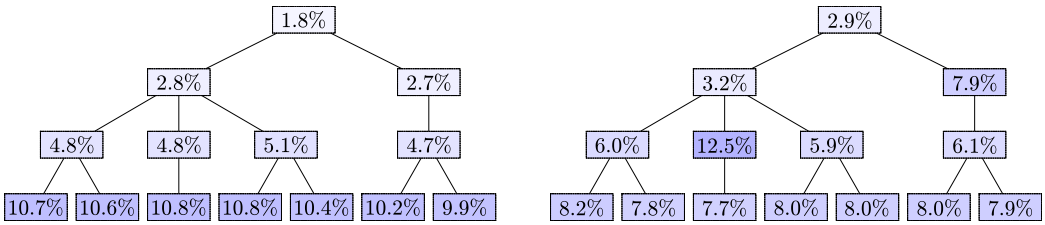
When calculating the fitness value, the cost has to be calculated, which normally means traversing the tree or the frontier and counting the various circuit components in $\mathcal{O}(n)$ time. However, by defining and maintaining the appropriate parameter, the cost information can be accessed in $\mathcal{O}(1)$ time, with the usual $\mathcal{O}(\log n)$ update time required by the evolutionary operators.

5.1.4 Operator biasing

As it was shown in Section 4.5.2, any selection weight $c(x)$ that can be defined as bottom-up property can be stored as parameter and used for biasing the random node selection. A straightforward way is to use standard tree properties to define selection weight. Some examples are shown below. Note that selection weight for leaves is always set to 0.

node selection weight	subtrees preferred during node selection
1 if $height < 15$, 0 otherwise	subtrees not higher than 15
$1 - breadth/height$	unbalanced trees that are tall, but narrow
$1/(height^2)$	small trees with exponentially increasing weight

The last two selection weights were used for 1000 random selections on the previously used test tree. The results can be seen in Figure 5.2.



(a) $X.nw = 1 - breadth/height$ (b) $X.nw = 1/(height^2)$

Figure 5.2: Selection frequency using different random node selection methods

More complex selection weights can also be defined. As Example 5.4 shows, for certain problems a component of the fitness can be calculated as a bottom-up parameter. This component can also be used for defining node selection weight, thus certain building blocks can be protected. Using node selection weight $N.cw = N.nw \cdot N.cost$, the algorithm will try to replace high-cost subtrees.

Sometimes a problem specific definition can be given for building blocks that should be protected, or the opposite, for those that should be selected with a higher probability. In the Boolean regression problem one can observe situations where the function represented by a tree is exactly the same as the one represented by one of its subtrees. For example, in Figure 5.1 the tree and its left subtree both calculate function `1111`. To try to avoid these, the node selection function can be defined as follows:

$$N.bw = \begin{cases} N.nw + 1, & \text{if } N.function = N.parent.function \\ N.nw & \text{otherwise} \end{cases}$$

Unfortunately, this definition does not satisfy the bottom-up condition, as it depends on a property of the parent node. Nevertheless, the calculation scheme can be changed so that whenever the function parameter is calculated for a node, the node weight is updated for the children having the same function. Another option is to increase the subtree selection weight $T.sw$ every time $T.function = N.function$ for a child N of node T. Although this is not the same, the tree itself will have a higher selection probability.

5.2 Semantic constraints

As mentioned earlier, the bottom-up parameters used in DTGP can be considered as synthesized attributes of an underlying attribute grammar. In DTGP, these attributes are mainly used to describe various properties of the derivation trees, such as size or height, or to bias the evolutionary operators with node selection weight, but as shown in the previous section, synthesized attributes can also contain semantic information.

Following the nomenclature of attribute grammars, in this section bottom-up properties or parameters will 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. In Example 5.3, attribute p always contains the function represented by the subtree under a node, but its calculation scheme differs for every rule. As defined in Section 4.4.2 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 in addition to the context-free grammar that defines the syntax and the rules that define the attributes, some additional semantic information is predefined that has to be adhered during the random tree generation process. That means the information has to be passed down to the subtrees, and might have 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 5.5 *(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 pre-defined value.

Example 5.6 *(Distribution set for integer attributes)*
Let us consider an attribute h representing *height* defined as usually

$$T.h = 1 + \max_{1 \le i \le n} \{T_i.h\}$$

In this case $D_h(4,2)$ is the following set

$$D_h(4,2) = \{(0,3),(1,3),(2,3),(3,3),(3,2),(3,1),(3,0)\}$$

This set is illustrated in Figure 5.3a over the space of all vectors with $T_i.h \le 5$ ($i \in \{1,2\}$). The values of $T.h$ are represented by colors.

Example 5.7 *(Distribution set for set-type attributes)*
Let us consider an attribute s showing the set of terminal symbols that can be found on the frontier of the tree. This is a synthesized attribute defined by the following function

$$T.s = \bigcup_{i=1}^{n} T_i.s$$

Distribution set $D_s(\{a,c\},2)$ has 9 elements as follows

$$\begin{aligned} D_s(\{a,c\},2) \quad = \quad & \{(\emptyset,\{a,c\}), \\ & (\{a\},\{c\}),(\{a\},\{a,c\}), \\ & (\{c\},\{a\}),(\{c\},\{a,c\}), \\ & (\{a,c\},\emptyset),(\{a,c\},\{a\}),(\{a,c\},\{c\}),(\{a,c\},\{a,c\})\} \end{aligned}$$

Subsets of finite, ordered sets can be represented as binary vectors. For example given an alphabet Σ with four symbols $\{a,b,c,d\}$ subset $\{b,c\}$ can be written as `0110`, and subset $\{a,c\}$ can be written as `1010`. Using the *binary or* operator (denoted by $|$), $T.s$ can be calculated as follows

$$T.s = T_1.s | T_2.s | \ldots | T_n.s$$

Using this notation, distribution set $D_s(\texttt{1010},2)$ is

$$\begin{aligned} D_s(\texttt{1010},2) \quad = \quad & \{(\texttt{0000},\texttt{1010}), \\ & (\texttt{1000},\texttt{0010}),(\texttt{1000},\texttt{1010}), \\ & (\texttt{0010},\texttt{1000}),(\texttt{0010},\texttt{1010}), \\ & (\texttt{1010},\texttt{0000}),(\texttt{1010},\texttt{1000}),(\texttt{1010},\texttt{0010}),(\texttt{1010},\texttt{1010})\} \end{aligned}$$

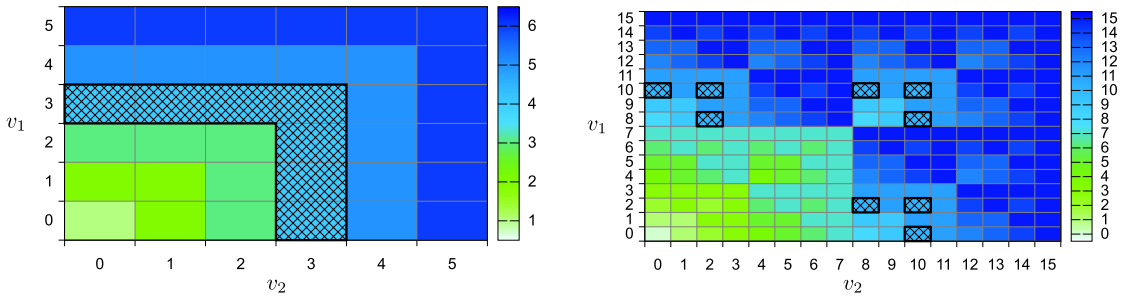
Binary vectors of length n can also be interpreted as n-bit integers. Using this interpretation, $D_s(\{a,c\},2)$ is shown in Figure 5.3b.

Definition 5.8 *(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 $D_a(v_0,n)$.

Attributes can be defined separately for each rule. Thus, there can be different distribution sets and distribution functions for each rule as well. Furthermore, if a given rule has n' nonterminal symbols on its right-hand side, only distribution sets and distribution functions with $n = n'$ are of interest. Therefore, for a given rule p and attribute a, parameter n is omitted, and the distribution sets are denoted by $D_{a,p}(v_0)$ whereas the distribution function is denoted by $\hat{f}_{a,p}$.



(a) $D_h(4,2)$ (b) $D_s(\{a,c\},2)$

Figure 5.3: Distribution sets

Remark 5.9 *(Distribution set for terminal functions)*
There are rules that do not have nonterminal symbols on the right-hand side. Such functions define the attributes as constants, that is

$$T.a = c_p$$

The definition of the distribution set can be applied to these as well, yielding the following:

$$D_{a,p}(v_0) = D_a(v_0, 0) = \{() \in (V_a)^0 \mid c_p = v_0\} = \begin{cases} \{()\} & \text{if } v_0 = c_p \\ \emptyset & \text{otherwise} \end{cases}$$

Since the value v_0 does not have to be distributed, $\hat{f}_{a,p}$ will be denoted by $\varnothing$.

With the help of distribution sets and distribution functions, it is possible to define attributes that put semantic restrictions on the derivation trees.

Definition 5.10 *(Forced synthesized attribute)*
A synthesized attribute a is called forced synthesized attribute, if

(i) For each $A \in \mathcal{N}$ $A.a$ is defined,

(ii) A required value $v_0 \in V_a$ for a of the root-node is given, and

(iii) For each rule $p : A \to \beta \in \mathcal{P}$, with $\beta \notin \Sigma^*$ there is a distribution function $\hat{f}_{a,p}(v)$, which is defined over $V' \subseteq V_a$.

Remark 5.11
Unlike synthesized attributes, forced synthesized attributes are not calculated based on subtree attributes. Instead, their values are distributed from parent to children nodes using the distribution functions. One can define a normal synthesized attribute with the same rules. In this case, the value of the forced and the normal synthesized attribute will be the same, even though they are calculated differently. This fact is a direct consequence of Definition 5.5.

Remark 5.12
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. Finding a value for a forced synthesized attribute involves finding the set of possible values and selecting one randomly, that is taking the value from the parent and *distributing* it among the children. In the special case when $|D_{a,p}(v_0)| \leq 1$ for each rule and each attribute value, the distribution function is deterministic, thus the forced synthesized attribute is also an inherited attribute.

Definition 5.13 *(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.

The simplest distribution function is the one that randomly selects an element of $D_{a,p}(v_0)$ with a uniform distribution. However, it is also possible to define a function that assigns lower or even 0 probability to certain elements. This way an additional semantic constraint or search bias can be introduced. If a modified distribution function is used such that for an attribute value $v_0 \in V_a$ $D_{a,p}(v_0) \neq \emptyset$, but $\hat{f}_{a,p}(v_0)$ is not defined, then the distribution set should be redefined and v_0 should be removed, so that

$$D_{a,p}(v_0) = \emptyset \iff v_0 \notin Dom(\hat{f}_{a,p}).$$

Example 5.14
Let us consider grammar $G = (\{A, B, S\}, \{a, b\}, \mathcal{P}, S)$ with the following rules and attribute definitions:

$$\begin{aligned}
S \to AB \quad & S.w = A.w + B.w \\
A_0 \to aA_1 \quad & A_0.w = 1 + A_1.w \\
A \to a \quad & A.w = 1 \\
B_0 \to bbB_1 \quad & B_0.w = 1 + B_1.w \\
B \to bb \quad & B.w = 2
\end{aligned}$$

This grammar generates language $L(G) = \{a^n b^{2m} | n, m \in \mathbb{Z}^+\}$. Synthesized attribute $w \in V_w$ with $V_w = \mathbb{N}$ contains the width, which is the length of the frontier. This attribute can be made forced synthesized by defining the following rule-specific distribution functions:

$$\begin{aligned}
S \to AB \quad & f_w(v_0) = (v_1, v_2), \text{ where } v_1 + v_2 = v_0 \\
A_0 \to aA_1 \quad & f_w(v_0) = (v_0 - 1) \\
A \to a \quad & \varnothing \\
B_0 \to bbB_1 \quad & f_w(v_0) = (v_0 - 2) \\
B \to bb \quad & \varnothing
\end{aligned}$$

This example shows several properties of the distribution functions, therefore it is useful to examine each rule separately.

$S \to AB$ – For this rule the distribution function is a random selection. The distribution sets contain all pairs having the sum of v_0, and none of them is empty.

It is also possible to use the following distribution function

$$\tilde{f}_w(v_0) = (v_1, v_2), \text{ where } v_1 = v_2 \text{ and } v_1 + v_2 = v_0.$$

With this modified function and appropriately changed distribution sets $D_w(2v+1) = \emptyset$ for $v \in \mathbb{N}$, the generated language can be changed to $L(G) = \{(a^{2n}b^{2n} | n \in \mathbb{Z}^+\}$.

$A_0 \to aA_1$ – Since there is only one nonterminal symbol on the right-hand side, the distribution function is deterministic. However, $D_w(0) = \emptyset$, because $0 - 1 = -1 \notin V_w$.

$A \to a$ – This rule has no nonterminal symbols on the right-hand side, therefore the attribute value is constant 1, thus the distribution function is not defined. The distribution set can be defined as described in Remark 5.9:

$$D_w(v_0) = \begin{cases} \{()\} & \text{if } v_0 = 1 \\ \emptyset & \text{otherwise} \end{cases}$$

In fact, for this rule the actual value of D_w is irrelevant, the only question is if it is empty or not, since v_0 does not have to be distributed, but its value must be the one defined by the rule, that is 1.

$B_0 \to bbB_1$ – This rule is similar to rule $A_0 \to aA_1$, because it also has only one nonterminal. Thus the distribution function is deterministic. There are two cases when the distribution set is empty: $D_w(1) = D_w(0) = \emptyset$.

$B \to bb$ – In this rule there are no nonterminal symbols on the right-hand side and the attribute is constant 2, therefore the distribution function is not defined, and the distribution sets are similar to the ones for rule $A \to a$:

$$D_w(v_0) = \begin{cases} \{()\} & \text{if } v_0 = 2 \\ \emptyset & \text{otherwise} \end{cases}$$

Two possible derivations using these rules and distribution functions with the attribute values in parentheses are shown below.

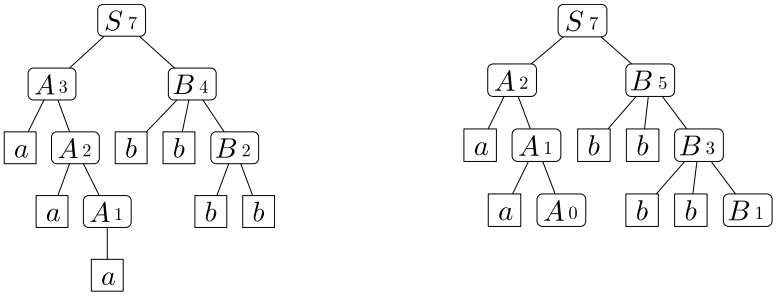
$$S(7) \Rightarrow A(3)B(4) \Rightarrow aA(2)B(4) \Rightarrow aaA(1)B(4) \Rightarrow aaaB(4) \Rightarrow aaabbB(2) \Rightarrow aaabbbb$$

$$S(7) \Rightarrow A(2)B(5) \Rightarrow aA(1)B(5) \Rightarrow aaA(0)B(5) \Rightarrow aaA(0)bbB(3) \Rightarrow aaA(0)bbbbB(1)$$

The first derivation is successful and yields a terminal word. The derivation tree belonging to this derivation is shown in Figure 5.4a. The second derivation fails for two reasons. In the third step, rule $A \to aA$ is applied yielding $A(0)$ that cannot be rewritten, because from A the shortest word that can be derived has length 1. This issue could have been avoided by removing 0 from V_w, because in that case $D_{w,A\to aA}(1) = \emptyset$, as $1 - 1 = 0 \notin V_w$ therefore rule $A \to aA$ cannot be selected.

The more serious issue can be seen in the last step, when $B(1)$ is inserted and there are no rules that can be applied to it, because every rule for B requires $w \geq 2$. Furthermore, there is no way avoiding $B(1)$ as soon as $B(5)$ is inserted, because only rule $B \to bbB$ can be applied unless $w = 2$, but that is never reached as w is always decreased by 2. Thus after applying the first rule, this derivation is condemned to fail. The solution is to redefine the distribution function not to allow odd numbers for rule $S \to AB$. The derivation tree for this derivation is shown in Figure 5.4b.

With the help of forced synthesized attributes, the evolutionary operators can be modified to respect the semantic constraints. Operators affected by this are random tree generation and crossover.



(a) successful (b) failed

Figure 5.4: Semantically constrained derivations

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. To achieve this, Algorithm 4.3 is replaced by Algorithm 5.1

```
RTG(root, limit, a, v_0)
 1  X := root.label
 2  if X ∈ Σ
 3     then return
 4  C := ∅
 5  for  each p : X → α in P
 6  do if (min[p] < limit) ∧ (D_{a,p}(v_0) ≠ ∅)
 7        then C := C ∪ {p}
 8  p̂ : X → β = RuleSelect(C)
 9  childlimits := LimitDistribution(limit, β) // random distribution of limit
10  values := f̂_{a,p̂}(v_0)
11  for  each Y in β
12  do child := node(Y)
13     RTG(child, childlimits[Y], a, values[Y])
14     add_child(root, child)
```

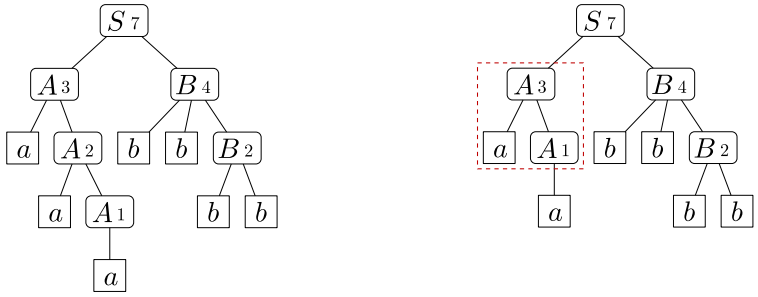
Algorithm 5.1: Random tree generation with semantic constraints

The changes are in line 6, 10 and 13. In line 6, when a rule is examined, not only its relation to the limit is checked, but also distribution set $D_{a,p}(v_0)$ is verified. Line 10 is added to the original algorithm to distribute v_0 among the children. Finally, line 13 has been updated so that the forced synthesized value is passed to the children.

As shown in Figure 5.4b, the derivation might lead to a dead-end. Furthermore, the derivation might also fail due to the size limitation. If, for example, the derivation shown before starts with $v_0 = 50$, but the height of the tree is limited to 10, the derivation will never result in a terminal word, thus it will be considered as failed derivation. Finding a general solution for avoiding these issues is a very complicated task, but specific solutions will be given in this thesis whenever examples for semantically constrained derivation are used in later chapters.

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. A conflict situation based on Example 5.14 is shown in Figure 5.5. A subtree with root labeled with A is replaced by another subtree with the same



(a) parent (b) offspring

Figure 5.5: Conflict caused by crossover

label. However, if forced synthesized attribute *width* is not checked, a subtree with attribute value 2 can be replaced with another one with attribute value 1. The result contradicts distribution function $f_w(v_0) = (v_0 - 1)$ of rule $A_0 \to aA_1$ at the marked place. In case of standard synthesized attributes this would not be a problem, because Algorithm 4.1 updates them after the operator was applied. However, forced synthesized attributes cannot be updated this way, because such an update might result an attribute value at the root different than the pre-defined value. In our example, *width* would be 6 at the root node labeled with S, but the pre-defined value was 7.

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 have to be labeled not only with nonterminals, but also with the values of the forced synthesized attribute, which means that there will be separate pools for subtrees having the same label at the root but different values for the attribute.

A straightforward implementation of an updated crossover operator is shown in Algorithm 5.2. The only change to the original algorithm is Line 3, where instead of the label, an index function is used to find the appropriate pool. The index value is calculated based on the label and the forced synthesized attribute values:

$$idx : \mathcal{N} \times V_{a_1} \times \cdots \times V_{a_n} \to \mathbb{N}$$

As it was mentioned in Section 4.5.5, there is a chance that 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. Normally, the number of pools depends on the number of nonterminals, that is $|\mathcal{N}|$. However, when forced synthesized attributes are also used, the number of possible pools is increased up to $|\mathcal{N}| \cdot |V_{a_1}| \cdot \cdots \cdot |V_{a_n}|$. Therefore, pool crossover cannot be applied when there are too many nonterminal–attribute value combinations, unless the population size is increased.

5.2.3 Constraint predicates

Forced synthesized attributes introduce constraints on the derivation by requiring the synthesized values to be equal to the ones passed down on the tree. That means, equality is explicitly required

```
PoolCrossover(population : P)
 1  for each T in P
 2  do N := SelectNode(T)
 3     I := Idx(N.label, N.fsa[])
 4     pools := pools ∪ {I}
 5     pool[I] := pool[I] ∪ {N}
 6  for each I in pools
 7  do for each N in pool[I]
 8     do pool[I] := pool[I] \ {N}
 9        M := RandomElement(pool[I])
10        if M ≠ nil
11          then SwapTree(N, M)
```

Algorithm 5.2: Pool crossover for derivation trees

by Definition 5.5. This equation can, however, be changed to allow an arbitrary predicate P_a to define the elements of the distribution set as follows

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

Example 5.15 *(Constraint predicate)*
Example 5.14 can be modified such that width w of the generated tree is limited by a pre-defined constant. For this, the *less-than-or-equal* ($\leq$) relation is used as predicate P_w and the distribution functions are defined as follows

$$\begin{aligned}
S \to AB \quad & \hat{f}_w(v_0) = (v_1, v_2), \text{ where } v_1 + v_2 \leq v_0 \\
A_0 \to aA_1 \quad & \hat{f}_w(v_0) = (v_1), \text{ where } 1 + v_1 \leq v_0 \\
A \to a \quad & \varnothing \\
B_0 \to bbB_1 \quad & \hat{f}_w(v_0) = (v_1), \text{ where } 2 + v_1 \leq v_0 \\
B \to bb \quad & \varnothing
\end{aligned}$$

The effect of the constraint introduced in Example 5.15 is very similar to the limit used by the random tree generator. Therefore, it is possible to remove the special tree limitation from the algorithm and replace it with a forced synthesized attribute using a constraint predicate. However, tree limitation makes use of special rule and nonterminal properties min_r and min_s guiding the rule selection so that the derivation is guaranteed to be successful, whereas a constrained derivation might get to a dead-end as shown in Figure 5.4b. Therefore, for tree size limitation it is better to use the special operator instead of constraint predicates.

5.2.4 Examples

In this section two examples are shown to demonstrate how forced synthesized attributes can be used to restrict the search space and improve the algorithm.

Matrix multiplication

Since matrix multiplication is associative, if a sequence of matrices of different sizes is multiplied, it is possible to optimize the costs by choosing the order in which the multiplication is carried out.

Finding the optimal order means finding an optimal set of parentheses. For the sake of simplicity, enough parentheses are used so that the usual left-to-right order of multiplication is completely superseded. This means that for $n > 1$ matrices exactly $n - 1$ pairs of parentheses are used. A random sequence of opening and closing parentheses is not always correct, but the set of valid sequences can be generated by the context-free grammar

$$G_{par} = (\{S\}, \{(,)\}, \mathcal{P}_{par}, S)\,,$$

where the set of rules $\mathcal{P}_{par}$ contains two rules:

$$\begin{aligned} S &\rightarrow (S)S \\ S &\rightarrow \lambda \end{aligned}$$

This grammar generates sequences of $n > 0$ pairs, a number which can be calculated as a synthesized attribute:

$$\begin{aligned} S_0 \rightarrow (S_1)S_2 \qquad & S_0.n = S_1.n + S_2.n + 1 \\ S \rightarrow \lambda \qquad & S.n = 0 \end{aligned}$$

To reduce the set of generated sequences to a given number of pairs, one can take attribute n and make it a forced synthesized attribute using the following distribution sets:

$$\begin{aligned} S_0 \rightarrow (S_1)S_2 \qquad & D(v_0) = \{(a,b) \in \mathbb{N}^2 \mid a + b = v_0\} \\ S \rightarrow \lambda \qquad & \text{if } v_0 = 0 \text{ then } D(v_0) = \{()\} \\ & \text{otherwise } D(v_0) = \emptyset \end{aligned}$$

The empty distribution sets are $D_{S\rightarrow(S)S}(0)$ for the first rule and $D_{S\rightarrow\lambda}(v_0)$, where $v_0 > 0$ for the second rule. This means that for any $v_0 \in \mathbb{N}$ there is a rule that can be applied, thus no failed derivations will occur.

Boolean regression

The standard example used in this thesis was the Boolean regression problem with 5 variables. As the results in Section 4.6 show, DTGP was able to find a correct function with 38% success rate. After discussing the possibility of semantic constraints, the question arises if it was possible to restrict the search space to the set of correct Boolean functions using a properly defined forced synthesized attribute.

In Section 5.1.2 it was presented, how the represented Boolean function can be calculated as parameter. Thus one might try to use it to define a forced synthesized attribute. For some of the rules there is exactly one nonterminal on the right-hand side, and the calculation of the parameter is a simple assignment, that is the distribution function is deterministic. The two rules where the distribution function is not deterministic are the following:

$$\begin{aligned} E_0 \rightarrow E_1 \vee T \qquad & E_0.p := E_1.p|T.p \\ T_0 \rightarrow T \wedge F \qquad & T_0.p := T_1.p\&F.p \end{aligned}$$

The distribution function will be described only for the first rule. For the second rule the same approach can be used. The distribution set is the following:

$$D_{E\rightarrow E\vee T}(v_0) = \{(a,b) \mid a|b = v_0\}.$$

For the definition of the distribution function, let us recall the truth table of the *or* function for values 0 (false), 1 (true) and $\star$ (unknown).

$\vee$	0	1	$\star$
0	0	1	$\star$
1	1	1	1
$\star$	$\star$	1	$\star$

The use of the unknown value makes sense, because if one parameter is true, the result will be true, independently from the value of the other parameter, as shown in the middle of the last column and the last row. Therefore, when the distribution function is defined, one might decide not to restrict the value, but use $\star$ instead. With the help of $\star$ one can define a more-generic-or-equal relation $\preceq$ for the values, pairs or sequences as follows:

Definition 5.16 *(More generic or equal relation)*
For sequences $a = a_1a_2 \dots a_n$ and $b = b_1b_2 \dots b_n$ with $a_i, b_i \in \{0, 1, \star\}$, a is more generic than or equals to b ($b \preceq a$), if and only if

$$\forall 1 \leq i \leq n : \quad a_i \in \{b_i, \star\}.$$

It means that for every position, the value in a is either the same as in b, or it is $\star$. If $b \preceq a$, but $a \neq b$ then a is more generic than b ($b \prec a$).

The distribution function can be defined bitwise. Given $v_0 = u_1u_2u_3 \dots u_n$ with $u_i \in \{0, 1, \star\}$ the goal is to construct $a = a_1a_2a_3 \dots a_n$ and $b = b_1b_2b_3 \dots b_n$ such that $a|b = v_0$, that is for each $1 < i < n$ $a_i \vee b_i = u_i$. The possible values for a_i and b_i for a given u_i can be found in the truth table, and they are also listed in Table 5.1.

u_i	a_i	b_i
0	0	0

u_i	a_i	b_i
1	0	1
	1	0
	1	1
	1	$\star$
	$\star$	1

u_i	a_i	b_i
$\star$	0	$\star$
	$\star$	0
	$\star$	$\star$

Table 5.1: Possible distribution values of a single bit

To construct a and b, the distribution function selects the values for a_i and b_i from a randomly selected line of the appropriate table. The resulting distribution function will work correctly, and the generated derivation trees will represent the pre-defined Boolean function. However, if the tables are examined closely, one can see that the distribution function has the possibility to distribute $\star$ as $(0, \star)$, or $(\star, 0)$, which means introducing a new restriction for a given bit. This is not necessary at all, and since $(\star, \star)$ is more generic than either $(0, \star)$ or $(\star, 0)$, the distribution function can be changed to always distribute $\star$ as $(\star, \star)$. A similar observation can be made for distributing the value 1 as well: $(1, \star)$ is more generic than $(1, 0)$, $(\star, 1)$ is more generic than $(0, 1)$ and both of them are more generic than $(1, 1)$. The updated tables with the more specific pairs removed is shown in Table 5.2.

u_i	a_i	b_i
0	0	0

u_i	a_i	b_i
1	1	$\star$
	$\star$	1

u_i	a_i	b_i
$\star$	$\star$	$\star$

Table 5.2: Possible distribution values of a single bit, without the more specific pairs

If the value $\star$ is used, the distribution sets for the four rules with no nonterminals on the right-hand side can also be relaxed. As an example, consider rule

$$V \to \bar{x}_0 \qquad V.p := \mathtt{1010}_2.$$

Normally for $v_0 \neq \mathtt{1010}_2$, the distribution set $D_{V\to\bar{x}_0}(v_0)$ would be empty. By allowing $\star$ the distribution set can be defined to be also nonempty if v_0 is more generic than $\mathtt{1010}_2$, that is

$$D_{V\to\bar{x}_0}(v_0) \neq \emptyset \iff \mathtt{1010}_2 \preceq v_0.$$

With the rules above one can implement a semantic constraint that will restrict DTGP to the set of correct Boolean functions, so the success rate of the algorithm is expected to be 100%. An example for a derivation tree created using this constraint can be seen in Figure 5.6.

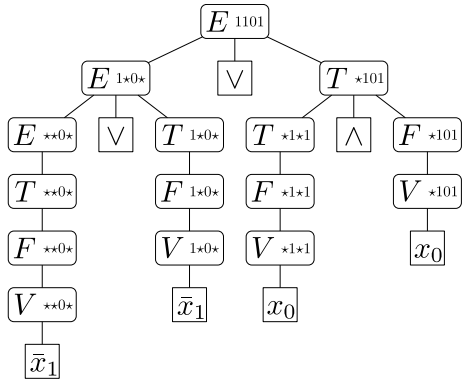


Figure 5.6: Derivation tree for semantically constrained Boolean regression

5.2.5 Limitations of semantic constraints

When semantic constraints are defined, one has to consider the failed derivations as well. With the Boolean regression grammar there are four rules that can result in failed derivation: the rules that rewrite V to a variable. Since there are no other rules for V, prematurely selecting rule $F \to V$ can yield a failed derivation. Therefore the distribution set for that rule has to be defined to allow v_0 values that are more generic than or equal to the values represented by the variables. It is useful to examine the number of these v_0 values to see the chance that rule $F \to V$ can be applied. Unfortunately this chance is almost exponentially decreasing as the length of the representation is increasing.

Lemma 5.17 *(Proportion of distributable values)*
Given a Boolean regression problem with $n = 2^k$ bit function representations, where k is the number of variables. Assume that the grammar from Section 5.1.2 is used with a semantic constraint on the function. Let us denote the set of possible values of forced synthesized attribute p with $\mathcal{V}$. That is $\mathcal{V} = \{\{0, 1, \star\}^n\}$. Let us denote the set of values for which the distribution set is non-empty with $\mathcal{V}^*$. That is $\mathcal{V}^* = \{v_0 \mid D_{F\to V}(v_0) \neq \emptyset\}$. In this case the proportion of the values that can be distributed is decreasing almost exponentially with respect to n, that is

$$\frac{|\mathcal{V}^*|}{|\mathcal{V}|} = \mathcal{O}(\log n \cdot c^{-n}), \text{ with } c > 1.$$

Proof 5.18
Given a rule $V \to x_l$ with function representation $b = b_1 b_2 \dots b_n$. The set of values for which the distribution set is not empty, that is the values that are more generic than or equal to b is

$$\mathcal{V}_l = \{v = v_1 v_2 \dots v_n \mid v_i \in \{b_i, \star\}\}.$$

Therefore $|\mathcal{V}_l| \leq 2^n$. For rule $V \to \bar{x}_l$ the set is denoted by $\bar{\mathcal{V}}_l$.

A value can be distributed by rule $F \to V$, if it can be distributed by $V \to x_j$ or $V \to \bar{x}_j$ for any $1 \leq j \leq k = \log n$. Thus

$$\mathcal{V}^* = \bigcup_{i=1}^{\log n} \left(\mathcal{V}_l \cup \bar{\mathcal{V}}_l\right).$$

This implies that

$$|\mathcal{V}^*| \leq \sum_{i=l}^{\log n} \left(|\mathcal{V}_l| + |\bar{\mathcal{V}}_l|\right) \leq \sum_{i=l}^{\log n} 2 \cdot 2^n = 2 \log n \cdot 2^n$$

Since each bit can have three values $(0, 1, \star)$, the total number of possible values is 3^n. Therefore

$$\frac{|\mathcal{V}^*|}{|\mathcal{V}|} \leq \frac{2 \log n \cdot 2^n}{3^n} = 2 \log n \frac{2^n}{3^n} = 2 \log n \left(\frac{2}{3}\right)^n = 2 \log n \cdot 1.5^{-n} = \mathcal{O}(\log n \cdot c^{-n}), \text{ with } c > 1.$$

□

Generally, when the derivation tree is constructed and the required phenotype is passed down, it is getting more and more generic, so the probability that rule $F \to V$ can be applied is increasing. It is therefore interesting to examine the proportion of distributable values if the number of elements with $\star$ is known.

Lemma 5.19
Let us denote the number of $\star$ values in a function representation u with $|u|_\star$. Let us denote the set of values having $\star$ in exactly m places with $\mathcal{V}_m$. That is $\mathcal{V}_m = \{u \in \mathcal{V} \mid |u|_\star = m\}$. Let $\mathcal{V}_m^* = \mathcal{V}_m \cap \mathcal{V}^*$. In this case

$$\frac{|\mathcal{V}_m^*|}{|\mathcal{V}_m|} = \Omega(c^{m-n}), \text{ with } c > 1.$$

Proof 5.20
Let us select a vector of m positions $\bar{a} = (a_1, \dots, a_m)$ with $1 \leq a_i \leq n$ and $a_i \neq a_j$ if $i \neq j$. For such a vector define the set of function representations that contain $\star$ exactly at these locations:

$$\mathcal{V}_{\bar{a}} = \{u = u_1 \dots u_n \in \mathcal{V}_m \mid u_{a_i} = \star\}.$$

Set $\mathcal{V}_m$ can be described as the disjoint union of these sets. The number of these sets $M \geq 1$ is not important.

$$\mathcal{V}_m = \biguplus_{\bar{a}} \mathcal{V}_{\bar{a}}$$

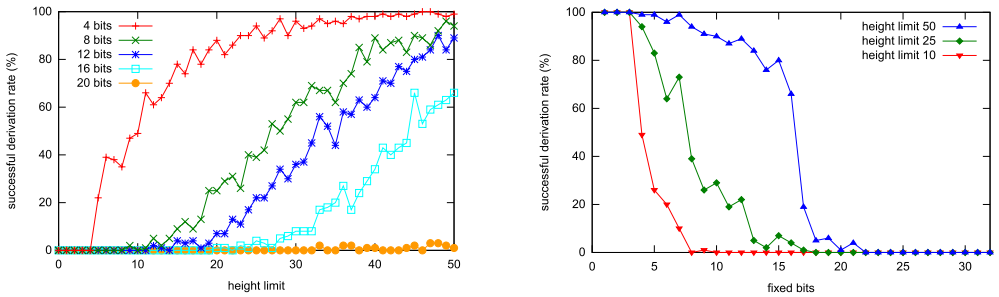
Let us examine the proportion of the elements in such a set that can be distributed by rule $F \to V$. There are m locations with $\star$, and the remaining $n - m$ locations can have two values (0 or 1). That is $|\mathcal{V}_{\bar{a}}| = 2^{n-m}$. $F \to V$ can distribute a value, if all these $n - m$ places match the sequence represented by one of the $2 \log n$ rules for V. There can be less than $2 \log n$ matches, but there is at least one. Therefore $1 \leq |\mathcal{V}_{\bar{a}} \cap \mathcal{V}^*|$. This means:

$$\frac{|\mathcal{V}_m^*|}{|\mathcal{V}_m|} = \frac{|\biguplus_{\bar{a}} (\mathcal{V}_{\bar{a}} \cap \mathcal{V}^*)|}{|\biguplus_{\bar{a}} \mathcal{V}_{\bar{a}}|} \geq \frac{M \cdot 1}{M \cdot 2^{n-m}} = 2^{m-n} = \Omega(c^{m-n}), \text{ with } c > 1.$$

□

Rule $E_0 \to E_1 \vee T$ distributes 1 as $(1, \star)$ or $(\star, 1)$, therefore if the number of ones in E_0 is x, the expected value of ones in both E_1 and T is $x/2$. A similar observation can be made for zeroes and rule $T_0 \to T_1 \wedge F$. Thus these rules replace on average half of the zeroes and ones with $\star$, until there is only one left of each. Therefore, it is expected that by increasing the height limit of the derivation trees one can ensure that the semantically constrained derivations will not fail.

To test this hypothesis, the forced synthesized attribute described above was used with the Boolean regression problem. As defined in the previous chapter, the function is represented by a 32bit value. On this value constraints of various sizes have been applied. First the last bit was fixed, then the last two, and so on. For each of these, random tree generation has been tested with tree height limits up to 50. Using a given constraint and a height limit, the random tree generator has been started 100 times. The number of successful derivations is shown in Figure 5.7.

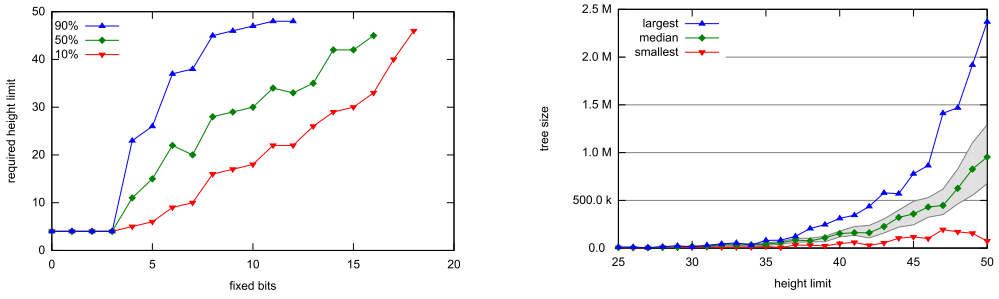


(a) for increasing height limit

(b) for increasing constraint size

Figure 5.7: Successful derivations using phenotype constraint

It can be seen that as the height limit is increased, the success rate is increasing as well, at least if 16 or less bits are fixed. It is also interesting to see, what height limit is required to achieve a given success rate. Figure 5.8a contains this information with respect to the number of fixed bits for success rates 10%, 50% and 90%. However, increasing the height limit increases the tree sizes as well. This information has also been measured and the results for 16 fixed bits are plotted on Figure 5.8b. As expected, the size of the tree is increasing exponentially, and for height limit 50 the median almost reaches a node number of 1 million.



(a) required height limit wrt. constraint size

(b) tree sizes at height limits

Figure 5.8: The effect of increased constraint on tree height and tree size

The results show that theoretically it is possible to introduce a semantic constraint on the represented Boolean function. However, practice shows that it is useful only up to a certain limit, because larger constraints require larger tree height limits, which in turn cause exponentially increasing tree sizes. Even though this particular issue only affects random tree generation, 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. Usually evolutionary algorithms use a randomized selection. Furthermore, genetic programming methods, like DTGP use random node selection and randomized tree generation. All three parts will be examined in this section to ensure that the search space remains intact and no unwanted bias is introduced during the process.

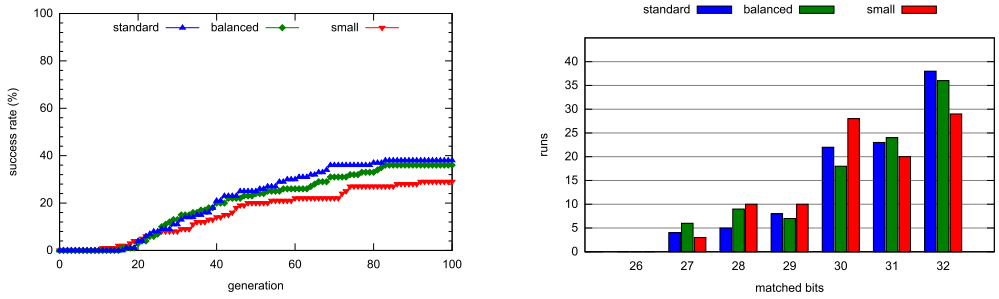
5.3.1 Selection

There are various selection types that are used with evolutionary algorithms, and any of these can be used with DTGP. The selection operates based on the fitness value, which is usually a numerical type, therefore a random selection is easy to implement. For example, implementing fitness proportional selection is straightforward using a roulette-wheel algorithm. Thus, regarding selections, DTGP is not different from any other EA.

5.3.2 Random node selection

As it was discussed in Section 4.5.2, random node selection was designed to take the node weight into account, thus the selection probability is proportional to the node weight. By default, that is if node weight is 1, it guarantees that each node has the same probability to be selected, as it can be seen in Figure 4.8b.

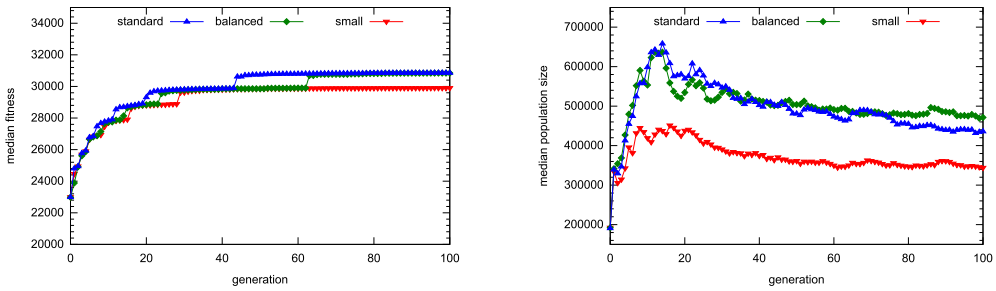
In Section 5.1.4 some examples were shown for defining a nonstandard node weight. To check the influence of this setting on the algorithm, some tests were made with the Boolean regression problem. The three previously mentioned examples were used for selection weight, and 100 independent runs were carried out with each. The results are shown in Figure 5.9.



(a) success rates (b) final results

Figure 5.9: Regression test results with various node selection weights

Of the three node weights, the standard one provides the best results. An explanation for this is that the other options restrict the search space by decreasing the probability of certain subtrees to be replaced. For mutation it is usual to change only small subtrees, but if the same selection weight is also used for crossover, it will prevent large trees to be swapped. The median value of the best fitness is plotted in Figure 5.10a, it also shows that the standard selection weight gives the best results. However, examining the population size plotted in Figure 5.10b one can see that using a selection weight that prefers small trees, the total size of the population can be significantly reduced. Thus, one can decide to allow a reduction of the success rate in exchange for the improved memory consumption caused by the reduced population size.



(a) median value of the best fitness

(b) median of the population size

Figure 5.10: Population statistics for various node selection weights

5.3.3 Random tree generation

The elements of the solution space are discovered by creating derivation trees that contain the solutions at their frontier. However, the distribution of the derivation trees is not the same as the distribution of the solutions. The relation between the space of derivation trees and solution space might be quite complex. Therefore even if a tree generator appears properly randomized, it might introduce a search bias on the solution space. The concerned steps in the random tree generator are the random rule selector and the subtree limit distributor, both of which will be examined.

The simplest way to examine the distribution in the solution space is to use a finite, ordered set for phenotypes, so that a histogram can be plotted using a two-dimensional chart. However, it is also important to examine more complex structures. Although these cannot be plotted directly on a histogram, some of their properties, like the derivation tree size can be used for such a purpose.

For the first type of test, a finite subset of positive integer numbers will be used. The elements of such a set can be generated by several grammars. The one used for our tests is the following: $G_i = (\{S, I, D, P\}, \{0, \ldots, 9\}, \mathcal{P}_i, S)$ where $\mathcal{P}_i$ is

$$
\begin{array}{rcl}
S & \rightarrow & PI \\
S & \rightarrow & P \\
I & \rightarrow & DI \\
I & \rightarrow & D \\
D & \rightarrow & 0 \mid 1 \mid 2 \mid \ldots \mid 9 \\
P & \rightarrow & 1 \mid 2 \mid \ldots \mid 9
\end{array}
$$

The idea is straightforward: an integer starts with a digit that is not zero (P) and it is followed by a sequence of digits (I). A sequence of digits is either a single digit (D) or a digit followed by a sequence.

All the nonterminals have multiple rules, however all those for D and P act in the same way: they generate a single terminal symbol. However, the ones for S and I are significantly different. The ones creating P or D respectively will end up in a single terminal symbol in the following step, but PI and DI can be rewritten to practically any possible integer.

It is also possible to create a more advanced grammar and use different rules for S and I and

remove nonterminal P completely:

$$\begin{aligned} S &\rightarrow I \\ I &\rightarrow II \\ I &\rightarrow D \end{aligned}$$

Let us denote the grammar using these rules with G_i'. This grammar will also need a semantic constraint so that the first digit is not 0.

To test the distribution of the generated solutions, a width limit of 3 has been introduced to limit the numbers to 999. Then 10^6 trees were generated using both grammars and the histogram of the generated numbers has been plotted as shown in Figure 5.11. These results show that

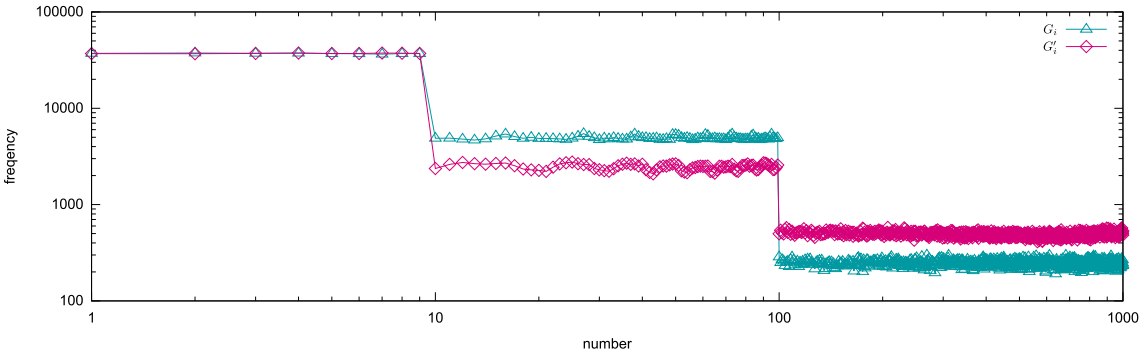


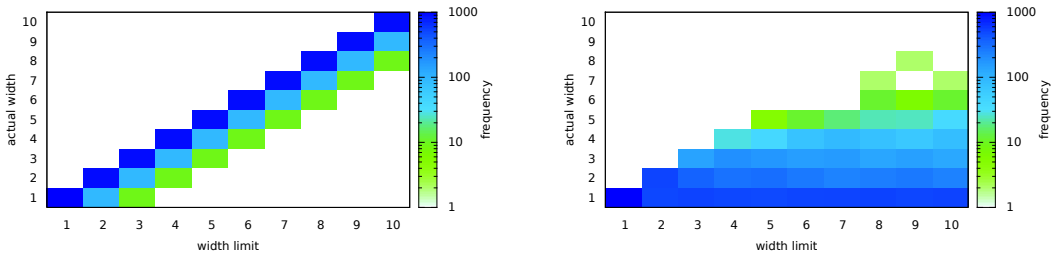
Figure 5.11: Distribution of generated integers

single digit numbers have far higher probability than two or three digit numbers, and three digit numbers have very low probability, although G_i' shows some improvement. On the other hand, the frequency of integers with the same number of digits is comparable. This means the distribution among trees of the same height or width is fairly uniform. Thus, the main issue is to increase the probability of generating larger trees. The quality of the random tree generator in this aspect can be examined using a different type of chart as well.

When a limit is set on the tree size and random trees are generated, it is expected that trees of all sizes within the limit are generated, and since the number of possible trees is growing with the size, one expects to have more of those. This information for increasing size limits can be plotted using so called heatmaps, that is two-dimensional matrices with colors representing the frequencies of the generated sizes. The expected results and the actual results using G_i are plotted in Figure 5.12.

Note that the expected results might not represent the ideal case. For DTGP it is better to have higher probability for smaller trees to avoid bloating that is usually causing a problem for GP. Furthermore, it is not necessary to create large trees with the random tree generator, they can be created later from smaller trees via mutation and crossover. Nevertheless, RTG is expected to generate at least a few large trees up to the limitations.

Similar plots can also be created for solution spaces that are not ordered or not finite. In the rest of this chapter such heat maps will be created for the Boolean regression grammar introduced earlier.



(a) expected results

(b) actual test results with G_i

Figure 5.12: Tree width frequencies for various width limits

Random rule selection

As shown above, the grammar has an influence on the random tree generation. However, sometimes it is not obvious how to change the grammar, and its influence on the randomization might be also difficult to foresee. On the other hand, random rule selection can be changed to assign weights to the rules. Some possibilities to define weights are listed below:

Minimum value of the rule – this is the smallest tree that can be generated starting with the given rule. The derivation cannot end in a smaller tree than the minimum, thus it provides a lower limit for the generated tree size.

Number of nonterminals – more nonterminals mean more possibilities for extending the tree, resulting larger trees.

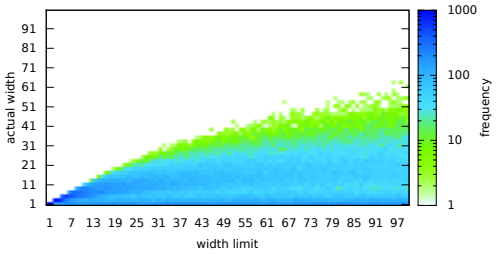
Number of follow-up rules – this is the sum of the number of rules for each nonterminal on the right-hand side. This represents the number of possibilities for continuing the derivation after a rule has been applied. Instead of the sum one can also use the product.

The Boolean regression problem has been tested with various weights. During each test the width limit was between 1 and 100, and 1000 trees were generated. The actual width of each tree has been measured and finally the heatmaps have been plotted. Note that the grammar defined in Section 4.6 will only generate words of odd length, therefore rows representing even lengths were removed from the picture. The results are shown in Figure 5.13.

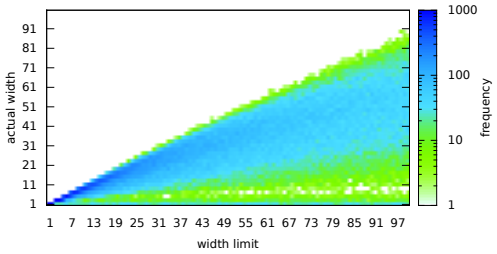
When the minimum tree size is used as weight, the random tree generator is able to use most of the limit, that is trees almost up to the limit are generated. For other rule selection weights, the limit is used when it is small enough, but when it reaches 100, only a few trees are generated with width above 50, so the effect of increasing the limit from 50 to 100 is very small.

To see the effect of the rule selection on the final results, the Boolean regression problem was tested with two setups: first the minimum tree size, then the product of the follow-up rules were used as selection weight. In both cases 100 independent runs were carried out, and the final results are plotted in Figure 5.14. They show that using the tree minimum provides the best results; although the success rate slows down earlier, it still keeps increasing during the last steps, because the algorithm can examine a larger part of the search space, thus it loses diversity in a slower rate.

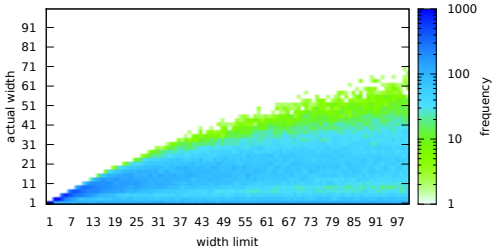
The behavior of the algorithm can also be examined during the generations: median fitness values and median population sizes are plotted in Figure 5.15. The best fitness values are similar, but there is a difference in the population size. When the product of follow-up rules is used, the



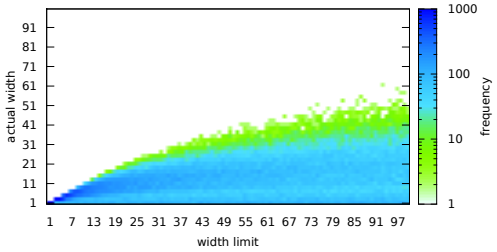
(a) no weight



(b) weighted by minimum value



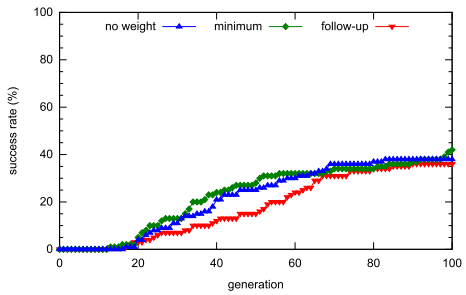
(c) weighted by number of nonterminals



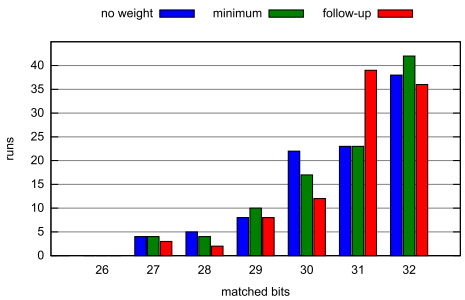
(d) weighted by product of follow-up rules

Figure 5.13: Tree width frequencies for the expression grammar

random tree generator generates smaller trees, so the total size of the population is smaller right from the beginning. On the other hand, if the rule minimum is used, the generated trees are larger, but that does not necessarily mean that the population size gets bigger, because it is counteracted by the size factor in the fitness value. In fact, it provides the opportunity for the mutation to replace larger trees, so a larger portion of the search space is examined and better individuals are found. Due to the size factor in the fitness value, better individuals also mean smaller individuals, thus the total size of the population decreases as the algorithm progresses.

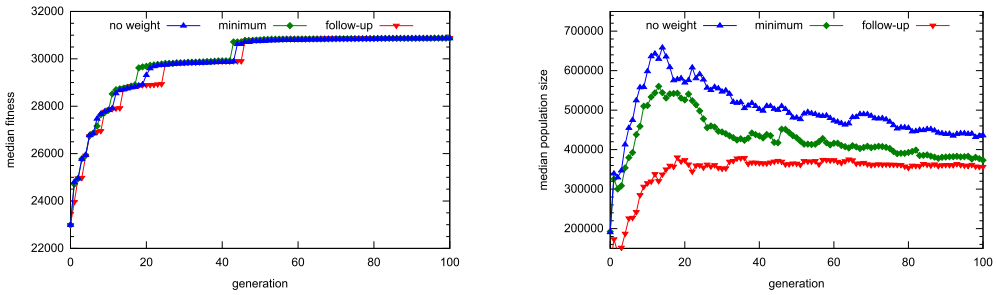


(a) success rates



(b) final results

Figure 5.14: Regression test results with various rule selection weights



(a) median value of the best fitness

(b) median of the population size

Figure 5.15: Population statistics for various rule selection weights

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. For the terminal nodes this value is constant, that is 1 in case the size or the width is limited, and 0 if the height is limited. If the height is limited, the distribution strategy is very simple: for each nonterminal the height limit is one less than the height limit for the parent. If the size or the width is limited, the calculation schema can be more complicated.

To calculate the limits one has to consider two factors: the limit for the parent node, and the minimum tree sizes for the child nodes. The minimum tree sizes for nonterminal symbols are given by min_s as described in Section 4.5.1. Usually there is a difference between the limit on the parent and the combined minimum sizes. This difference can be arbitrarily distributed among the subtrees, and there are various strategies how to do this:

Random – for each nonterminal the same weight is used, this is the standard setting.

Weighted – each nonterminal is weighted, for example by the minimum tree size.

Sequential – after limits are distributed, subtrees are created sequentially, for example from left to right. Then for each tree the actual size is measured and the unused limit is re-distributed among the remaining nonterminals.

The sequential strategy is not used because it introduces a bias: subtrees further to the right get larger limits. However, the weighted distribution might improve random tree generation, as it gives larger limits for nonterminals that are expected to be able to generate larger trees. Nevertheless, the tests showed no significant improvement for the Boolean regression problem or for the integer generation, therefore it is enough to use the standard limit distribution strategy.

5.4 Summary

In this chapter some details of the DTGP algorithm have been examined, and opportunities for improvements and extensions have been identified. First it has been demonstrated how the parameters stored in the derivation tree can be used to extend the method. These parameters can be used for several purposes:

Information retrieval – with the help of the parameters some information needed for evaluation can be made available in constant time. Such information includes *frontier*, *represented function*, partial or complete *fitness value*.

Operator configuration – the information stored at the nodes can be used to modify the behavior of operators, for example by biasing the *random node selection*.

It was also presented how *semantic constraints* can be added to the algorithm. Some examples were shown and the limitations were discussed. This extension has been achieved using the following concepts:

Distribution sets and functions define how a pre-defined parameter value can be distributed among the subtrees.

Forced synthesized attributes define a bottom-up parameter with the distribution sets and distribution functions.

Semantically constrained derivation is the method used to apply forced synthesized attributes during the derivation. It requires a specially defined random tree generator.

Index function for attributed nodes is used instead of node labels to identify the appropriate pool during a pool crossover.

Finally the randomized parts of the algorithm have been examined and the effects of weighting have been tested. The considered methods are:

Random node selection has an effect on the operators by determining what kind of subtrees are selected for replacement, which in turn can have an influence on the population size.

Rule selection has an effect on the random tree generator, and can positively influence the final results by widening the range of the search space covered by the algorithm.

Size limit distribution has a minor effect on the random tree generator.

6 DTGP applications

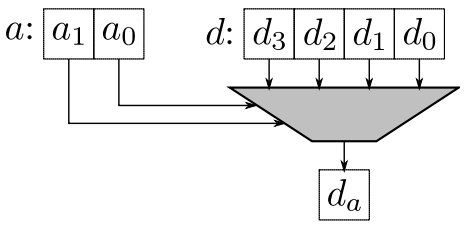
In the previous chapters it was presented how derivation tree based genetic programming can be used to solve a Boolean regression problem. As a GGGP approach, DTGP is applicable to any problem where the solution space can be described by a context-free grammar.

In this chapter some additional examples will be discussed briefly. The first example is the *6-Multiplexer* problem, which is a standard example for GP. The second example is the *traveling salesman problem*, which is often used to test optimization methods. DTGP has been used in a joint project with the Fraunhofer Institute for Integrated Circuits to optimize *finite input response filters*. [51] This practical application is summarized in Section 6.3.

6.1 6-Multiplexer

A standard example for genetic programming is the multiplexer problem. The multiplexer is a logical circuit with n address bits and 2^n data bits. Its output is the value of the data bit identified by the address bits. The simplest case is the 3-Multiplexer problem with 1 address bit and 2 data bits.

There are several ways to define the allowed circuits, for example one can allow logical gates or simple programs. In this section the set of operators $\{AND, OR, NOT, IF{-}THEN{-}ELSE\}$ will be used to represent solutions. The problem for $n = 2$, that is the 6-Multiplexer is illustrated in Figure 6.1. In many aspects this problem is similar to the Boolean regression example used in this thesis.



(a) setup

```
SIX-MULTIPLEXER(a, d)
1  if a_1
2     then if a_0
3             then return d_3
4             else  return d_2
5     else  if a_0
6             then return d_1
7             else  return d_0
```

(b) a solution

Figure 6.1: The 6-Multiplexer Problem

6.1.1 Unbiased solution

For the first test with DTGP we will use grammar G_{6m} as introduced by Whigham to demonstrate his GGGP system called CFG-GP [47]:

$$G_{6m} = \{\{S, B, T\}, \Sigma_{6m}, \mathcal{P}_{6m}, S\}, \text{ where}$$

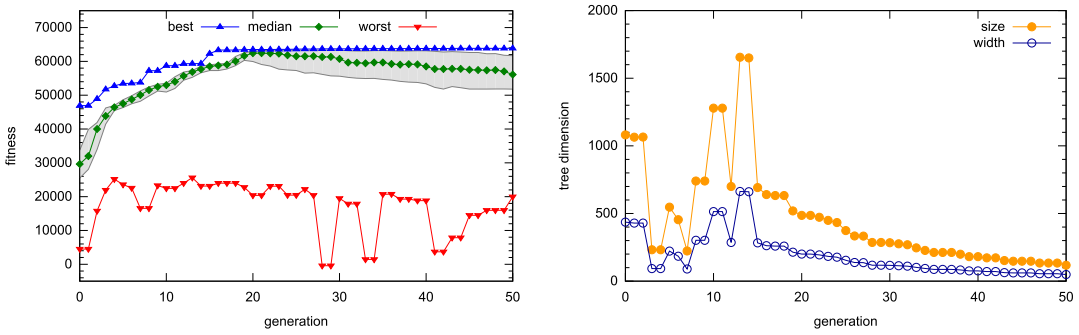
$$\Sigma_{6m} = \{and, or, not, if, a_0, a_1, d_0, d_1, d_2, d_3\},$$

and the rules in $\mathcal{P}_{6m}$ are the following:

$$\begin{aligned}
S &\rightarrow B \\
B &\rightarrow not(B) \\
B &\rightarrow and(B, B) \mid or(B, B) \\
B &\rightarrow if(B, B, B) \\
B &\rightarrow T \\
T &\rightarrow a_0 \mid a_1 \\
T &\rightarrow d_0 \mid d_1 \mid d_2 \mid d_3
\end{aligned}$$

The test setup was similar to the one used for Boolean regression: 1000 individuals were created and 100 steps were done. The fitness calculation was also the same, all 64 possible evaluations were checked and every match increased the fitness by 1000, then the size of the tree was subtracted to get the fitness value.

The fitness values and the best tree dimensions for a single run are shown in Figure 6.2. Since the most significant improvements occurred in the first half of the process, only the first 50 generations are plotted. The charts show that the single run could reach a fitness over 63000, meaning a 64-bit match.



(a) fitness values

(b) best individual dimensions

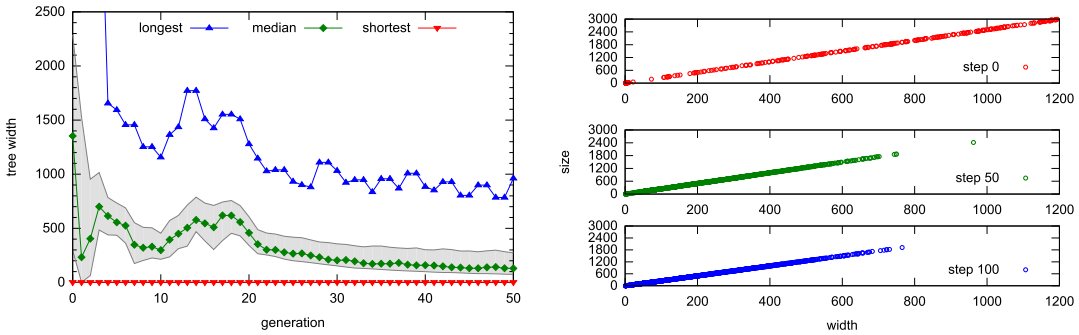
Figure 6.2: 6-Multiplexer single test results with G_{6m}

The best solution found by the single run is shown in Figure 6.3. Structurally it is similar to the simple manual solution, there are three *if* branches and four *return* statements for the four individual bits. The logic is a bit different, and there are some obvious places for manual adjustments (for example $\neg\neg d_0$), but the overall size is very small.

The tree dimensions during the single run are depicted in Figure 6.4. For the largest part of the population the width quickly drops below 500, and after 50 steps 75% of the individuals is shorter than 250. It can also be seen that there is an approximately linear relation between the tree size and the tree width, just like in the case of the Boolean regression problem.

```
SIX-MULTIPLEXER(a, d)
 1  if ¬(d0 ∧ ¬(a0 ∨ a1))
 2    then if ¬a0
 3           then return d2 ∧ a1
 4           else  if a1
 5                   then return d3 ∧ a0
 6                   else  return d1
 7    else  return ¬¬d0
```

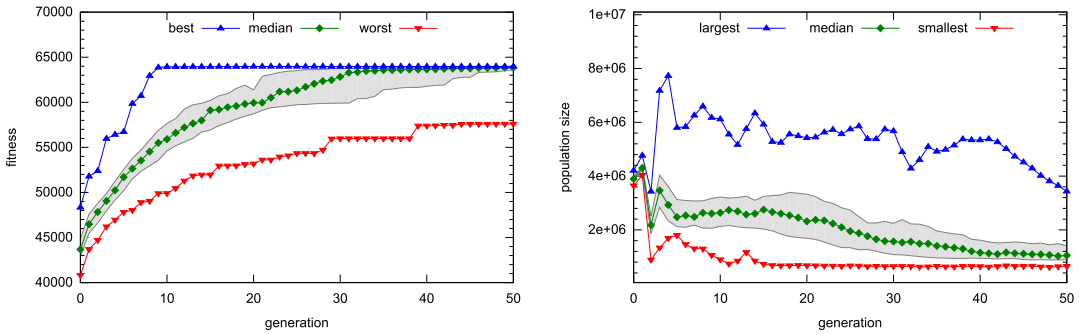
Figure 6.3: Best solution found by a single DTGP run



(a) tree width (b) width-size ratio

Figure 6.4: 6-Multiplexer individual dimensions

As no general conclusions can be made from a single run, another test was carried out in which 100 independent runs were made. The results are plotted in Figure 6.5. It can be seen that after 10 steps there was already at least one population with best fitness over 63000, and by generation 50 most of the populations reached that stage, that is they found a 64-bit match.



(a) fitness values (b) population size

Figure 6.5: 6-Multiplexer results

The results show that DTGP can solve the 6-Multiplexer problem without any special settings. However, it is interesting to examine if the results can be improved by using a different grammar, or additional options provided by DTGP.

6.1.2 Grammatical bias

Grammar G_{6m} is only one possibility to describe the solutions. Whigham achieved the best results using the following grammar, which is denoted by $G_{6m-if-a0-if-a1}$ in his work [49], but will be called G_{6mi} here for the sake of simplicity: $G_{6mi} = \{\{S, B, T, I\}, \Sigma_{6m}, \mathcal{P}_{6mi}, S\}$, where the set of nonterminals is the same as before and the rules in $\mathcal{P}_{6mi}$ are the following

$$\begin{aligned} S &\rightarrow if(a_0, I, B) \\ I &\rightarrow if(a_1, B, B) \\ B &\rightarrow not(B) \\ B &\rightarrow and(B, B) \mid or(B, B) \\ B &\rightarrow if(B, B, B) \\ B &\rightarrow T \\ T &\rightarrow a_0 \mid a_1 \\ T &\rightarrow d_0 \mid d_1 \mid d_2 \mid d_3 \end{aligned}$$

The first rule is changed and the second rule is new compared to $\mathcal{P}_{6m}$. These changes make the first two steps during a left derivation deterministic:

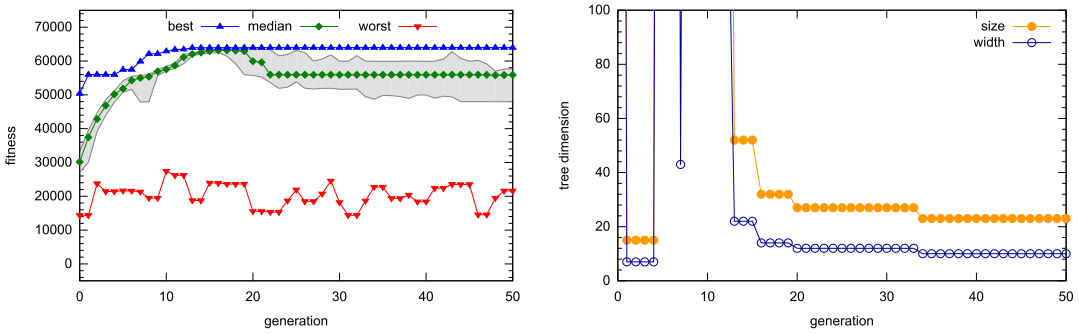
$$S \Rightarrow if(a_0, I, B) \Rightarrow if(a_0, if(a_1, B, B), B)$$

Therefore, the structure of each individual will be forced to the one shown in Figure 6.6.

```
SIX-MULTIPLEXER(a, d)
1  if a0
2     then if a1
3             then [B]
4             else [B]
5     else [B]
```

Figure 6.6: The structure of the individuals defined by grammar G_{6mi}

The fitness values and the dimensions of the best individuals during a single test run using G_{6mi} are shown in Figure 6.7. The solution can be seen in Figure 6.8.



(a) fitness values

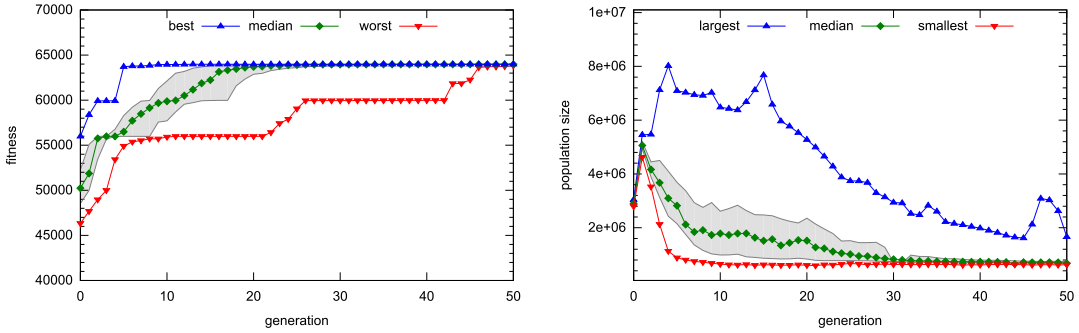
(b) best individual dimensions

Figure 6.7: 6-Multiplexer single test results with G_{6mi}

```
SIX-MULTIPLEXER(a, d)
1  if a0
2    then if a1
3           then return d3
4           else  return d1
5    else  if a1
6           then return d2
7           else  return d0
```

Figure 6.8: Best solution found by a single DTGP run using G_{6mi}

The result found by DTGP using G_{6mi} has the exact same structure as the manually created example. It is the direct consequence of the pre-defined format shown in Figure 6.6. Just like in the case of previous examples, 100 independent runs were carried out using grammar G_{6mi}. The results are shown in Figure 6.9. These plots show that using a grammatical bias, DTGP can find



(a) best fitness (b) population size

Figure 6.9: 6-Multiplexer test results for 100 independent runs with G_{6mi}

64-bit match for the 6-Multiplexer problem within 50 steps, but in most cases it is reached within 20 steps, and in the best case less than 10 steps are enough.

6.1.3 Semantic constraints

Although using G_{6mi} improves the results, the grammar itself contains too much information about the expected solution. In general it is not possible to use the knowledge that will only be found after the optimization problem is solved. However, some other restrictions can be defined without a-priori information regarding the optimal solution.

Looking at the result in Figure 6.3 one can see that the conditions for the *if* statements use not only the address bits, but the data bits as well. Furthermore, the returned value is calculated not only using the data bits, but also the address bits, although it can be considered as masking or selecting with the help of logical operators. Nevertheless, one might want to restrict the conditions to the address bits and the calculations to the data bits.

Such a semantic constraint can be introduced by defining a forced synthesized attribute called *type*. It will be used to show if the expression represented by the nonterminal symbol is address or data. Furthermore, a third value is used to show if the type is invalid, for example address and data are combined with logical *or*. That is

$$V_{type} = \{address, data, invalid\}$$

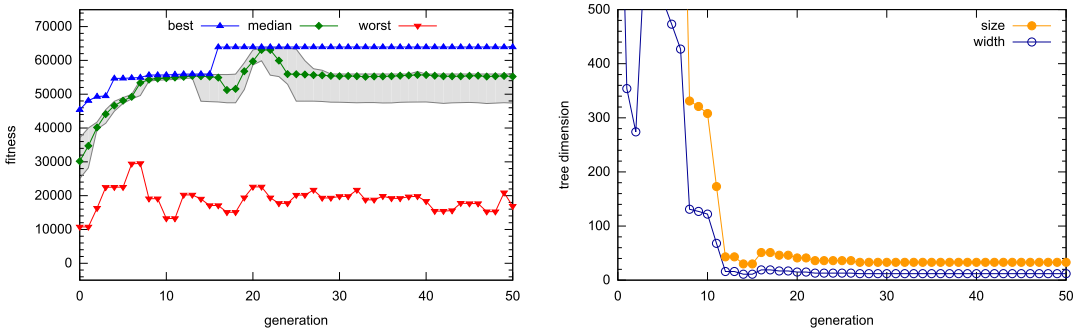
Attribute *type* is a synthesized attribute, as it can be calculated in a bottom-up fashion by extending grammar G_{6m} with the following rules:

$$
\begin{array}{rl}
S \to B & S.type = B.type \\
B_0 \to not(B_1) & B_0.type = B_1.type \\
B_0 \to and(B_1, B_2) \mid or(B_1, B_2) & \text{if } B_1.type = B_2.type \text{ then } B_0.type := B_1.type \\
 & \text{otherwise } B_0.type := invalid \\
B_0 \to if(B_1, B_2, B_3) & \text{if } B_1.type \neq address \text{ then } I.type := invalid \\
 & \text{if } B_2.type = B_3.type \text{ then } I.type := B_2.type \\
 & \text{otherwise } I.type := invalid \\
B \to T & B.type := T.type \\
T \to a_0 \mid a_1 & T.type := address \\
T \to d_0 \mid d_1 \mid d_2 \mid d_3 & T.type := data
\end{array}
$$

This schema means that the parameters of the operators have the same type, except the first parameter of the *if* expression, which is always an address. If this condition holds then this type is also used as the type for the whole expression, otherwise the type is invalid. The semantic constraint can be introduced by setting the forced value of S to *data*, and using the following distribution functions or distribution sets to define grammar G_{6ms}:

$$
\begin{array}{rl}
S \to B & f(t) = (t) \\
B \to not(B) & f(t) = (t) \\
B \to and(B, B) \mid or(B, B) & f(t) = (t, t) \\
B \to if(B, B, B) & f(t) = (address, t, t) \\
B \to T & f(t) = (t) \\
T \to a_0 \mid a_1 & D(address) = \{()\},\ D(data) = D(invalid) = \emptyset \\
T \to d_0 \mid d_1 \mid d_2 \mid d_3 & D(data) = \{()\},\ D(address) = D(invalid) = \emptyset
\end{array}
$$

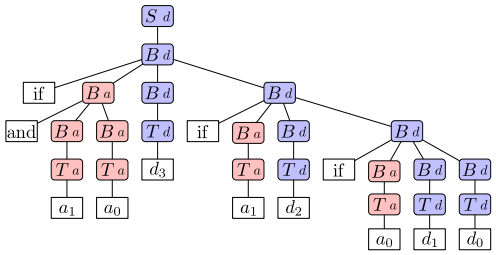
The results from a single run are shown in Figure 6.10. It shows that the course of the process is similar to the previous ones, but the size of the best individual is smaller. The final result of



(a) fitness values (b) best individual dimensions

Figure 6.10: 6-Multiplexer single test results with G_{6ms}

the single run can be seen in Figure 6.11, where both the derivation tree and the phenotype are shown. The colors in the derivation tree represent the value of forced synthesized attribute *type*.



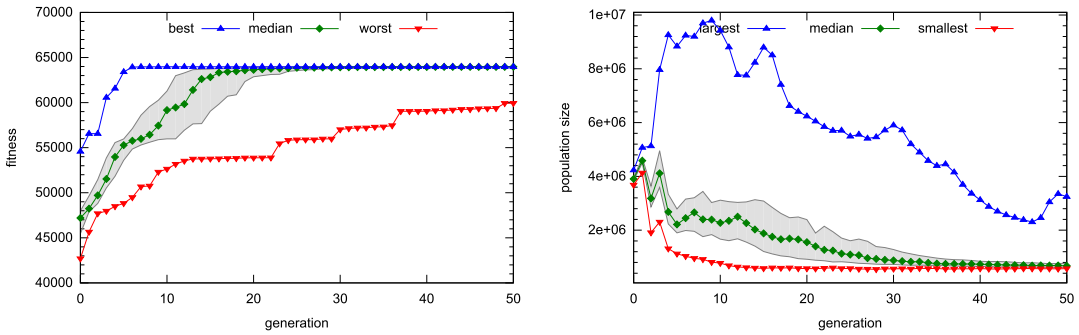
(a) derivation tree

```
SIX-MULTIPLEXER(a, d)
1  if a1 ∧ a0
2     then return d3
3     else if a1
4             then return d2
5             else if a0
6                     then return d1
7                     else return d0
```

(b) the solution

Figure 6.11: Best individual found by a single DTGP run with G_{6ms}

The results for 100 independent runs are plotted in Figure 6.12. They are similar to the results with G_{6mi}: the best run reaches a 64-bit match within a few steps, and most of the runs achieve it within 25 steps.



(a) best fitness

(b) population size

Figure 6.12: 6-Multiplexer test results for 100 independent runs with G_{6ms}

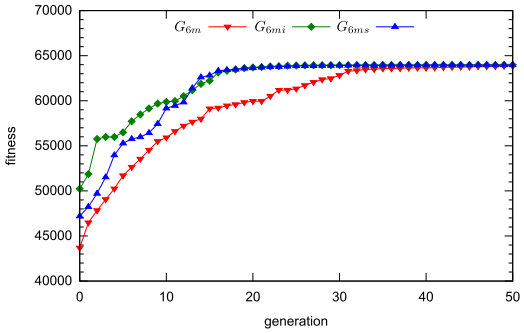
Note that the issue of failed derivations as mentioned in Section 5.2 does not appear with this grammar. The only case a derivation could fail due to an empty distribution set is when a rule for T has to be found. However, if the forced value of *type* is *address* or *data* there will be applicable rules, so the derivation can be finished successfully.

6.1.4 Multiplexer summary

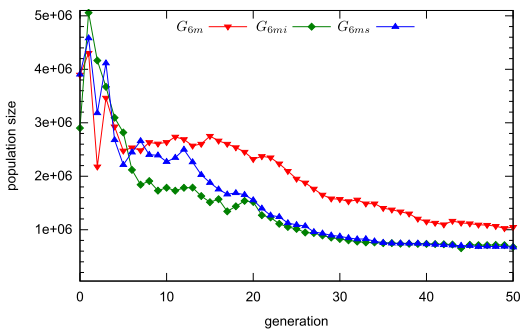
In this section three possibilities were presented for solving the 6-Multiplexer problem using derivation tree based genetic programming. The mean fitness values and mean population sizes during the process are plotted in Figure 6.13 and the final results of the three approaches are compared in Figure 6.14. The success rate is very similar for G_{6mi} and G_{6ms}, 100% is reached around step 50. Grammar G_{6m} yields worse results, but by step 100 a success rate of 97% is reached.

The results show that using a biased grammar like G_{6mi} better results can be achieved. However, even with a generic grammar like G_{6m}, with no a-priory knowledge, DTGP can solve the 6-Multiplexer problem, and the results are even better when semantic constraints are used.

If these results are compared to the ones in [49], one can conclude that using the given parameters, DTGP outperforms CFG-GP, which only achieves 37% and 88% probability of success with G_{6m} and G_{6mi} respectively. However, one must note that CFG-GP used a population size of 50, while DTGP used 100 individuals. Furthermore, CFG-GP uses post-selection to filter out too

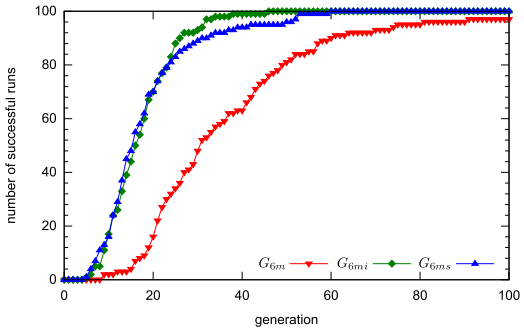


(a) mean fitness

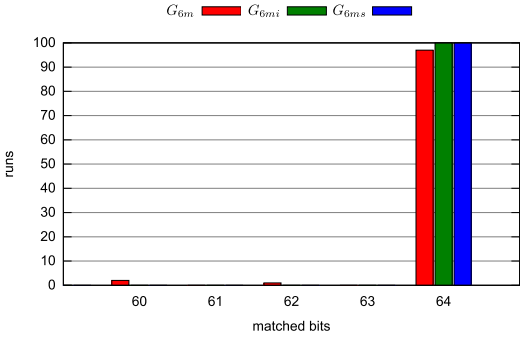


(b) mean population size

Figure 6.13: 6-Multiplexer average population characteristics with various grammars



(a) success rate



(b) found matches

Figure 6.14: 6-Multiplexer results with various grammars

large individuals, whereas DTGP has only a limit on the generated individuals, which is forced by the RTG algorithm, and later the size is not bounded. Instead, the size of the individual is used in the fitness value to drive the process toward smaller trees. This allows larger and thus more diverse trees, and with the logarithmic operators DTGP can handle these within an acceptable time.

6.2 Traveling salesman problem

The *traveling salesman problem* (TSP) is a well-known NP-complete optimization problem. [9] The task is to find the shortest tour through a given set of cities. There are several algorithms designed to solve the TSP [4], but it is also used as a benchmark for generic optimization algorithms. Finding a solution for the TSP is also a challenge in the field of genetic algorithms, and during the last decades several solutions have been proposed. [24]

The TSP can easily be reformulated in terms of graphs. Given a graph, that is a network of cities connected with roads $N = (C, R)$, where $C = \{c_1, c_2, \ldots, c_n\}$ is the finite set of cities, and $R \subseteq C \times C$ is the finite set of roads. The roads can be denoted by indices, as $r_1, r_2, \ldots$ or they can be specified by the starting and ending cities as $r_{i,j} = (c_i, c_j)$ pairs. There is also a function defining the weight of each road representing the length or cost:

$$w : R \to \mathbb{R}$$

The goal is to find a circle in the graph with minimal cost such that the circle visits all cities.

It is often required that the cities are visited exactly once. It is also common to assume a total graph, with cost set to infinity for edges between cities that are not connected by a road. In this section, it will be allowed to visit the cities more than once, and also a total graph is assumed. Furthermore, it will also be allowed not to visit all the cities, but for each missed city a penalty will be given. Note that it is possible to define the TSP such that multiple roads are defined between two cities, but it will not be used here. Furthermore, a road from a city to itself will be allowed, meaning $R = C \times C$. In the following sections the number of cities will be denoted by n, thus the number of roads is n^2.

There are two ways to define the candidate solutions. They can be described as sequences of $k \in \mathbb{N}$ cities, or as sequences of roads connecting $l \in \mathbb{N}$ cities:

$$S_C = \{c_{i_1} c_{i_2} \dots c_{i_k} \in C^* \mid \forall 1 \leq j \leq k-1 : (c_{i_j}, c_{i_{j+1}}) \in R\}$$

$$S_R = \{r_{i_1,i_2} r_{i_2,i_3} \dots r_{i_{l-1},i_l} \in R^*\}$$

In words, the solution is a sequence of cities such that there is a route between each adjacent city. In case of road sequences, the solution is a sequence of roads such that each road ends in the city where the adjacent road starts. In case the graph is total, any sequence of cities is allowed, because a road exists between every city.

6.2.1 Regular grammar for road sequence

One way to generate road sequences is to define a regular grammar with nonterminal symbols P_i representing a path starting at city c_i with $1 \leq i \leq n$, that is for each city. Such a path can be replaced by a road $r_{i,j}$ to city c_j and a path P_j, for any road starting from city c_i. That is

$$G_{tsp,r} = (\mathcal{N}_{tsp,r}, \Sigma_{tsp,r}, \mathcal{P}_{tsp,r}, S),$$

where $\mathcal{N}_{tsp,r} = \{P_i \mid 1 \leq i \leq n\} \cup \{S\}$, $\Sigma_{tsp,r} = \{r_{k,l} \mid 1 \leq k, l \leq n\}$ and $\mathcal{P}_{tsp,r}$ is defined as follows:

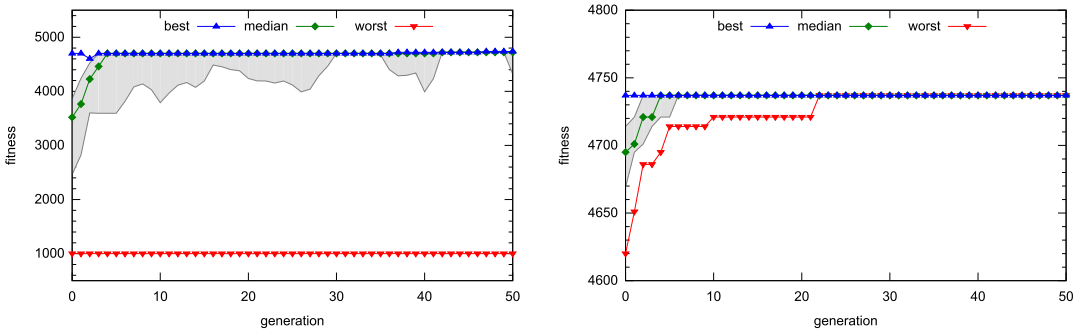
$$\begin{aligned} S &\to P_i && \forall 1 \leq i \leq n \\ P_i &\to r_{i,j} P_j && \forall 1 \leq i, j \leq n \\ P_i &\to r_{i,j} && \forall 1 \leq i, j \leq n \end{aligned}$$

For fitness calculation two parameters can be introduced. The cost can be computed in parameter *cost*, and the set of the visited cities can be stored in parameter *visited*. Given the weight function w, these parameters are calculated as follows:

$$\begin{aligned} S \to P_i \quad & S.cost := P_i.cost \\ & S.visited := P_i.visited \\ P_i \to r_{i,j} P_j \quad & P_i.cost := P_j.cost + w(r_{i,j}) \\ & P_i.visited := \{i\} \cup P_j.visited \\ P_i \to r_{i,j} \quad & P_i.cost := w(r_{i,j}) \\ & P_i.visited := \{i, j\} \end{aligned}$$

This setup was tested on a small example. Given the five Irish international airports: Dublin, Shannon, Cork, Knock and Waterford. Find the shortest circle to visit all of them, where the cost of a segment is the great circle distance between the two airports. Note that the grammar only generates paths with a randomly selected starting point. Therefore, an additional step from the last visited city to the first one has been added during the genotype-phenotype mapping.

The evolutionary algorithm used 1000 individuals and 100 independent runs were done. The fitness function was the number of visited cities multiplied by 1000, minus the cost of the circle. Since the grammar only calculates the cost for the path from the first city to the last, and not the circle, the cost for the road from the last city to the first has been added to the *cost* parameter calculated using the schema described above. The fitness values are plotted in Figure 6.15 and the optimal result found by the algorithm is shown in Figure 6.16.



(a) single run (b) multiple runs

Figure 6.15: Test results for the small TSP

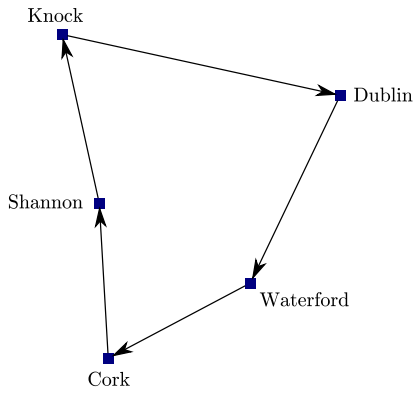
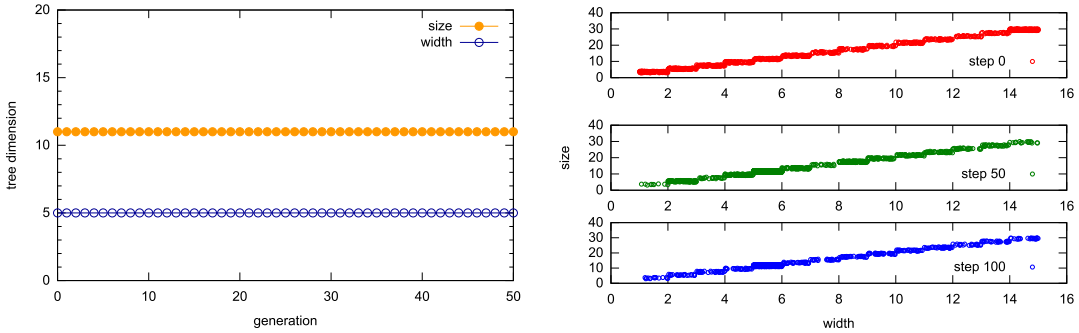


Figure 6.16: Solution found for the small TSP

Due to the definition of the rules, the size-width ratio converges to 2 as the size increases, because there are k leaves, k internal nodes labeled with P and the root node labeled with S. If the dimensional data is plotted it also shows this direct relation, as it can be seen in Figure 6.17. Note that for better visibility, random numbers from the interval $[0, 1)$ have been added to the width and to the height values, so that the dots representing the trees would not cover each other.

The 5 city TSP is a relatively simple problem, so it is no surprise that DTGP is able to find an optimum very quickly. However, the above mentioned grammar cannot be extended to larger problems in an efficient manner. Although the size of the nonterminal set is linear, that is $|\mathcal{N}_{tsp,r}| = \mathcal{O}(n)$, the size of terminal set and number of production rules are quadratic, that is $|\Sigma_{tsp,r}| = \mathcal{O}(n^2)$ and $|\mathcal{P}_{tsp,r}| = \mathcal{O}(n^2)$.

As discussed in Section 4.5.6, the success rate of pool crossover is inversely proportional to the size of the nonterminal set, and the operator cost of mutation is directly proportional to the number of rules.



(a) best individual dimensions

(b) size-width ratio

Figure 6.17: Sizes during the small TSP test

6.2.2 Regular grammar with parameters

To overcome the issue of increasing symbols and rules, one can store city indices as parameters and use semantic constraints as introduced in Section 5.2 to ensure a correctly described path. To achieve this, a parameter called *step* $\in \mathbb{N}^2$ is defined, which is a pair $[s_1, s_2]$ describing a path from city c_{s_1} to city c_{s_2}. Parameter *step* assigned to a nonterminal symbol will be interpreted as path between two cities, whereas if it is assigned to terminal symbol r it will mean a road between the two cities. Thus, the solutions for the TSP problem can be described by the following grammar:

$$G_{tsp,s} = (\{S, P\}, \{r\}, \mathcal{P}_{tsp,s}, S),$$

where $\mathcal{P}_{tsp,s}$ is defined as follows:

$$\begin{aligned} S &\to P \qquad f_{step}([s_1, s_2]) = ([i, \varnothing]), \text{ where } 1 \le i \le n \\ P &\to rP \qquad f_{step}([s_1, s_2]) = ([s_1, j], [j, \varnothing]), \text{ where } 1 \le j \le n \\ P &\to r \qquad f_{step}([s_1, s_2]) = ([s_1, k]), \text{ where } 1 \le k \le n \end{aligned}$$

Note that the second item of parameter *step* assigned to P is always $\varnothing$, and it is therefore ignored. It is consistent with the previous interpretation of P that it represents a path starting from a given city, but the endpoint is not specified. The first rule of the grammar finds a random starting point c_i. The second rule adds a road from the starting city to a randomly chosen next city c_j. The rest of the path must start at c_j. The last rule is used to stop the path after making a last step to c_k. Note that unlike in the previous examples, the parameter values are not constant for the terminal symbols, thus a distribution function is defined for the last rule as well.

To see how the second grammar works, and also to show how it differs from the first one, consider the following two derivation sequences for path $c_2c_3c_1c_5$, first using regular grammar $G_{tsp,r}$:

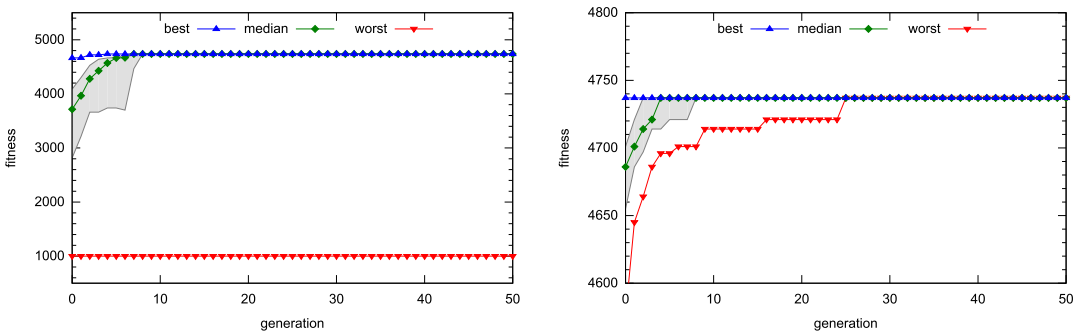
$$S \Rightarrow P_2 \Rightarrow r_{2,3}P_3 \Rightarrow r_{2,3}r_{3,1}P_1 \Rightarrow r_{2,3}r_{3,1}r_{1,5},$$

then using regular grammar $G_{tsp,s}$ with semantic constraint:

$$S \Rightarrow P([2, \varnothing]) \Rightarrow r([2, 3])P([3, \varnothing]) \Rightarrow r([2, 3])r([3, 1])P([1, \varnothing]) \Rightarrow r([2, 3])r([3, 1])r([1, 5])$$

The city codes are the same, and the calculation scheme seems to be the same as well, but in the first case 4 nonterminals (one for each city) and 3 terminals (one for each road) are used, whereas in the second case only 2 nonterminals (S, P) and a single terminal (r) is needed.

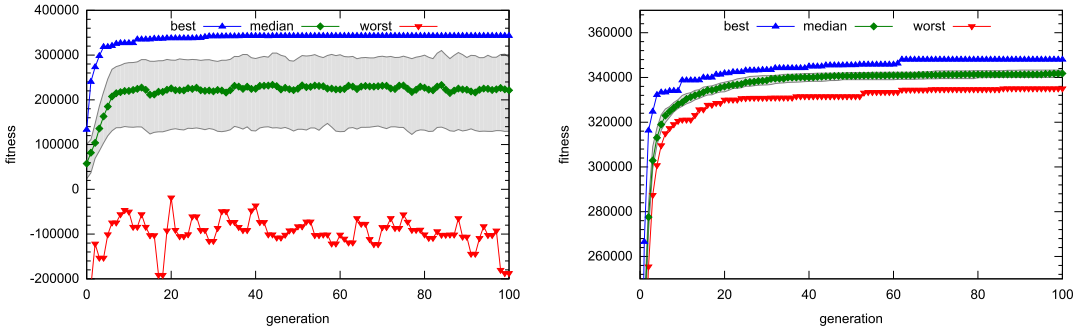
The 5 city TSP has been tested with this grammar as well. DTGP found the same optimal solution, although this time it took a bit longer as it can be seen in Figure 6.18.



(a) fitness values (b) best individual dimensions

Figure 6.18: Test results for the small TSP with parameterized regular grammar

The results show that grammar $G_{tsp,s}$ does not work as good for the 5 city problem as $G_{tsp,r}$, but it has a big advantage: it can be applied to larger problems. As an example, it has been tested on dj38, a problem containing 38 cities in Djibouti. This example has been downloaded from [1]. The parameters for DTGP were the same, but the fitness function has been slightly adjusted: each visited city increased the fitness by 10000 because the circles are longer, thus the costs are higher. The results are shown in Figure 6.19.



(a) single run (b) multiple runs

Figure 6.19: Test results for the large TSP with parameterized regular grammar

If only the fitness values are observed, the results look promising. However, if the path is plotted on the map, as shown in Figure 6.20, one can observe, that the result is far from the optimal route.

If the evolutionary process is observed closely and the individuals are examined, one can find that the degenerate derivation trees caused by the regular grammar are causing a problem. When a subtree is selected as a target for an evolutionary operator, it means that a suffix of the road sequence is selected. Thus the operators can only change the suffixes of the sequence, and it is not possible to replace some internal sections. To solve this issue, one can try to use a different grammar, that has rules with more than one nonterminal symbol on the right-hand side.

The result of the test shows another place for potential improvement. To ensure that all cities are visited, the fitness is increased for each visited city. However, it is not a guarantee. Therefore, one can try to introduce a semantic constraint based on synthesized attribute *visited*.

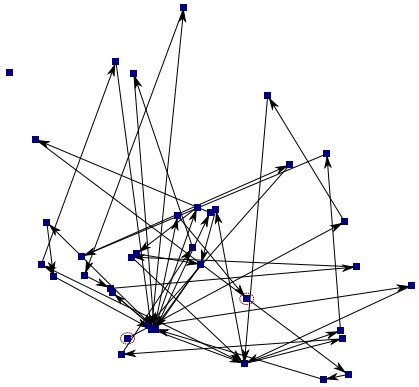


Figure 6.20: Large TSP result using regular grammar with parameters

6.2.3 Context-free grammar with semantic constraint

The idea of denoting a path by nonterminal P as mentioned in the previous section can be extended such that not only the starting city but also the ending city is defined. This can be achieved by making use of the second item of the *step* parameter, which was always set to $\varnothing$ for P. If it is used to contain the ending city instead, the set of road sequences can be described by the following context-free grammar:

$$G_{tsp,c} = \{\{S, P\}, \{r\}, \mathcal{P}_{tsp,c}, S\},$$

where the set of rules $\mathcal{P}_{tsp,c}$ is defined as follows:

$$\begin{aligned} S \to P \quad & \hat{f}_{step}([s_1, s_2]) = ([i, i]), \text{ where } 1 \leq i \leq n \\ P \to PP \quad & \hat{f}_{step}([s_1, s_2]) = ([s_1, j], [j, s_2]), \text{ where } 1 \leq j \leq n \\ P \to r \quad & \hat{f}_{step}([s_1, s_2]) = ([s_1, s_2]) \end{aligned}$$

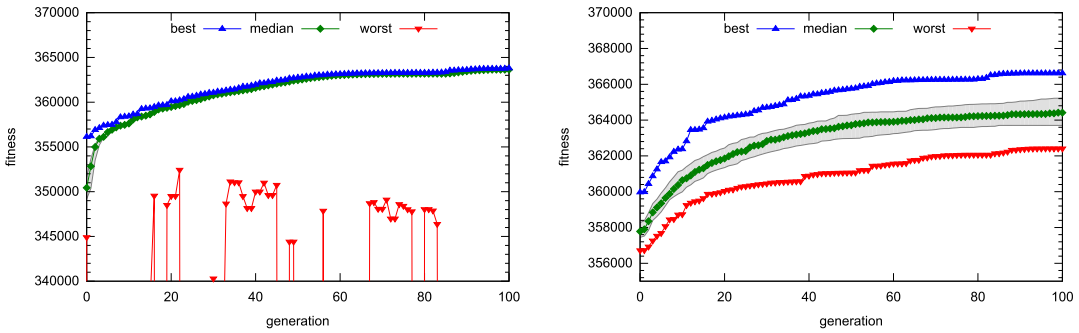
To use parameter *visited* as a forced synthesized attribute, the following distribution sets can be defined with the help of parameter *step*:

$$\begin{aligned} S \to P \quad & D_{visited}(v) = \{(v \cup \{i\})\} \\ P \to PP \quad & D_{visited}(v) = \{(v_1, v_2) \mid \{s_1, j\} \subseteq v_1 \wedge \{j, s_2\} \subseteq v_2 \wedge v_1 \cup v_2 = v\} \\ P \to r \quad & D_{visited}(v) = \{()\}, \text{ if } v \cap \{s_1, s_2\} = \emptyset, \text{ otherwise } D_{visited}(v) = \emptyset \end{aligned}$$

Note that for the second rule s_1 is included in v_1 and similarly s_2 is included in v_2, because these cities will always be covered by the first and second path respectively, thus including them will not add an extra constraint on the derivation. On the other hand, adding s_1 to v_1 means that it does not have to be added to v_2. The test results using this grammar can be seen in Figure 6.21, and the best result found by the algorithm is shown in Figure 6.22.

6.2.4 TSP summary

The results of the various TSP tests show that it is possible to construct a DTGP algorithm for optimizing a TSP route, although the found solution is not always optimal. That shows that even though DTGP is highly-configurable and can be applied to many kinds of problems, it is not well-suited for these kinds of tasks. 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. Also note that there are very few examples in the literature of canonical or grammar guided genetic programming



(a) single run (b) multiple runs

Figure 6.21: Large TSP results with context-free grammar

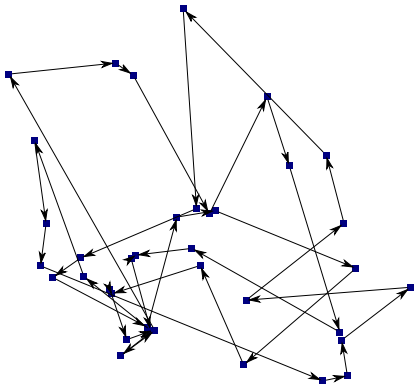


Figure 6.22: Result for the large TSP with context-free grammar

being applied to solve the TSP. Instead of trying to apply GGGP to the TSP, there were some attempts to use grammatical evolution to find ant colony optimization algorithms that can find the solutions. [13, 40]

6.3 Finite input response filters

A specialized version of DTGP has been used in a real-world application with the Fraunhofer Institute for Integrated Circuits to optimize *finite input response filter* (FIR) structures. [51, 50] FIR filters are commonly used in digital signal processing. A filter structure is primarily defined by its transfer function. During the design of FIR structures, this transfer function is given and then a circuit is constructed that describes the function. The usual components allowed in the circuit are shifters, adders, multipliers and delayers. The goal is to minimize the cost of the circuit by reducing the number of components, and preferring cheaper components (like shifters) over more expensive ones (like multipliers).

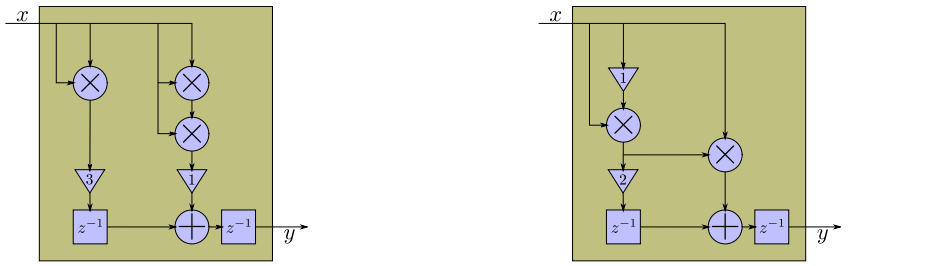
The transfer function can be described as a sum of unique polynomial component terms:

$$y = \sum_{k=0}^{n} \alpha_k \cdot 2^{\gamma_k} \cdot x^{\delta_k} \cdot z^{-\beta_k},$$

with $0 \leq \delta_k \leq P$, $0 \leq \beta_k \leq M$ and $n \leq \max\{M, P\}$, where P is called the degree of the structure and M is called the order of the structure. The set of transfer function descriptions is denoted by $\mathcal{F}$. A component term has three parts:

- Coefficient $\alpha_k \cdot 2^{\gamma_k}$ depends on the multiplier α_k and the shifting factor γ_k.
- Exponent δ_k.
- Representation of the delay $z^{-\beta_k}$ with β_k delay cycles.

The structures themselves are created from shifters, adders, multipliers and delayers. Two examples taken from [50] for implementing transfer function $2x^3z^{-1} + 8x^2z^{-2}$ are shown in Figure 6.23.



(a) $y = (x^2 \cdot 2^3 \cdot z^{-1} + x^2 \cdot x \cdot 2) \cdot z^{-1}$ (b) $y_1 := ((x \cdot 2) \cdot x)$; $y = (y_1 \cdot 2^2 \cdot z^{-1} + y_1 \cdot x) \cdot z^{-1}$

Figure 6.23: Examples for implementing $2x^3z^{-1} + 8x^2z^{-2}$

These structures can also be described using the following functions:

- $shift_k$: $\mathcal{F} \to \mathcal{F}$, where $shift_k(p) = 2^k \cdot p$
- add: $\mathcal{F}^2 \to \mathcal{F}$, where $add(p_1, p_2) = p_1 + p_2$
- mul: $\mathcal{F}^2 \to \mathcal{F}$, where $mul(p_1, p_2) = p_1 \cdot p_2$
- $delay_k$: $\mathcal{F} \to \mathcal{F}$, where $delay_k(p) = p \cdot z^{-k}$

The set of valid FIR structures can be described by context-free grammar G_{FIR} [51]:

$$G_{FIR} = \{\{S, E, N\}, \{shift, add, mul, delay, x, (,), comma\}, \mathcal{P}_{FIR}, S\},$$

where the set of rules $\mathcal{P}_{FIR}$ is the following:

$$\begin{array}{lcl} S & \to & E \\ E & \to & shift(E, N) \\ E & \to & add(E, E) \\ E & \to & mul(E, E) \\ E & \to & delay(E, N) \\ E & \to & x \\ N & \to & 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \end{array}$$

The polynomial represented by the structure can be calculated as a synthesized attribute *poly* using the calculation schema described above.

Using an extended version of DTGP, the EvolFIR system has been created to find optimal FIR structures for pre-defined transfer functions. [51] The EvolFIR system uses attribute grammars and slightly modified methods for tree generation and random node selection. Furthermore, it employs special derivation tree representations, because it was necessary to create redundancy-free structures. To make sure that the described transfer function is the same as the one defined

in advance, EvolFIR applies a decomposition operator whenever a rewriting rule is applied, so that the composition of the polynomials in the subtrees give the required polynomial.

It should be noted that while EvolFIR used *poly* as inherited attribute and decomposition functions, it is also possible to use *poly* as forced synthesized attribute and define the appropriate distribution sets and distribution functions.

6.4 Summary

In this chapter three examples have been presented to demonstrate how DTGP can be applied to various problems.

The 6-Multiplexer problem shows that DTGP can create simple program structures. Furthermore grammar G_{6ms} demonstrated how forced synthesized attributes can be used to ensure type safety. In this example it means separating address and data.

The traveling salesman problem is a good example for problems that do not have strict syntactical structures. DTGP can be applied to such problems as well, although with limited success.

Optimizing FIR structures is a real-world problem, where DTGP – with some modifications – can provide good results that are comparable to those received with other special FIR optimization tools.

7 Conclusions in English

In this 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 predefined context-free grammar to represent individuals, and applies genetic programming to 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 still more complex. However, in this 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.

One of the challenges in defining the operators was the *random node selection*. Defining a linear time selection is straightforward, but in this thesis a logarithmic time algorithm was presented that makes use of additional parameters stored at the nodes. The other challenge was to improve the success rate of the crossover operator. Due to the requirement that subtrees to be swapped must have the same label in their root nodes, either the success rate decreases or the time complexity increases. Therefore, in this thesis the classical crossover operator was replaced by a newly defined operator called *pool crossover*. Pool crossover has the same effect as classical GP crossover, but it has a logarithmic time complexity and in most cases practically 100% success rate.

How to use parameters to improve the algorithm was also shown. One important usage is for the random node selection mentioned above. Furthermore, as presented in this thesis, bottom-up parameters can 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 this 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 filtered out with the help of the fitness function.

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

The findings of this thesis can be summarized as follows:

Thesis I. Derivation tree based genetic programming, as defined in this 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 the 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 distribution sets, distribution functions and forced synthesized attributes, DTGP 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.

There are several areas that were out of the scope of this thesis and there are further possible ways to improve the algorithm. An interesting area would be to experiment with various parameters and use them for biasing the random node selection, the operators or the fitness evaluation.

This thesis did not examine the aspects of implementing DTGP. The algorithms needed for DTGP are described in this thesis, but of course there are other ways to implement the same functionality. There are ways to reduce the storage requirements as well. For example, it is possible to store identical subtrees only once. Furthermore, offsprings created from the same parents tend to have large portions that are identical. Thus, it might be interesting to examine the possibility to store the original trees once and then only the changes.

An advantage of population based optimization methods, like evolutionary algorithms, is the possibility of easy parallelization. Using current trends of multi-core processors and general purpose computing on graphics processing units (GPGPU) it is possible to speed up the algorithm, although implementing a parallel version of certain components, like pool crossover, is not a trivial task.

Since the concept of derivation trees is defined not only for context-free grammars, DTGP might be adapted to other systems as well, for example to context-free hyperedge replacement grammars.

8

Conclusions in Hungarian

A dolgozatban egy új *nyelvtan vezérelt genetikus programozási* (GGGP) eljárás, a *derivációs fa alapú genetikus programozás* (DTGP) került definiálásra és kiértékelésre. Ez a módszer egy előre megadott környezetfüggetlen nyelvtan feletti derivációs fákat használja az egyedek reprezentálására, és ezekre a fákra alkalmaz genetikus programozást. Ennélfogva a *derivációs fa alapú GGGP*-k kategóriájába sorolható, és más nyelvtan vezérelt eljárásokhoz hasonlóan garantálni tudja, hogy az előállított egyedek az adott nyelvtan szempontjából mindig *szintaktikailag helyesek*.

A lineáris GGGP megközelítésekkel összehasonlítva a DTGP adattípusa nagyobb, bár legtöbbször nem aszimptotikusan, de mindenképp összetettebb. Ez a dolgozat bemutatta, hogyan lehet az evolúciós operátorokat helyesen és hatékonyan definiálni úgy, hogy ne csak a derivációs fák legyenek helyesek, hanem az időbonyolultság is logaritmikus maradjon az esetek többségében.

Az egyik kihívás az operátorok definiálásánál a *véletlen csúcskiválasztás*. Egy lineáris idejű kiválasztás egyszerűen definiálható, de ebben a dolgozatban egy olyan logaritmikus idejű algoritmust mutattunk be, amely a csúcsokban tárolt paramétereket hasznosítja. A másik kihívás a keresztezés operátor hatékonyságának növelése. Mivel a megcserélendő részfák gyökereinek azonos címkével kell rendelkezniük, ezért vagy a hatékonyság foka csökken, vagy a futási idő nő. A dolgozatban a klasszikus keresztezés operátort ezért egy újonnan definiált operátor, a *készlet keresztezés* helyettesíti. Ennek az operátornak ugyanaz az eredménye, mint a klasszikus GP keresztezés operátornak, logaritmikus idejű, és legtöbb esetben gyakorlatilag 100%-os hatékonyságú.

Azt is bemutattuk, hogyan lehet paramétereket alkalmazni az algoritmus továbbfejlesztésére. Ennek egy fontos alkalmazása a fent említett véletlen csúcskiválasztás. A lentről-felfelé paraméterek arra is alkalmasak, hogy a fitness számításhoz használható információt tároljunk el. Némely esetben a fenotípus, vagy akár a fitness érték is kiszámítható paraméterként, és így a kiértékelés időigénye konstans, egy további, logaritmikus idejű művelettel a paraméterek frissítésére.

A lentről-felfelé paraméterek értékeit előre is megadhatjuk, mint *szemantikai korlátokat*. A disztribúciós halmazok és disztribúciós függvények használatával ezeket az értékeket fentről lefelé haladva lehet továbbadni a részfáknak a véletlen fa generálása során. Ezt az eljárást *szemantikailag korlátozott derivációnak* nevezzük. A szemantikai korlátok alkalmazása egy jelentős előrelépés a korábbi GGGP eljárásokhoz képest, mert eddig ilyen korlátozásokat csak a fitness függvénybe lehetett beépíteni. Ebben az esetben olyan egyedek is létrejöttek, amelyek nem elégítették ki a szemantikai korlátokat, és ezeket később a fitness érték alapján kellett kiszűrni.

Ebben a dolgozatban a DTGP módszer egy logikai regressziós problémán került részletes bemutatásra. Ezen kívül a DTGP-t teszteltük a 6-Multiplexer problémán, valamint bemutattuk, hogyan lehet alkalmazni az utazóügynök problémára. A módszer *FIR (finite input response) filterek* optimalizálására való gyakorlati alkalmazását szintén összefoglaltuk.

A dolgozat megállapításainak összefoglalása:

I. Tézis A derivációs fa alapú genetikus programozás, ahogy ebben a dolgozatban definiáltuk, egy specializált evolúciós algoritmus, ami különböző optimalizálási problémák megoldására alkalmazható a fekete doboz elv alkalmazásával, miközben az előállított egyedek szintaktikai helyessége is garantált.

- **a**. A jól definiált véletlen fa generátorral a DTGP érvényes derivációs fákat állít elő miközben a szükséges méretkorlátozást is betartja.
- **b**. Az operátorok alkalmazása előtt a DTGP logaritmikus időigénnyel ki tud választani egy véletlen csúcsot a derivációs fában, miközben biztosítja, hogy a kiválasztás valószínűsége minden csúcsra ugyanaz. Továbbá a kiválasztható csúcsok halmaza korlátozható, és szükség esetén nem uniform szelekciós súly is alkalmazható.
- **c**. A gyengén teljesítő standard keresztezés helyettesíthető a készlet keresztezéssel, ami ugyanolyan időigényű, de legtöbbször 100%-os hatékonysággal működik.

II. Tézis A derivációs fák kibővített adattípusának köszönhetően, megfelelően definiált paramétereket alkalmazva, a DTGP viselkedése alakítható, és az algoritmus továbbfejleszthető.

- **a**. Megfelelő információ csúcsokban történő eltárolásával bizonyos esetekben a fitness függvény konstans időben kiértékelhető. A paraméterek frissítése további számítást igényel az operátorok alkalmazása után, de ez nem növeli a teljes időbonyolultságot.
- **b**. Paraméterek használatával a véletlen csúcskiválasztás, és ezáltal az evolúciós operátorok, valamint a fitness kiértékelés is befolyásolható.
- **c**. Megkötött szintetizált attribútumok segítségével szemantikus korlátok helyezhetők az algoritmusba.

III. Tézis A DTGP alkalmazható különféle problémákra, különösen akkor, ha a megoldásoknak olyan struktúrájuk van, amely egy környezetfüggetlen nyelvtannal leírható.

A módszer néhány aspektusa, illetve az algoritmus több lehetséges továbbfejlesztési iránya túlmutat a dolgozat keretein. Egy érdekes téma a különféle paraméterek vizsgálata, illetve ezek alkalmazása a véletlen csúcskiválasztás, az operátorok, vagy a fitness kiértékelés befolyásolására.

A dolgozat nem vizsgálta a DTGP implementálásának részleteit. A DTGP megvalósításához szükséges algoritmusokat bemutattuk, de természetesen a funkcionalitás más módon is implementálható. A tárigény csökkentésére is van lehetőség, például az azonos részfákat lehetne csak egy példányban tárolni. Ezen kívül, az azonos szülőktől leszármazott egyedek sokszor jelentős részben megegyeznek. Ezért érdekes lenne megvizsgálni azt a lehetőséget, hogy csak az eredeti fákat tároljuk el egészben, utána pedig csak a változtatásokat.

A populáció alapú optimalizálási eljárások, így az evolúciós algoritmusok egyik előnye, hogy könnyen párhuzamosíthatók. Napjaink trendjeit követve a többmagos processzorok és a GPGPU lehetőségeit kihasználva az algoritmus felgyorsítható, bár néhány komponens, mint például a készlet keresztezés párhuzamosított változatának megvalósítása nem triviális feladat.

Mivel a derivációs fák koncepciója nem csak a környezetfüggetlen nyelvtanokra létezik, a DTGP más rendszerekre is áttehető, például a környezetfüggetlen hiperél-átíró nyelvtanokra.

A Notations

A.1 Derivation trees

$A[T_1, T_2, \ldots, T_n]$	a derivation with a root node having label A, and children $T_1, T_2, \ldots, T_n$
$N[T_1, T_2, \ldots, T_n]$	a derivation with root node N, and children $T_1, T_2, \ldots, T_n$
$T[A]$	a derivation tree rooted with label A
$N.p$	parameter p of node N
$T.p$	property p of tree T, or parameter p at the root node of T
$\mathcal{T}(X)$	the set of all derivation trees rooted with label $x \in X$
$\mathcal{T}(G)$	the set of all derivation trees over grammar G

$N.parent$	reference to the parent of node N
$N.children$	indexed reference to the children of node N
$N.label$	the label of node N
$N.size$	size of the subtree rooted at N, that is the sum of all nodes in the subtree
$N.nw$	node selection weight used for random node selection
$N.sw$	subtree weight, that is the sum of the node selection weigths for each node

A.2 Miscallenous

$Dom(f)$	the domain of function f. Outside of this set f is not defined.

List of Figures

List of Tables

List of Algorithms and Procedures

Index

Bibliography

[1] TSP library at Georgia Tech. `http://www.tsp.gatech.edu/`.

[2] E. Alba and J. M. Troya. A survey of parallel distributed genetic algorithms. *Complexity*, 4(4):31–52, Mar. 1999.

[3] 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.

[4] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton University Press, Princeton, NJ, USA, 2007.

[5] W. Banzhaf and W. B. Langdon. Some considerations on the reason for bloat. *Genetic Programming and Evolvable Machines*, 3(1):81–91, Mar. 2002.

[6] P. J. Bentley. *Generic Evolutionary Design of Solid Objects using a Genetic Algorithm*. PhD thesis, University of Huddersfield, 1996.

[7] M. Brameier and W. Banzhaf. *Linear Genetic Programming*. Number XVI in Genetic and Evolutionary Computation. Springer, 2007.

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

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. The traveling-salesman problem. In *Introduction to Algorithms, Second Edition*, chapter 35.2. The MIT Press and McGraw-Hill Book Company, 2001.

[10] C. Darwin. *On the Origin of Species*. Murray, London, 1859.

[11] K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms*. Wiley-Interscience Series in Systems and Optimization. John Wiley & Sons, Chichester, 2001.

[12] P. Deransart, M. Jourdan, and B. Lorho. *Attribute grammars: definitions, systems and bibliography*. Springer-Verlag New York, Inc., New York, NY, USA, 1988.

[13] M. Dorigo and L. M. Gambardella. Ant colonies for the traveling salesman problem. *BioSystems*, 43:73–81, 1997.

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

[15] L. J. Eshelman and J. D. Schaffer. Real-coded genetic algorithms and interval-schemata. In L. D. Whitley, editor, *FOGA*, pages 187–202. Morgan Kaufmann, 1992.

[16] D. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3(5):493–530, 1989.

[17] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Reading, MA, 1989.

[18] 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.

[19] J. H. Holland. *Adaption of Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, Michigan, 1975.

[20] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Language, and Computation*. Addison–Wesley, Reading, MA, 1979.

[21] C. Jacob. *Principia Evolvica – Simulierte Evolution mit Mathematica*. Dpunkt Verlag, 1997.

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

[23] W. B. Langdon, T. Soule, R. Poli, and J. A. Foster. The evolution of size and shape. In L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press, Cambridge, MA, USA, June 1999.

[24] P. Larrañaga, C. M. H. Kuijpers, R. H. Murga, I. Inza, and S. Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13(2):129–170, Apr. 1999.

[25] S.-C. Lin, Punch, and Goodman. Coarse-grain parallel genetic algorithms: categorization and new approach. In *Proceedings of the 1994 6th IEEE Symposium on Parallel and Distributed Processing*, SPDP '94, pages 28–37, Washington, DC, USA, 1994. IEEE Computer Society.

[26] S. Luke. *Essentials of Metaheuristics*. Lulu, 2009. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[27] P. Marenbach, K. D. Betterhausen, and S. Freyer. Signal path oriented approach for generation of dynamic process models. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 327–332, Stanford University, CA, USA, 28–31July 1996. MIT Press.

[28] 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.

[29] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Artificial Intelligence. Springer-Verlag, 1992.

[30] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

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

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

[33] C. B. Pettey, M. R. Leuze, and J. J. Grefenstette. A parallel genetic algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pages 155–161, Hillsdale, NJ, USA, 1987. L. Erlbaum Associates Inc.

[34] 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).

[35] I. Rechenberg. *Evolutionsstrategien: Optimierung Technischer Systeme nach Prinzipen der Biologischen Evolution*. Fromman-Holzboog, Stuttgart, 1973.

[36] C. Ryan, J. Collins, and M. O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In W. Bhanzaf et al., editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 83–95, Paris, 14-15Apr. 1998. Springer Verlag.

[37] H.-P. Schwefel. *Numerische Optimierung von Computer-Modellen mittels der Evolutionstrategie*, volume 26 of *ISR*. Birkhaeuser, Basel/Stuttgart, 1977.

[38] T. Soule and J. A. Foster. Effects of code growth and parsimony pressure on populations in genetic programming. *Evolutionary Computation*, 6(4):293–309, Winter 1998.

[39] R. Tanese. Distributed genetic algorithms. In *Proceedings of the third international conference on Genetic algorithms*, pages 434–439, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[40] J. Tavares and F. B. Pereira. Automatic design of ant algorithms with grammatical evolution. In A. Moraglio, S. Silva, K. Krawiec, P. Machado, and C. Cotta, editors, *EuroGP*, volume 7244 of *Lecture Notes in Computer Science*, pages 206–217. Springer, 2012.

[41] A. Turing. Intelligent machinery. Technical report, National Physical Laboratory, Teddington, England, 1948.

[42] A. Turing. Intelligent machinery. In C. R. Evans and A. D. J. Robertson, editors, *Cybernetics: Key Papers*. University Park Press, 1968.

[43] R. Ványi. Object oriented design and implementation of a general evolutionary algorithm. In K. Deb, R. Poli, W. Banzhaf, H.-G. Beyer, E. Burke, P. Darwen, D. Dasgupta, D. Floreano, J. Foster, M. Harman, O. Holland, P. L. Lanzi, L. Spector, A. Tettamanzi, D. Thierens, and A. Tyrrell, editors, *Genetic and Evolutionary Computation – GECCO-2004, Part II*, volume 3103 of *Lecture Notes in Computer Science*, pages 1275–1286, Seattle, WA, USA, 26-30 jun 2004. Springer-Verlag.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] P. A. Whigham. A schema theorem for context-free grammars. In *1995 IEEE Conference on Evolutionary Computation*, volume 1, pages 178–181, Perth, Australia, 29Nov.- 1Dec. 1995. IEEE Press.

[49] P. A. Whigham. *Grammatical Bias for Evolutionary Learning*. PhD thesis, School of Computer Science, University College, University of New South Wales, Australian Defence Force Academy, Canberra, Australia, 14 Oct. 1996.

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

[51] 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.

[52] 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.