



BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

**IMPLEMENTATION OF GRAMMATICAL EVOLUTION
SYSTEM**

IMPLEMENTACE SYSTÉMU GRAMATICKÉ EVOLUCE

BACHELOR'S THESIS

BAKALÁŘSKÁ PRÁCE

AUTHOR

AUTOR PRÁCE

JAN SVOBODA

SUPERVISOR

VEDOUCÍ PRÁCE

Prof. Ing. LUKÁŠ SEKANINA, Ph.D.

BRNO 2017

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů

Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Svoboda Jan**

Obor: Informační technologie

Téma: **Implementace systému gramatické evoluce**
Implementation of Grammatical Evolution System

Kategorie: Umělá inteligence

Pokyny:

1. Seznamte se s problematikou gramatické evoluce.
2. Navrhněte metodu založenou na gramatické evoluci, která umožní generovat, popř. optimalizovat jednoduchý kód.
3. Vytvořenou metodu implementujte jako knihovnu, která může být využita v dalších projektech. Zaměřte se na optimalizaci implementace z pohledu výkonnosti.
4. Knihovnu experimentálně ověřte na zadaných problémech, zejména v úloze symbolické regrese. Proveďte rovněž základní statistické vyhodnocení experimentů.
5. Zhodnoťte dosažené výsledky.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 a 2 zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese

<http://www.fit.vutbr.cz/info/szz/>

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Sekanina Lukáš, prof. Ing., Ph.D., UPSY FIT VUT**

Datum zadání: 1. listopadu 2016

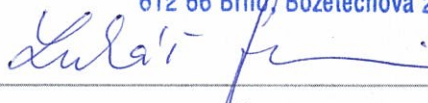
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta informačních technologií

Ústav počítačových systémů a sítí

612 66 Brno, Božetěchova 2



prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstract

Grammatical evolution is a relatively new approach to genetic programming that can automatically create solutions to various problems in an arbitrary programming language. This thesis summarizes the principles and algorithms of grammatical evolution and overviews the existing systems. Accompanying the thesis is a software called Gram – a new library offering high performance and applying the best programming principles such as modular code and automated testing. It has been compared to the best-performing available solution and showed over 30% improvement in execution time. Gram has also been successfully used to automate test-driven development, a technique commonly used to create software with automated tests. The thesis and the software project provide a solid ground for further research and allow for the application of grammatical evolution in new areas.

Abstrakt

Gramatická evoluce je relativně nový přístup ke genetickému programování, který dokáže automatizovaně řešit různé problémy vytvářením programů v libovolném programovacím jazyce. Tato práce shrnuje principy a algoritmy gramatické evoluce a poskytuje přehled o existujících systémech. Byla vytvořena nová knihovna Gram, která nabízí vysoký výkon a dodržuje dobré programátorské zvyklosti, jakými jsou modulárnost a automatické testování. Porovnání tohoto systému s nejvýkonnějším dostupným řešením ukázalo zlepšení v době výpočtu překračující 30 %. Gram byl také úspěšně použit pro automatizaci testy řízeného vývoje, techniky běžně používané při vytváření softwaru s automatizovanými testy. Tato práce a doplňující softwarový projekt tedy poskytují solidní základ pro další výzkum a umožňují využití gramatické evoluce v nových oblastech.

Keywords

Grammatical Evolution, Genetic Programming, Evolutionary Algorithms, Artificial Intelligence, Formal Languages.

Klíčová slova

Gramatická evoluce, genetické programování, evoluční algoritmy, umělá inteligence, formální jazyky.

Reference

SVOBODA, Jan. *Implementation of Grammatical Evolution System*. Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Sekanina Lukáš.

Implementation of Grammatical Evolution System

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Prof. Ing. Lukáš Sekanina, Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Jan Svoboda
May 17, 2017

Acknowledgements

I would like to thank my supervisor Prof. Lukáš Sekanina for his advice, motivation and friendly approach.

Contents

1	Introduction	3
2	Introduction to Grammatical Evolution	4
2.1	Genetic Programming	4
2.1.1	Initialization	5
2.1.2	Fitness Evaluation	5
2.1.3	Selection	6
2.1.4	Genetic Operators	7
2.2	Grammatical Evolution	8
2.2.1	Genotype-phenotype Mapping	8
2.2.2	Genetic Operators	10
2.2.3	Unique Properties of Grammatical Evolution	10
3	Grammatical Evolution Implementations	12
3.1	GEVA	12
3.1.1	Goal Definition	12
3.1.2	Initialization	13
3.1.3	Selection	13
3.1.4	Crossover	14
3.1.5	Mutation	14
3.1.6	Other Aspects	14
3.2	AGE	14
3.2.1	Goal Definition	14
3.2.2	Initialization	14
3.2.3	Selection	15
3.2.4	Crossover	15
3.2.5	Mutation	15
3.2.6	Other Aspects	15
4	Gram: System Design	16
4.1	System Overview	16
4.2	Success Definition and Evolution Run	17
4.3	Initialization	18
4.4	Evaluation and Fitness Function	18
4.5	Reproduction	19
4.5.1	Selection	19
4.5.2	Crossover	19
4.5.3	Mutation	20

4.6	Other Tools	20
4.6.1	BNF parser	20
4.6.2	Logger	21
4.7	Automated Tests	21
4.8	Build System	22
5	Performance Tuning	23
5.1	Profiling	23
5.2	Implementation Details	24
5.3	Random Number Generators	24
5.4	Optimizing Mutation Operator	26
5.5	Fitness Caching	27
5.6	Using Multiple Cores	28
5.7	Profiling Optimized Grammatical Evolution System	29
6	Experiments	30
6.1	Symbolic Regression	30
6.2	Automating Test-Driven Development	32
6.2.1	Fitness Function	32
6.2.2	Experiment Setup	33
6.2.3	Results and Discussion	34
7	Conclusion	36
	Bibliography	38
	Appendices	40
A	Building Gram	41
B	Example Gram Project	42
C	Used PHP Grammar	44

Chapter 1

Introduction

Genetic algorithms (GA) have been successfully used for creating solutions to hard optimization and search problems since the 1970s. GA relies on operators inspired by the principles of genetics (such as mutation, crossover, and selection) to drive a set of initial candidate solutions to the desired state.

While genetic algorithms evolve fixed-length binary strings, a more recent approach called genetic programming (GP) operates on individuals in the form of computer programs. One of the most prominent pioneers of GP was John Koza, author of the first monograph dealing with Genetic Programming [6].

Later, many variations on genetic algorithms were proposed. One of them is grammatical evolution (GE), introduced in the late 1990s by researchers from University of Limerick. GE can create computer programs in arbitrary language thanks to its flexible representation of candidate solutions.

Although GE has been around for almost 20 years, there are not many open-source and well-performing libraries that would allow wide public to experiment with grammatical evolution and contribute to its advancement.

Therefore, the goal of this thesis is to create a library with a solid design and good performance that would make it possible for all developers and researchers to apply GE in new areas and to contribute their ideas to the open-source community.

The text of this thesis is divided into following chapters:

- Chapter 2 introduces genetic programming and grammatical evolution and summarizes the advantages of grammatical evolution.
- Chapter 3 provides an overview of existing GE systems, the algorithms they implement and summarizes their advantages and disadvantages.
- Chapter 4 describes the design of the proposed library called **Gram**.
- Chapter 5 explains performance aspects of grammatical evolution systems.
- Chapter 6 compares Gram with available solutions and presents an application of grammatical evolution in real-world software development.
- Chapter 7 discusses the results of the thesis and suggests next directions of research.

Chapter 2

Introduction to Grammatical Evolution

In this chapter, we will introduce the principles of genetic programming and grammatical evolution.

- In Section 2.1, we will review general mechanisms of genetic programming.
- In Section 2.2, we will describe unique aspects of grammatical evolution.

2.1 Genetic Programming

Genetic programming (GP) is an approach to automatic generation of computer programs inspired by the evolutionary process and genetic mechanisms.

It is an iterative algorithm which first generates a *population* (i.e. a set) of candidate solutions. GP then repeatedly performs transformations of the candidate solutions, which often results in a better set of programs. The process is repeated until it finds a viable solution or available time is spent. Visualization of this process can be seen in Figure 2.1.

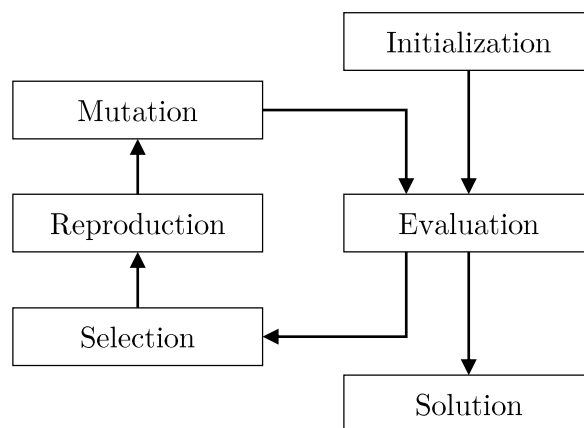


Figure 2.1: Algorithm of genetic programming

GP systems usually work with a *tree representation* of the program where the nodes correspond to non-terminals (control structures, functions or operators) and the leaf nodes

represent terminals of the programming language (variable names, keywords, parenthesis, etc.). See Figure 2.2 for an example of a GP individual.

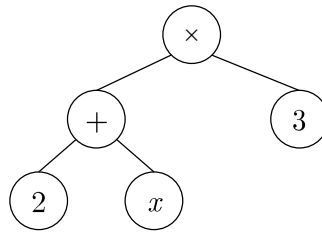


Figure 2.2: Tree representing the expression $(2 + x) \times 3$

There are other versions of GP, for example *Linear GP* or *Cartesian GP*, but we will not deal with them in this thesis.

2.1.1 Initialization

As in other evolutionary algorithms, the initial population in GP is typically generated randomly. The simplest methods, full and grow, also impose a limitation on the tree depth.

In the *full method*, nodes are randomly chosen from the set of non-terminals of the programming language until it reaches the maximum depth. Beyond that depth, it chooses only from the set of terminals.

The *grow method* is very similar, except that before reaching the maximum depth, it is allowed to choose from the whole set of language symbols – both non-terminals and terminals. This approach results in trees of various sizes and shapes, whereas the full method creates trees that have only branches of the same length.

Neither the grow nor full method generate trees of diverse enough sizes and shapes. The lack of diversity led to the creation of their combination referred to as the *ramped half-and-half method* [14]. This approach generates half of the population using the grow method and the second half using the full method. Additionally, the tree depth limit is not set to a single fixed number. Instead, it is randomly chosen from a range of allowed depths, resulting in trees of a wide variety of shapes and sizes.

2.1.2 Fitness Evaluation

At this stage, the algorithm does not have any way of knowing which of the generated programs are good at solving the defined problem. This is the purpose of a *fitness function*. It represents the goal of the evolution. In a single-objective scenario, the fitness function f is defined as:

$$\begin{aligned}
 f : S &\rightarrow \mathbb{R} && \text{where } S \text{ is a set of all programs and} \\
 f(s) &&& \text{where } s \in S \text{ is a score of individual program } s.
 \end{aligned}$$

To determine the fitness of individuals, GP, unlike other evolutionary algorithms, evaluates the individuals. The evaluation usually involves recursively walking through the tree representing an individual and interpreting subtrees along the way.

The individuals' fitness score a value that can be based on a number of different factors or their combination, for example:

- the difference between correct output and the actual output;
- the amount of time to bring the environment to the desired state;
- the accuracy of a process (for example of pattern recognition);
- or the compliance structure with specified design criteria.

In some cases, the objective may be comprised of multiple goals, often being in opposition to each other. The goal of the GP system is to find the best trade-off of those aspects.

2.1.3 Selection

Similarly to other evolutionary algorithms, genetic operators are applied with higher probability to individuals with better fitness. This means that high-scoring individuals have a higher chance of having more children. The two most common selection methods of selection are *tournament selection* and *roulette-wheel selection*.

Tournament Selection

In tournament selection, k individuals are randomly picked from the entire population. The algorithm then compares their fitness scores and chooses the best one which is then moved to the set of parents. This step is repeated m times, where m is the number of parents.

What *best* means depends on the particular fitness function and, therefore, should be configurable for the tournament. In cases where the fitness values represent the difference between correct and actual output, lower values are preferred. On the other hand, higher values are considered better in situations where the fitness value corresponds to individual's accuracy.

Note that tournament selection does not take into account the relative differences between fitness values. The sole goal of a fitness function in combination with the tournament selection is to determine if one individual is better than other, not by how much.

Roulette-wheel Selection

Roulette-wheel selection places the individuals on an imaginary roulette. The size of the field for each individual – and consequently the probability of selection $p(i)$ – is proportional to its fitness value:

$$p(i) = \frac{f_i}{\sum_{j=1}^n f_j} \quad \text{where } n \text{ is the size of population}$$

The selection can be imagined similar to spinning the roulette in a casino.

Unlike tournament selection, in the roulette-wheel method, the fitness has to be a positive number, and higher values are automatically considered better. Moreover, the relative magnitude of fitness plays an important role, as it directly affects the probability of selection. To accommodate for that, the fitness can undergo a process called the *fitness scaling*.

There are a number of different ways how one can scale the fitness and they are chosen with regard to the problem at hand and to the used fitness function. The goal is to ensure that fitter solutions have a higher probability of selection while worse individuals still have a non-zero probability they will be chosen for reproduction.

2.1.4 Genetic Operators

Genetic operators are used to alter the genetic structure of individuals chosen by the selection process. Two most commonly used operators are *crossover* and *mutation*.

Crossover

Crossover in genetic programming combines elements of two tree structures. It chooses a crossover point (a node) in both trees and works with subtrees of the two chosen nodes, as illustrated by Figure 2.3.

The crossover points are usually not chosen with an uniform probability. Koza suggested the widely used approach of choosing non-terminal nodes 90 % of the time and leaves in the remaining 10 % [6], which prevents the operator to work with too insignificant parts of the parental trees.

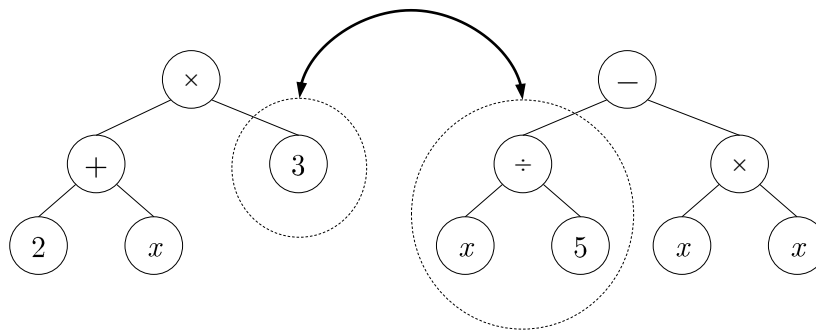


Figure 2.3: Parents in crossover and the selected subtrees

The two subtrees are then swapped between the parents, causing the tree of the first parent contain a part of the second parent's tree and vice versa. (see Figure 2.4).

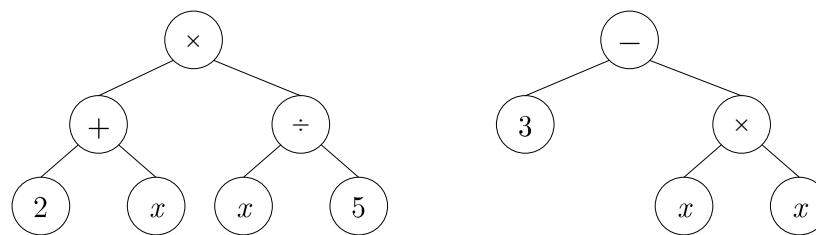


Figure 2.4: Offsprings created by swapping parent subtrees

The newly created offspring trees are placed in the next population of candidate solutions. This process exchanges genetic material of two viable solutions, which can result in an even better solution.

Mutation

The most common type of mutation in genetic programming works by randomly selecting a mutation point in a tree and replacing the whole subtree by a new, randomly generated tree.

Another form of mutation is a point mutation which chooses a random node in a tree and replaces the symbol it contains by a random symbol with the same arity. For example, an addition symbol can be replaced with multiplication operator, as both of them are binary and operate on real numbers.

2.2 Grammatical Evolution

In genetic programming, the evolved programs are represented as syntax trees, and the genetic operators work directly with that structure. There is no clear distinction between genotype and phenotype of the individual, as observed in nature. In biology, genotype is the actual genetic information. Phenotype include all observable characteristics of an individual.

The implication is that the genetic operators are closely tied to the chosen program representation, meaning they cannot be reused for other representations or languages.

Grammatical evolution overcomes this limitation by introducing a clear distinction between genotype and phenotype. The genetic operators work with the genotype, represented as a linear (binary) string, and the phenotype is derived from it by a mapping process [13]. The target language is specified by a formal grammar, usually in Backus-Naur form.

2.2.1 Genotype-phenotype Mapping

The mapping process starts by placing the start symbol of the grammar at the beginning of new phenotype. The given genotype – a binary string – is converted to a vector of integers, corresponding to biological *codons* that determine which amino acid will be used during protein synthesis.

The integers are one by one used to choose production rules of non-terminals in the phenotype. The rules are indexed starting from zero and the rule on index r is chosen:

$$r = c \bmod R$$

where c is the codon (integer) value from genotype and R is the total number of rules for the current non-terminal.

The non-terminal is replaced by symbols of the chosen rule. This process is repeated until the phenotype contains only terminals of the grammar. If the algorithm reaches the end of the genotype while the phenotype still contains non-terminals, the so-called wrapping event occurs. The genotype is simply read again from the beginning, and the mapping continues.

To prevent endless loops, grammatical evolution systems usually impose a limit on the number of wrapping events. If the mapping process exceeds this limit, it assigns a bad fitness score to the individual, effectively eliminating it from the reproduction process.

Example

We provide an example of a complete mapping process that maps genotype from Figure 2.5 to a phenotype with the use of grammar that can be seen in Listing 2.1. The grammar represents a formal language that is able to express simple mathematical expressions with one variable x , a constant and three binary operators.

genotype	4	1	0	5	3	1
----------	---	---	---	---	---	---

Figure 2.5: Grammatical evolution genotype

Listing 2.1: Example of a BNF grammar

A)	<code><expr></code>	<code>::= (<expr> <op> <expr>)</code>	<code>(0)</code>
		<code> <var></code>	<code>(1)</code>
B)	<code><op></code>	<code>::= +</code>	<code>(0)</code>
		<code> -</code>	<code>(1)</code>
		<code> *</code>	<code>(2)</code>
C)	<code><var></code>	<code>::= x</code>	<code>(0)</code>
		<code> 1</code>	<code>(1)</code>

First, the phenotype contains the start symbol of the grammar. In BNF, the first rule is automatically considered to be the start rule:

`<expr>`

and the first genotype value is used to choose one of two production rules of the start symbol: $r = 4 \bmod 2 = 0$. The start rule is therefore replaced by the rule with number 0:

`(<expr> <op> <expr>)`

In the next step, the second codon with value 1 is used to map the first non-terminal in the phenotype: `<expr>`. Index of the production rule is again calculated using the modulo operator: $r = 1 \bmod 2 = 1$ and the rule on index 1 containing non-terminal `<var>` replaces the `<expr>` non-terminal, resulting in the following phenotype:

`(<var> <op> <expr>)`

The third genotype value is used to expand the `<var>` non-terminal by choosing rule $r = 0 \bmod 2 = 0$ that contains a single terminal `x`:

`(x <op> <expr>)`

The `<op>` non-terminal is expanded with the use of the fourth genotype codon with value of 5: $r = 5 \bmod 3 = 2$ to terminal `*`:

`(x * <expr>)`

The last non-terminal in phenotype – `<expr>` – is expanded to `<var>`, as the fifth integer in genotype maps to rule number 1: $r = 3 \bmod 2 = 1$:

`(x * <var>)`

The `<var>` non-terminal is finally mapped to terminal `1` using the last genotype codon with value 1: $r = 1 \bmod 2 = 1$. The result of the mapping process is phenotype:

`(x * 1)`

2.2.2 Genetic Operators

The principle of genetic operators is very similar to operators in genetic programming. The main differences stem from the different representation of an individual.

Crossover

Crossover in grammatical evolution is fairly straightforward compared to the crossover used in genetic programming thanks to the use of linear integer vector. When two parental genotypes are to be combined, they are both split at a random position. The offspring is constructed by concatenating the first part of the first genotype with the second part of the second genotype, as illustrated by Figure 2.6.

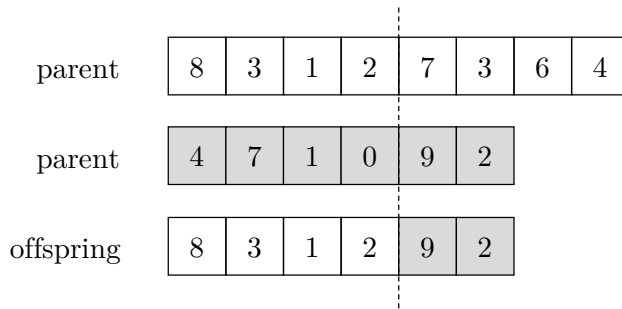


Figure 2.6: Crossover in grammatical evolution

There is a second type of crossover that is sometimes used in grammatical evolution: two-point crossover. The difference between simple one-point crossover is that the parental genotypes are split into three pieces (at two random points), and the offspring is created by swapping the middle part of the first parental genotype with the second part of the second parental genotype.

Mutation

The mutation operator iterates over the entire genotype and with defined probability changes codons to random values. An alternative version of mutation works directly with bits of the integer that represents a codon.

2.2.3 Unique Properties of Grammatical Evolution

The distinction of genotype and phenotype in grammatical evolution implies a separation of search and solution spaces, which can result in benefits such as unconstrained search of the genotype while ensuring the validity of the phenotype [1].

The wrapping mechanism in the genotype-phenotype mapping is similar to overlapping genes – a phenomenon observed in many viruses, bacteria, and mitochondria – that allows for a reuse of the same genetic material in different genes [12].

Since the modulo operation used by the mapping process transforms a large range of numbers into few rules of the grammar, genotype mutation often does not affect the individual's phenotype. This is similar to the neutral mutation and codon degeneracy

observed in biological systems, which may improve the genetic diversity as according to the neutral theory of evolution [5].

Chapter 3

Grammatical Evolution Implementations

Currently, there are only few publicly available implementations of grammatical evolution: GEVA, AGE, PonyGE, gramEvol, libGE, GERET, and PyNeurGen. Unfortunately, most of them are not maintained anymore.

In this chapter, we will focus on GEVA and AGE. GEVA implements most of the commonly used GE algorithms and can be used as a reference for development of new libraries. Notable features of AGE include flexible, modular architecture and great performance, which are also the goals of the project accompanying this thesis.

We will briefly describe the implemented algorithms and assess the strengths and weaknesses of both projects. That will be useful during development of a new GE library described in Chapter 4.

3.1 GEVA

GEVA is a framework for grammatical evolution developed by Natural Computing Research & Applications group in University College Dublin. The latest version – 2.0 – was released in June 2011.

The framework is implemented in Java and provides both command line interface and simple GUI. Its interesting features include modularization of the algorithms and the ability to combine those modules into pipelines [11].

3.1.1 Goal Definition

The pivotal class of the whole framework is `AbstractRun` which manages the main evolution loop. The loop runs until it reaches the maximum number of generations or until the method `foundOptimum` returns `true`. This method has by default access to high-level information about the fitness and length of individuals in the current population.

While users of the framework are free to create their own implementation of the method, the provided information may not be sufficient for more complex use-cases, where direct inspection of each individual might be necessary.

Moreover, the generation limit is hard-coded, meaning users who want to have a different limit than the default 100 generations must re-implement the whole `run` method, which is very complex because it orchestrates the whole evolution.

3.1.2 Initialization

GEVA offers the following initialization algorithms:

- random initialization,
- full method,
- grow method,
- ramped half-and-half method.

Although grammatical evolution uses a binary string to represent individuals, GEVA incorporates initialization strategies specific to the tree representation of individuals borrowed from genetic programming: the full and grow methods described in Section 2.1.

The initialization process constructs a derivation tree based on the BNF grammar along with the usual GE genotype. This allows to leverage advantages of both genetic programming (great population diversity thanks to a wide range of tree shapes) and grammatical evolution (simple representation).

The genotype is implemented as a variable length vector of 32-bit integers, the size of whose cannot be changed.

3.1.3 Selection

Supported selection methods include both methods described in Section 2.1.3: the tournament and roulette-wheel selection.

The problem is that developers of GEVA made the assumption that a lower fitness is always better and did not provide an easy way to change this.

That results in the necessity for ensuring that the better solutions are assigned a lower fitness, which can be confusing in scenarios where the fitness represents an accuracy, and better individuals are expected to have high fitness score. This can be done by inverting the score or changing its sign.

While this trick works for the tournament selection, it fails in the case of roulette-wheel, because it automatically rescales the fitness using the following equation:

$$f'_i = \frac{F - f_i}{F} \quad F = \sum_{j=1}^n f_j \quad \text{where } n \text{ is the population size}$$

The rescaling maps all fitness values to a very narrow interval, making the roulette-wheel selection unsuitable in some scenarios, as it can make very fit individuals indistinguishable from the very poor ones. Unfortunately, the scaling cannot be easily changed, so the user is forced to write the whole selection algorithm again when the need for a different scaling function arises.

In addition, GEVA also offers an option for the user to manually pick individual through the GUI. While it probably will not be used in real-world applications, it can be useful for education.

3.1.4 Crossover

Genetic operators are the most advanced aspect of the GEVA framework. Besides the one-point and two-point crossovers mentioned in Section 2.2.2, GEVA implements a subtree crossover.

Similar to the initialization methods, it slightly diverts from the original idea of grammatical evolution and builds derivation trees from the genotype of both selected individuals. The crossover then swaps randomly chosen subtrees of the same type between the two trees similarly to the GP crossover. The derivation trees are then serialized back to the original representation.

3.1.5 Mutation

Similarly to the crossover operator, the mutation operator is available in more sophisticated variants. In addition to the usual codon-level mutation, GEVA offers a structural mutation and subtree mutation that use the individuals' derivation tree to perform a codon change that does affect only a small part of the individual [2].

3.1.6 Other Aspects

The source code of GEVA is distributed in an archive that can be downloaded from the website of Natural Computing Research & Applications group. The downside of this approach is that nobody except the original authors can contribute new code and improve the framework.

The project contains automated unit tests, which ensure changes in source code do not introduce new bugs. Some classes contain `main` methods without any mention in the documentation. I suspect the authors wanted to have a way how to do quick tests, but it is not a common pattern and seems strange given the project already has proper unit tests.

3.2 AGE

AGE is a software project accompanying the bachelor's thesis of Adam Nohejl [10]. It is a C++ library focused on implementation of standard grammatical evolution algorithms, modularity, adequate documentation, reproducibility of results and performance.

The library is used in a small number of applications solving symbolic regression, Santa Fe ant trail, time tabling and few other problems.

3.2.1 Goal Definition

The evolutionary algorithm stops after a defined number of generations or when the best individual reaches a certain fitness level. In comparison with GEVA, this is somewhat limited and does not allow for more complex statistical analysis.

3.2.2 Initialization

AGE implements a random initializer which creates individuals of a variable size randomly chosen from a specified range.

The second available initialization method is ramped half-and-half with a number of customizable parameters such as range of maximum depths, the ratio of full and grow trees, whether all trees should be unique, etc.

3.2.3 Selection

Besides the usual tournament and roulette-wheel methods, AGE offers an interesting combination of the two. The algorithm is called „Wetzel ranking“ and works like the tournament method, but the selection of tournament competitors is done using a roulette-wheel selection. The thesis does not describe its advantages and does not offer any measurements.

The library offers a set of fitness scaling algorithms that are particularly useful in roulette-wheel selection. Unlike in GEVA, they are easy to change and are useful for the selection method.

3.2.4 Crossover

The crossover operator is present in the form of a simple single-point crossover. It can be configured to either change or keep the length of genotypes.

3.2.5 Mutation

AGE comes with three variants of mutation:

- bit mutation,
- slow bit mutation,
- codon mutation.

The first two algorithms work directly on bits representing the genotype. When we consider that the algorithm calls the random number generator for each bit in the genotype when deciding whether to mutate it or not, it becomes apparent that it is not optimal from the performance standpoint. Although the number of generator calls can be reduced by an optimization described in Chapter 5, the author does not provide an explanation why he decided to implement the bit-level mutation in addition to a more straightforward integer-level mutation.

3.2.6 Other Aspects

AGE is currently distributed as an archive through author’s personal website, which makes any contributions from other people impossible.

The library algorithms are not covered by any automated tests, making any change a potential source of bugs and therefore not suitable for real production applications.

The architecture is solid, with a clear separation of concerns and adequate level of abstraction. All fundamental parts of the grammatical evolution engine are hidden behind abstract classes (interfaces), making it fairly easy to create new implementations of involved algorithms.

The main advantage over GEVA is faster execution time. On symbolic regression, it performs over 10 time better than GEVA [10].

Chapter 4

Gram: System Design

The goal of the new grammatical evolution library **Gram** is to combine the best parts of both systems from previous chapter.

The main objective is to design a modular library that allows users to easily supply their own implementation of any algorithm. This is achieved with careful design and the use of interfaces, that are described later in this chapter.

Another important aspect is good portability. This is achieved by using the CMake build manager that generates platform-specific build configurations. Gram can be therefore easily built on Linux, Mac and Windows operating systems. While a CMake project cannot quite match the portability of Java used in GEVA, it is certainly better than plain Unix Makefiles.

The third objective is to create a open-source project that is easily approachable. The code is hosted on GitHub, the de-facto hosting service for open-source projects. The project also has thorough automated tests that help developers to faster understand the library interface and to grasp the functionality of each module. Both implementations described in previous chapter are distributed as archives on the authors' websites and do not offer a way how to contribute to their development.

This chapter in detail describes the class structure of the Gram library along with the build system and test suite.

4.1 System Overview

As shown in Figure 4.1, the `Evolution` class is the central part of the library. It handles evaluation and reproduction of the population as well as communication with the user through the `Logger` interface.

The evaluation is realized by an user-supplied implementation of the `Evaluator` interface, which is wrapped by an `EvaluationDriver`, for example `MultiThreadDriver`.

The population is a set of individuals that contain the genotype – a vector of integers. Note that individuals do not contain the phenotype, as it is dynamically mapped inside an `Evaluator` implementation.

The reproduction process is handled by an implementation of the `Reproducer` interface, such as `PassionateReproducer`. It uses instances of the `Selector`, `Crossover` and `Mutation` interfaces.

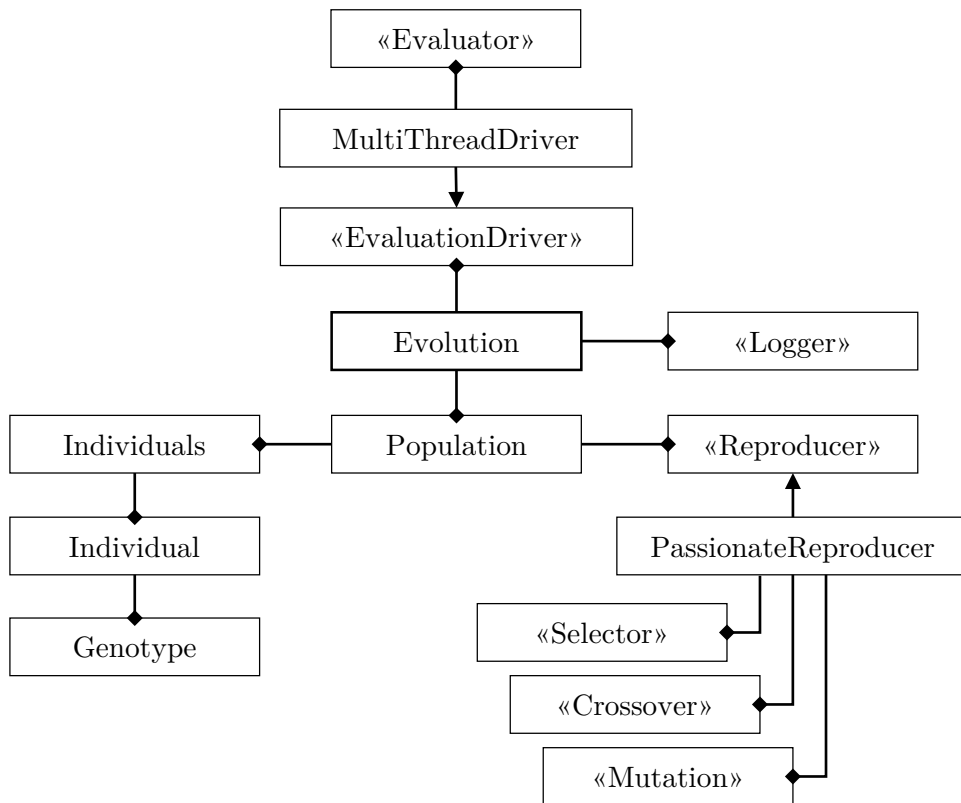


Figure 4.1: Diagram of Gram’s core classes

4.2 Success Definition and Evolution Run

The most significant class in the library is `Evolution` which provides a `run` method that starts the evolutionary algorithm. From its signature (see Listing 4.1) it is apparent that it works with a population of individuals and a terminating condition.

The fact the population is supplied by the user rather than initialized inside the method makes it easy to use the library in situations where the user already has an existing population. It might be the result of a previous run or a set of hand-crafted individuals.

Listing 4.1: Signature of the `run` method of `Evolution` class

```

Population Evolution::run(Population& population,
    function<bool(Population&)> successCondition)
  
```

The `successCondition` parameter is a function that represents the goal of the evolution. Both systems described in Chapter 3 terminate the algorithm after a certain number of generations or after the best individual reaches a certain level of fitness. While that might be sufficient for benchmarks and simple demos, in real applications, it might be desirable to evolve a whole group of good solutions and do additional operations with them outside of the grammatical evolution library.

Listing 4.2: Example of a success condition lambda function

```
[] (Population& population) -> bool {  
    return population.lowestFitness() < 0.1;  
}
```

The lambda function allows for more complex termination conditions by providing access to the whole population of individuals. The user is free to do an advanced statistical analysis on the individuals or just to simply set a fitness or generation limit. For an example of a simple implementation of `successCondition`, see Listing 4.2.

4.3 Initialization

For more ordinary situations, Gram users can create an initial population with the use of the `Initializer` abstract class. Its interface consists of a single method `initialize` (Listing 4.3) which creates a new population of the given size that reproduces with the specified reproducer.

Listing 4.3: Signature of the `initialize` method of `Initializer` abstract class

```
Population Initializer::initialize(unsigned long count,  
                                   shared_ptr<Reproducer> reproducer)
```

Gram by default offers a simple `RandomInitializer` class that performs a purely random initialization. More complex algorithms can be easily implemented by creating a new child class of `Initializer`.

4.4 Evaluation and Fitness Function

The evaluation of individuals is completely in the hands of the library user. Its aspects are specific to the problem at hand, but it usually consists of three basic steps:

- mapping individual's genotype to a phenotype,
- evaluating the phenotype,
- determining the fitness.

The evaluation takes place in a class inheriting from the base `Evaluator` abstract class. It defines the `evaluate` method which can be seen on Listing 4.4.

Listing 4.4: Signature of the `evaluate` method of `Evaluator` class

```
double Evaluator::evaluate(Genotype& genotype)
```

A complete example of a math evaluator can be found in Appendix B.

The evaluators are wrapped in an `EvaluationDriver` whose responsibility is to do some preparation steps before the evaluation itself. Even though users are free to implement their own drivers, Gram already contains with two implementations – `SingleThreadDriver` and `MultiThreadDriver`.

The single thread variant iterates over the whole population and evaluates the individuals one by one. The multi-thread driver is more complex as it creates a number of new threads, splits the population into chunks and assigns each chunk to a thread to evaluate.

From a performance standpoint, it would be beneficial to create a driver that implements the thread pool pattern. That would prevent from repeated creation and termination of threads for each generation. This process is very time-consuming and having few background threads for the whole evolution run would eliminate this bottleneck.

4.5 Reproduction

The reproduction process is invoked by calling the `reproduce` method on the current population. Internally the population calls the `reproduce` method on its `Reproducer` member. The signature of the method is shown in Listing 4.5.

Listing 4.5: Signature of the `reproduce` method of `Reproducer`

```
Individuals Reproducer::reproduce(Individuals& individuals)
```

The only implementation of the abstract class is called `PassionateReproducer`. It creates the new population entirely by applying the crossover operator on pairs of individuals chosen by a selector and then tries to mutate every single offspring.

The class internally depends on `Selector`, `Crossover` and `Mutation` abstract classes that are described in the following sections.

4.5.1 Selection

Picking parents from a set of individuals is the responsibility of the `Selector` abstract class. Its pivotal method is called `select` and the signature can be seen in Listing 4.6.

Listing 4.6: The `select` method for picking parents

```
Individual& Selector::select(Individuals& individuals)
```

Gram by default implements the tournament selection described in Subsection 2.1.3. As suggested, the notion of a good fitness can be configured and in this case, the configuration is done by an `IndividualComparer` object passed to the selector in a constructor. When the tournament selector randomly chooses a number of individuals, it compares their fitness values with the comparer, rather than directly with a relation operator. The comparer signature is provided in Listing 4.7.

Listing 4.7: The interface of `IndividualComparer`

```
bool IndividualComparer::isFirstFitter(Individual& first,  
                                       Individual& second)
```

This abstraction could be used to add support for multi-objective optimization in the future. The raw fitness is rarely used in any other part of the system, which would make it easy to write a custom implementation of the `IndividualComparer` and compare the individuals on a whole set of criteria instead of only one fitness value.

4.5.2 Crossover

The crossover operator combines two existing individuals into a new one. Gram makes the interface very readable by providing a `mateWith` method on the `Individual` class. As shown by Listing 4.8, it depends on generic `Crossover` abstract class that can be used to create new implementations.

Listing 4.8: Mate with

```
Individual Individual::mateWith(Individual& partner ,  
                                Crossover& crossover)
```

The `mateWith` method internally uses the crossover's method from Listing 4.9.

Listing 4.9: Crossover

```
Genotype Crossover::apply(Genotype& first ,  
                           Genotype& second)
```

By default, Gram contains the simple one-point crossover. However, the abstraction of the operator makes it easy to experiment with more complicated algorithms. In recent years, more sensible crossovers such as the structure-preserving crossover were proposed [4].

4.5.3 Mutation

Mutation is the second genetic operator used in grammatical evolution. In the Gram library, it can be used by passing a `Mutation` object to the `mutate` method of `Individual` as Listing 4.10 shows.

Listing 4.10: Mutate

```
void Individual::mutate(Mutation& mutation)
```

The individual handles the call in a similar fashion to how it handles the crossover operator – by calling the `apply` method of the `Mutation` operator (Listing 4.11).

Listing 4.11: Mutation

```
Genotype Mutation::apply(Genotype genotype)
```

Gram offers two mutation implementations. The first one is called `NaiveCodonMutation` and its naivety lies in iterating over each genotype codon and deciding whether to mutate it or not by executing an expensive call to a random number generator.

For that reason, Gram also has the `FastCodonMutation` that avoid frequent calls to the random number generator by using the inversion method. This optimization is in greater detail described in Chapter 5.

4.6 Other Tools

Apart from the essential algorithms for grammatical evolution, Gram also offers a few useful tools that make working with the library easier, namely a BNF grammar parser and a logging mechanism.

4.6.1 BNF parser

As BNF grammars are at the very core of grammatical evolution, Gram provides a parser that makes plugging in different grammars an easy process.

There are many variants and extensions of BNF. The format compatible with Gram's parser is on Listing 4.12. Note terminals enclosed in double-quotes.

Listing 4.12: BNF grammar format

```
<expr> ::= "(" <expr> <op> <expr> ")" | <var>
<op>    ::= "+" | "-" | "*"
<var>   ::= "x" | "1"
```

The grammar can be supplied as the form of a string to the `BnfRuleParser` method `parse` whose signature is on Listing 4.13.

Listing 4.13: Signature of the `parse` method

```
ContextFreeGrammar parse(string input)
```

4.6.2 Logger

Gram has logging capabilities to provide users an immediate feedback on the evolution run. This is enabled by the `Logger` abstract class that offers two methods shown in Listing 4.14.

Listing 4.14: Logger methods

```
void logProgress(Population& population)
void logResult(Population& population)
```

The `logProgress` method is called after evaluation of each population. The `logResult` method is called at the end of the evolution run as the user may want to log the result in a different way than a population in the middle of the run.

The default logger implementation in Gram is called `NullLogger` and does not perform any action. Users can easily implement their own logger that is specific to their problem.

4.7 Automated Tests

The source code of Gram is thoroughly tested. It comes with unit tests that verify the behavior of grammatical evolution algorithms. The project also contains an acceptance test that verifies the cooperation of all library modules. It can also serve as a starting point for users creating a new Gram project as it shows all necessary configuration and usage of each class.

Table 4.1: Code metrics

Type	Lines
Production code	1247
Testing code	1416

The ratio of production and testing code is in Table 4.1. It shows that the testing code more than doubles the number of lines in the project and also requires extra work and maintenance.

Automated tests, however, give developers working on the library a greater level of confidence when refactoring an existing code and creating new modules. Gram source code is hosted on GitHub, and after each code change, the continuous integration server runs all tests and verifies the change did not break the existing code and behaves according to specification.

Moreover, potential users can see that the library handles all edge-cases, rendering it reliable for use in production applications.

The tests themselves are written in a C++ testing framework called Catch developed by Phil Nash. A mocking library FakeIt by Eran Pe'er was used to isolate the tested code from the rest of the system and make random number generators more transparent.

4.8 Build System

Gram uses a cross-platform tool CMake to manage the build process. It allows developers to generate configuration for the native build system, for example, Makefiles on Linux, Visual Studio solutions on Windows or Xcode projects on Mac OS.

To build Gram, the operating system needs to have the following software installed:

- Git,
- CMake,
- a native build system,
- a C++14 compiler.

All details regarding the build process of Gram along with an example build can be found in [Appendix A](#).

Chapter 5

Performance Tuning

This chapter provides recommendations for creating high-performance grammatical evolution engines. It describes the optimization process Gram underwent including language-specific factors and variations of standard algorithms used in grammatical evolution.

- In Section 5.1, we will explain the process of profiling.
- In Section 5.2, we will analyze influence of various implementation details.
- In Section 5.3, we will explore the performance impact of random number generators.
- In Section 5.4, we will learn about an alternative approach to the mutation operator.
- In Section 5.5, we will describe fitness caching techniques.
- In Section 5.6, we will investigate performance gains of parallel evaluation.

All measurements were taken on a machine with Intel Core i7-7700K running at 4.2 GHz with 16 GB of 2666 MHz DDR4 memory. The operating system was Ubuntu 16.04, and all executables were compiled with Clang 3.8.

5.1 Profiling

To optimize a program for performance effectively, we usually use tools called profilers. Those tools perform a dynamic program analysis by collecting data such as memory usage, frequency of instructions or the frequency and duration of function calls.

The tool used for optimizing Gram's performance is called `perf` and it is exclusively available on Linux operating systems. This profiler collects statistical data by sampling the application during execution. It periodically probes the program's call stack using interrupts and approximates how often are individual functions called and how much of the execution time they consume.

This allows developers to focus on optimizing the most frequently used parts of the code. Without profilers, it is not obvious where the performance bottlenecks are, and developers may end up optimizing functions that have a negligible impact on performance.

5.2 Implementation Details

Generally speaking, development in programming languages such as C or C++ gives developers a very good control over the performance. During the optimization of Gram's performance, the most significant speedups were obtained by improvements in the memory management.

Avoiding object copying appeared to be very beneficial. In one case, adding a single `&` character to create a reference instead of a whole object by copying yielded a performance improvement of over 35 %. The code was located in the genotype-phenotype mapper (`ContextFreeMapper::map`) and was executed for all codons in the genotype of each individual in all generations.

The second issue was the memory allocation. In grammatical evolution systems, there are many algorithms that are constructing a defined number of items, for example:

- initializer creating a new population of individuals,
- initializer creating a new genotype of an individual,
- crossover creating a new genotype by combining two existing genotypes,
- or the tournament selector placing individuals in a tournament.

Rather than allocating memory for each element, we can allocate memory once for all of them, saving a large number of expensive system calls. In C++, most containers have a `reserve` memory that is built for this reason. Using it led to another significant performance improvements.

5.3 Random Number Generators

From the theory of grammatical evolution described in Section 2.2, it is apparent that it heavily relies on generating random numbers. Two principal methods are used to generate random numbers. The first measures a physical phenomenon that is considered to be random, such as atmospheric noise. The second method uses algorithms that are able to generate sequences of apparently random values from a short initial sequence called the seed.

Although the first method produces truly random numbers, it is very slow and requires additional equipment. Thus, the second method that generates only pseudo-random values is commonly used as it provides good enough results for most applications.

In this section, we will measure the impact of three pseudo random number generators on the GE performance:

- linear congruential generator – `minstd_rand` from C++ `<random>` module,
- Mersenne Twister – `mt19937` from C++ `<random>` module,
- XorShift – the 32-bit variant [8].

For their historical importance, linear congruential generators are included in the test despite not being very good by today's standards [7]. Mersenne Twister [9] is the default random number generator in many applications, and XorShift promises high performance with acceptable randomness parameters [8].

The tests were conducted on a simple symbolic regression problem with parameters listed in Table 5.1. Variations of this setup will be used for all experiments discussed in this chapter.

Table 5.1: Parameters of GE for symbolic regression problem used for performance tuning

Objective	Find approximation of $f(x) = x^4 + x^3 + x^2 + x$
Grammar	See Listing 5.1
Fitness cases	21 points for $x \in [-1.0, 1.0]$
Raw fitness	Sum of squared errors for all fitness cases
Population size	500 individuals
Genotype length	200 codons, fixed
Codon size	32bits
Selection	Tournament, size 5
Crossover	One-point, probability: 1.0
Mutation	Codon-level, probability: 0.1
Initialization	Random
Mapping	Maximum wraps: 3
Maximum generations	100
Success predicate	Lowest fitness score in the population under 0.000001

Listing 5.1: The BNF grammar used in the symbolic regression problem

```

<expr> ::= ( <expr> <op> <expr> ) | <var>
<op>   ::= + | - | *
<var>  ::= x | 1

```

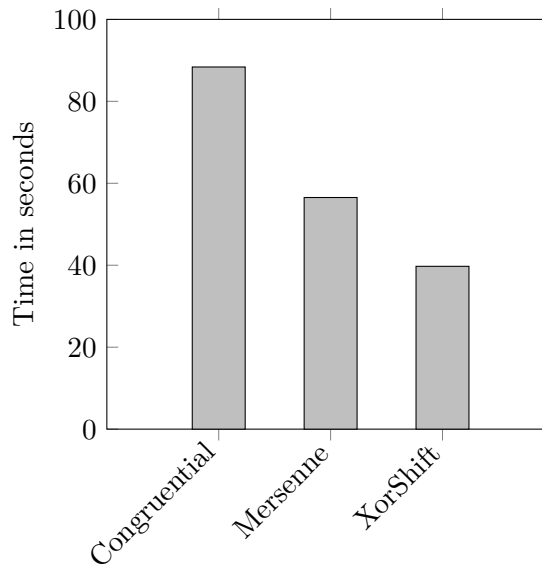


Figure 5.1: Time consumed by 1000 runs of GE using different random number generators

Figure 5.1 shows a great performance difference between the three algorithms. Simple symbolic regression is more than twice as slow when using linear congruential generator

compared to XorShift. This is most likely due to the modulo operator used in each iteration of the congruential algorithm. Mersenne Twister appears to be more than 40 % slower than XorShift, which is not surprising given its superior randomness rankings. The usage of a random number generator of lesser quality did not affect the success rates of grammatical evolution.

In the next chapter, we will focus on optimizing the mutation operator that uses the random number generators most often in our grammatical evolution system.

5.4 Optimizing Mutation Operator

As we have established in the previous section, the choice of a random number generator can have a great impact on the performance of grammatical evolution systems. Here we will try to further reduce the bottleneck by optimizing the biggest consumer of random numbers – the mutation operator.

In our version of GE, the decision whether to mutate a single codon follows a Bernoulli distribution with the probability of mutation p . The number of unaffected codons until the next occurrence of mutation follows a geometric distribution with the same parameter p .

Random variates of this distribution can be calculated with the use of the inversion method [3]. This method uses the cumulative distribution function F along with a random variable $u \in (0, 1)$ following a uniform distribution. Given that F can be inverted, the random variate s can be computed using Equation 5.1.

$$F^{-1}(u) = s = \left\lfloor \frac{\ln(1 - u)}{\ln(1 - p)} \right\rfloor \quad 0 < u < 1 \quad 0 < p < 1 \quad (5.1)$$

The value of s denotes the number of steps until the next mutation takes place. This calculation can, therefore, be used instead of checking each codon whether it should be mutated, effectively reducing the complexity of the check from $O(n)$ to $O(p \cdot n)$, where p is usually a value close to zero.

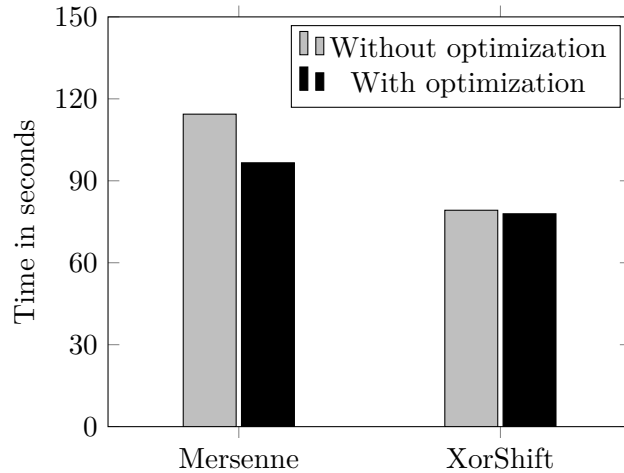


Figure 5.2: Time consumed by 1000 runs of GE with mutation probability $p = 0.2$ with and without the optimization

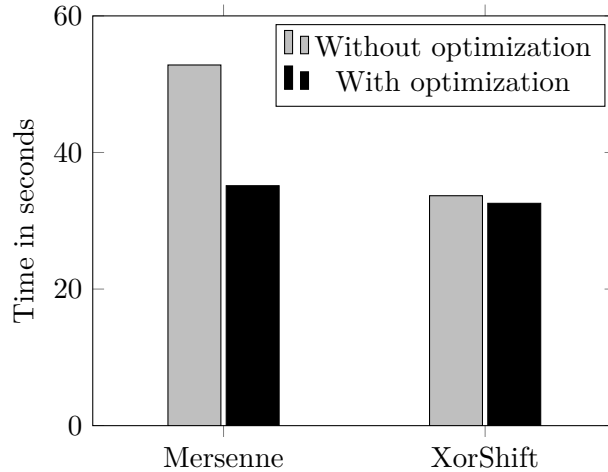


Figure 5.3: Time consumed by 1000 runs of GE with mutation probability $p = 0.05$ with and without the optimization

Figure 5.2 shows the performance improvement of the optimized mutation in combination with two different random number generators and high mutation probability $p = 0.2$. While we can see that the optimization does not have any significant impact on the XorShift generator, in the case of Mersenne Twister, it yields a 15 % improvement in execution time.

Results of runs with a lower probability $p = 0.05$ can be seen in Figure 5.3. This more real-world setting again does not show any improvement for XorShift generator. Mersenne Twister, on the other hand, performs 35 % faster with the optimization turned on, which effectively brings its execution time on the same level as XorShift. That leaves no practical reason for using XorShift instead of the standardized and statistically superior Mersenne Twister.

5.5 Fitness Caching

Evaluation of individuals and calculation of their fitness score is the most time-consuming task in many applications. One can avoid repeating this calculation for the same individuals by employing a technique called caching.

Example Gram applications employ a hashtable that contains the fitness values indexed by the individuals' phenotype – a string of characters. The problem with this approach is that all individuals have to always undergo an expensive genotype-phenotype mapping process to retrieve the fitness score from the cache.

The only way how to prevent repeated mapping is to index the fitness values by the genotype. In our experiments, this approach did not provide any improvement over the original system with phenotype-based caching. The ineffectiveness of this solution stems from the fact that a large number of genotypes can be mapped to the same phenotype. The fact that populations are usually very genetically diverse then leads to a high count of cache misses, meaning only a few individuals can avoid the expensive evaluation.

In Figure 5.4, we can see the benefit of phenotype-based caching where it reduced the overall execution time by 38 %. Of course, the more expensive the fitness function is, the greater the performance benefit. It is also important to account for the grammar size

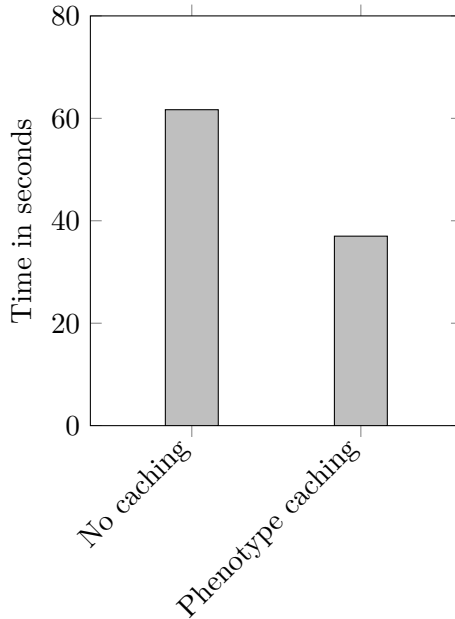


Figure 5.4: Time consumed by 100 runs with two caching setups

because use more complex grammars will lead to greater phenotype diversity. That will in turn cause lower number of cache hits, rendering the caching technique less beneficial.

5.6 Using Multiple Cores

A common approach to reducing the execution time of many types of applications is employing multiple processor cores. Gram uses this technique only for the evaluation of individuals, as it usually is the most time-consuming task in grammatical evolution. In our tests, the computational intensity of real-world tasks was simulated by repeating the evaluation of symbolic regression individuals 100 times.

The `MultiThreadDriver` creates n threads for the evaluation in each generation. There are more effective approach to multi-threading, such as the *thread pool* pattern, which creates n threads at the start of application and reuses those threads throughout the application run. This removes the overhead caused by repeated creation and termination of new threads.

However, in Figure 5.5, we can see that even our more straightforward implementation of multi-threading yields a significant performance improvement. Using two cores instead of one brings the overall execution time of 100 runs down by 32 %.

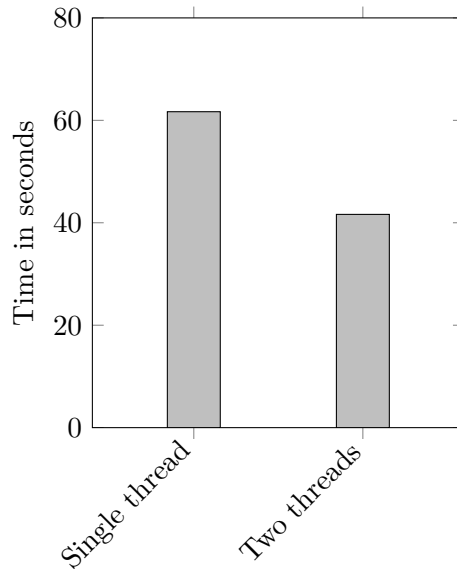


Figure 5.5: Time consumed by 100 runs of GE with one and two threads enabled

5.7 Profiling Optimized Grammatical Evolution System

Figure 5.6 shows the percentage of execution time consumed by individual GE algorithms in Gram with all optimizations described in this chapter enabled.

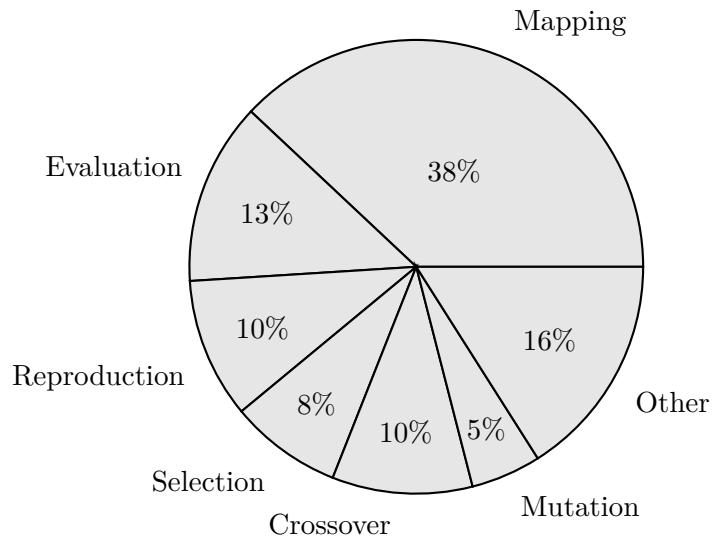


Figure 5.6: Percentage of execution time consumed by algorithms in optimized GE system

A notable finding is that over one third of the execution time (38 %) is spent on the genotype-phenotype mapping. This is probably due to frequent use of the modulo operator, which tends to be incredibly slow on today’s processors. Although the percentage will be lower in applications with more expensive individual evaluation, the mapping process seems like a good candidate for further optimizations.

Chapter 6

Experiments

In this chapter, we will test the Gram library in two applications.

- In Section 6.1, we will focus on symbolic regression, comparing the new library to an existing implementation.
- In Section 6.2, we will apply grammatical evolution on a novel task: automating test-driven development.

As mentioned in Chapter 5, all measurements were taken on a machine with Intel Core i7-7700K running at 4.2 GHz with 16 GB of 2666 MHz DDR4 memory. The operating system was Ubuntu 16.04, and all projects were compiled using Clang 3.8.

6.1 Symbolic Regression

Symbolic regression is a statistical process that searches the space of mathematical expressions to find the model that best fits the given dataset. Historically, symbolic regression has been used by many as a benchmark application for grammatical evolution systems.

We will compare Gram to AGE, a C++ library described in Chapter 3. We will focus on the execution time needed for evolving a viable solution as well as the frequency of success for each generation. Our graphs show a cumulative frequency of success – the sum of frequencies of success in the current and all previous generations – a commonly used metric for examining the quality of grammatical evolution runs.

To test both libraries, we performed 1000 runs with the target being a fourth degree polynomial: $x^4 + x^3 + x^2 + x$. The parameters of grammatical evolution are listed in Table 6.1.

Listing 6.1: The BNF grammar used in symbolic regression problem

```
<expr> ::= ( <expr> <op> <expr> ) | <var>
<op>    ::= + | - | *
<var>   ::= x | 1
```

Table 6.1: Parameters of GE for symbolic regression problem

Objective	Find approximation of $f(x) = x^4 + x^3 + x^2 + x$
Grammar	See Listing 6.1
Fitness cases	21 points for $x \in [-1.0, 1.0]$
Raw fitness	Sum of squared errors for all fitness cases
Population size	500 individuals
Genotype length	200 codons, fixed
Codon size	32 bits
Selection	Tournament, size 5
Crossover	One-point, probability: 1.0
Mutation	Codon-level, probability: 0.1
Initialization	Random
Mapping	Maximum wraps: 3
Maximum generations	50
Success predicate	Lowest fitness score in the population under 0.000001

As can be seen in Figure 6.1, Gram performs better compared to AGE, if more than 15 generations are spent. That might be due to a slightly different reproduction algorithm. Gram creates all new individuals using the crossover operator and then tries to apply mutation to all of them. AGE, on the other hand, creates new individuals either by crossover or mutation of individuals from the previous generation, never applying both operators on a single individual. This might lead to lower genetic diversity in later generations and consequently to lower success rates.

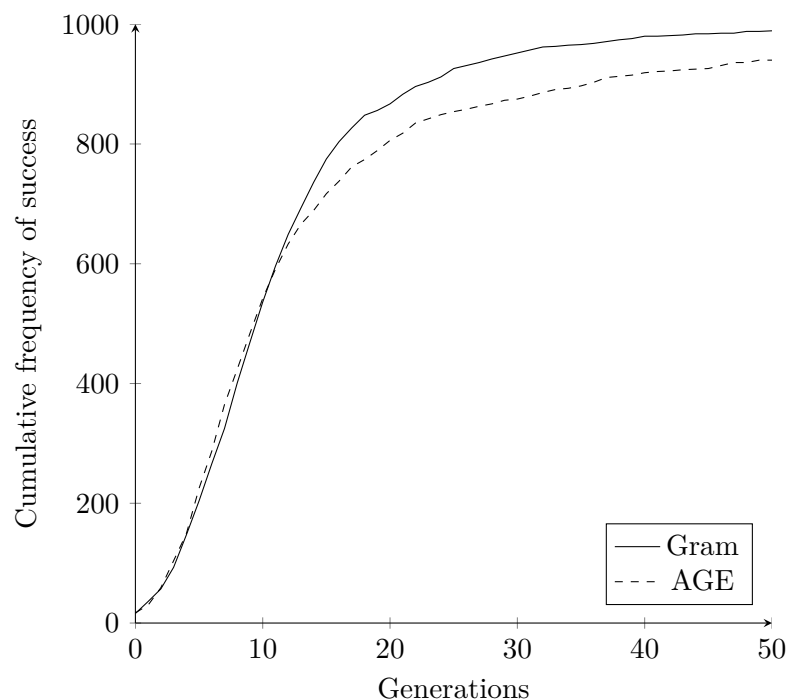


Figure 6.1: Frequency of cumulative success in 1000 GE runs of symbolic regression with target function $x^4 + x^3 + x^2 + x$

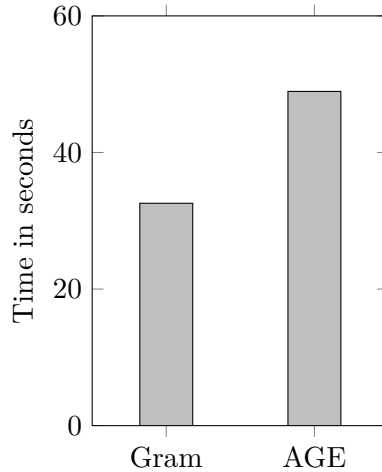


Figure 6.2: Time consumed by 1000 GE runs of symbolic regression with target function $x^4 + x^3 + x^2 + x$

Figure 6.2 shows the CPU time consumed by both applications where Gram appears to be 33 % faster than AGE. This is probably caused by higher success rates of Gram shown in Figure 6.1 along with faster evaluation, phenotype-based fitness caching and overall more optimized implementation. It is worth noting, that AGE uses a fast random number generators and the optimized mutation operator described in Chapter 5.

6.2 Automating Test-Driven Development

In addition to the obligatory symbolic regression problems, Gram has been also used for a novel task: automating test-driven development.

Test-driven development is a widely used process of creating software products with automated tests. In this process, developers first write tests based on given specifications and then proceed to write the minimal amount of production code that passes those tests.

Automated testing plays an important role in quality assurance of software products. Its purpose is to automatically verify that each version behaves according to specifications. That leads to less time spent on manual testing and lower number of bugs, which can significantly reduce the cost of development and maintenance of software.

Automating the creation of production code could allow developers to either create more specifications or make them more detailed, possibly speeding up the process of software development or making the resulting product more reliable.

6.2.1 Fitness Function

To apply grammatical evolution to this problem, we have to find a way how to determine the fitness of candidate solutions. For this purpose, we use test assertions. Most testing frameworks allow developers to verify, whether the tested code returned the expected value. We can determine the fitness by calculating the distance between the desired and actual value. Our system calculates the fitness score f for a candidate solution s using equation:

$$f(s) = \sum_{i=1}^t dist(e_i, a_i) \quad (6.1)$$

where t is the number of test assertions, e_i is the expected value and a_i is the actual value returned by the tested code.

Computation of the distance in assertions varies based on the data type. In our tests, we have used a simple equation for numeric types:

$$dist(e, a) = abs(e - a)$$

In assertions working with arrays, we have to account for the situation where the number of elements in the actual array is not the same as in the desired array. In those cases, we simply set the fitness to a predefined constant C . For arrays of the same size n , the resulting fitness is equal to the sum of distances of each pair of corresponding elements:

$$dist(e, a) = \begin{cases} \sum_{i=1}^n dist(e_i, a_i), & \text{if arrays are of the same size } n, \\ C, & \text{otherwise.} \end{cases}$$

6.2.2 Experiment Setup

The target language of choice is PHP with its testing framework PHPUnit. PHP is suitable for this task because of its weak typing and implicit type conversions. While not always desirable in real-world systems, these properties make the language more forgiving to semantic type-related error that usually occur in code generated by grammatical evolution.

The goal of our experiment is to generate the `array_filter` function based on hand-written automated tests. The generated function must pass all tests for the experiment to be considered successful.

The first parameter of this function is an array of elements to be filtered. The second parameter is a lambda function that takes one argument – an element of the array – and decides whether it should be kept in the array or not. `array_filter` returns an array of elements that meet the filter requirements. The signature of the function is in Listing 6.2

Listing 6.2: Signature of the `array_filter` function

```
function array_filter(array $input, callback $filter): array
```

The `$filter` callback used in tests is in Listing 6.3 and it passes integers that are greater than zero. The `$input` array along with the expected output is shown in Table 6.2.

Listing 6.3: The `$filter` used in tests

```
$filter = function (int $item) { return $item > 0 }
```

Table 6.2: Data used in tests of `array_filter`

Input	Correct output
<code>[]</code>	<code>[]</code>
<code>[-10, -5, -3, -1]</code>	<code>[]</code>
<code>[-10, -1, 3, 5]</code>	<code>[3, 5]</code>
<code>[1, 20, 42]</code>	<code>[1, 20, 42]</code>

The parameters of grammatical evolution are listed in Table 6.3.

Table 6.3: Parameters of GE for evolving `array_filter` function

Objective	Create the <code>array_filter</code> function
Grammar	See Listing C.1
Fitness cases	See Table 6.2
Raw fitness	See Equation 6.1
Population size	200 individuals
Genotype length	40 codons, fixed
Codon size	32 bits
Selection	Tournament, size 5
Crossover	One-point, probability: 1.0
Mutation	Codon-level, probability: 0.15
Initialization	Random
Mapping	Maximum wraps: 3
Maximum generations	100
Success predicate	Fitness is 0

6.2.3 Results and Discussion

In Figure 6.3, we can see that in most runs the system was able to create a viable solution in less than 60 generations. This is probably due to a relatively limited grammar that contains only rules necessary to create the target function.

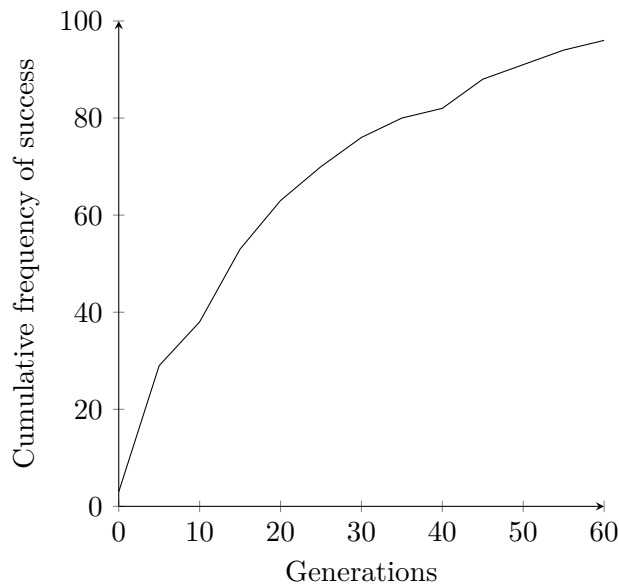


Figure 6.3: Cumulative frequency of success in 100 GE runs of evolving `array_filter`

The generated `array_filter` function is shown in Listing 6.4. Some of the solutions contain redundant code after line 3 (e.g. repeated initialization of the `$output` variable) and unreachable code after the `return` statement on line 9. While this does not affect the behavior of the function, it makes it harder to understand, which could lead into more difficult maintenance of the code in the future. However, defects like this can be automatically fixed by using a static analysis tools after the evolution finds a viable solution.

Listing 6.4: The evolved `array_filter` function

```
1 <?php
2 function array_filter($input, $filter) {
3     $output = [];
4     foreach ($input as $item) {
5         if ($filter($item)) {
6             $output[] = $item;
7         }
8     }
9     return $output;
10 }
```

Despite using weakly typed language with implicit type conversions, a great number of candidate solutions were semantically flawed. Most semantic errors were caused by the usage of undefined variables – an aspect of programming languages that is not captured in their formal grammars.

This could be eliminated by using information provided by static analysis tools in the rule selection process.

While the evolved code is fairly simple, this experiment shows its possible to utilize grammatical evolution in real-world software engineering process. Systems like this could be used to create more reliable software, faster.

Chapter 7

Conclusion

This thesis introduced grammatical evolution, a grammar-based approach to genetic programming originally proposed by Michael O’Neil and Conor Ryan. We have reviewed the basic principles of genetic programming and described the novel approach of grammatical evolution to the representation of individuals and consequently the crossover and mutation operators.

Following the introduction is a review of two grammatical evolution implementations: GEVA and AGE. We have examined the implemented GE algorithms, reflected on the overall system design and summarized strong and weak points of each implementation.

Based on the knowledge from previous chapter, we have proposed a new library implementing grammatical evolution called Gram. Main goals of the project were good performance, modular architecture and reliability ensured by thorough automated tests – a mix of the best features of the mentioned implementations.

Next chapter outlined the performance tweaks utilized in the Gram library. We have found that an important performance consideration is memory management, notably avoiding unnecessary memory allocations. We have documented the performance impact of random number generators in grammatical evolution systems. Our benchmark revealed that the fastest RNG is Xorshift, followed by Mersenne Twister (40 % slower) and linear congruential engine (100 % slower). The next optimization reduced the number of random number generator calls in the mutation operator. Thanks to the inversion method, we were able to erase the performance difference between the widely regarded Mersenne Twister and the less popular XorShift generator. We have also introduced a phenotype-based fitness caching which improved the performance by 38 %. The last optimization is the utilization of multiple CPU cores for the fitness evaluation. In our benchmark, doubling the number of cores brought the execution time down by another 32 %.

Gram was also compared to an existing GE implementation. In symbolic regression problems, its execution time was 32 % lower compared to AGE, the faster of the two reviewed systems.

We have also used GE in test-driven development: we were able to automate the process of creating function implementation by evolving an `array_filter` function based solely on simple unit tests. While the evolved function is fairly simple, we have shown that grammatical evolution could be used in the process of software development. This part of the thesis was also presented in the form of a poster on Excel@FIT 2017 conference.

This thesis and the accompanying software project have fulfilled its goals. The Gram library could be further improved by employing more complex initialization, crossover and

mutation operators using derivation trees as seen in GEVA. Recent research shows they can be superior to the original operators proposed by O’Neill and Ryan.

From our benchmarks, it is apparent, that the mapping process is currently the slowest part of the library because of frequent use of the modulo operator. Focusing on building grammars that do not require use of such operator could yield another significant performance improvements.

As we have demonstrated in Chapter 6, grammatical evolution can be applied to process of creating software with automated tests. Leveraging static analysis tools might lead to a system that is able to create complex production code in chosen language based solely on unit tests. I would like to see more research focused on this particular application of GE.

I hope the Gram library to act as a platform for further experiments and research of grammatical evolution, as it has shown competitive performance and easy application to various types of tasks.

Bibliography

- [1] Banzhaf, W.: Genotype-Phenotype-Mapping and Neutral Variation – A case study in Genetic Programming. In *Parallel Problem Solving from Nature III*, vol. 866. Springer-Verlag. October 1994. ISBN 3-540-58484-6. pp. 322–332.
- [2] Byrne, J.; O’Neill, M.; Brabazon, A.: Structural and Nodal Mutation in Grammatical Evolution. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*. ACM. July 2009. ISBN 978-1-60558-325-9. pp. 1881–1882.
- [3] Devroye, L.: *Non-Uniform Random Variate Generation*. Springer. 1986. ISBN 0-387-96305-7.
- [4] Harper, R.; Blair, A.: A Structure Preserving Crossover In Grammatical Evolution. In *2005 IEEE Congress on Evolutionary Computation*, vol. 3. IEEE. 2005. ISBN 0-7803-9363-5. pp. 2537–2544.
- [5] Kimura, M.: *The Neutral Theory of Molecular Evolution*. Cambridge University Press. 1983. ISBN 9780521317931.
- [6] Koza, J. R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*. A Bradford Book. 1992. ISBN 0262111705.
- [7] Marsaglia, G.: Random Number Generators. *Journal of Modern Applied Statistical Methods*. vol. 2, no. 1. May 2003: pp. 2–13.
- [8] Marsaglia, G.: Xorshift RNGs. *Journal of Statistical Software*. vol. 8, no. 14. 2003: pp. 1–6.
- [9] Matsumoto, M.; Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*. vol. 8, no. 1. January 1998: pp. 3–30.
- [10] Nohejl, A.: *Grammatical Evolution*. Bachelor’s thesis. Faculty of Mathematics and Physics, Charles University in Prague. 2011.
- [11] O’Neill, M.; Hemberg, E.; Gilligan, C.; et al.: GEVA - Grammatical Evolution in Java (v2.0). June 2011. url: <http://ncra.ucd.ie/geva/geva.pdf>.
- [12] O’Neill, M.; Ryan, C.: Grammatical Evolution. *IEEE Transactions on Evolutionary Computation*. vol. 5, no. 4. August 2001: pp. 349–358.

- [13] O'Neill, M.; Ryan, C.: *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language (Genetic Programming)*. Springer. 2003. ISBN 978-1-4613-5081-1.
- [14] Poli, R.; Langdon, W. B.; McPhee, N. F.: *A field guide to genetic programming*. Published via <http://lulu.com> and freely available at <http://www.gp-field-guide.org.uk>. 2008.

Appendices

Appendix A

Building Gram

Gram uses cross-platform tool CMake to manage the build process. The instructions below apply for Linux-based operating systems. Building on Windows and Mac OS might slightly vary.

To be able to proceed with the process, your operating systems has to have the following software installed:

- Git,
- CMake,
- a native build system,
- a C++14 compiler.

The sequence of commands on Listing A.1 downloads the Gram source code, compiles it as a static library and also creates executable binaries running unit and acceptance tests.

Listing A.1: Gram build process

```
$ git clone https://github.com/jansvoboda11/gram
$ cd gram
$ mkdir build
$ cd build
$ cmake -DGRAM_BUILD_TESTS ..
$ make
```

Paths of the created binaries are listed in Table A.1.

Table A.1: Paths to Gram binaries

Static library	<code>gram/build/src/libgram.a</code>
Unit tests	<code>gram/build/test/unit/utest</code>
Acceptance tests	<code>gram/build/test/acceptance/atest</code>

Appendix B

Example Gram Project

Listing B.1: Example of a Gram evaluator

```
class StringDiffEvaluator : public Evaluator {
public:

    StringDiffEvaluator(unique_ptr<ContextFreeMapper> mapper, string desired)
        : mapper(move(mapper)), desired(desired) {
        //
    }

    double evaluate(const Genotype& genotype) noexcept {
        try {
            return calculateFitness(mapper->map(genotype));
        } catch (...) {
            return 1000.0;
        }
    }

private:

    double calculateFitness(string program) {
        unsigned long shorter = min(desired.length(), program.length());
        unsigned long longer = max(desired.length(), program.length());

        double fitness = static_cast<double>(longer - shorter);

        for (unsigned long i = 0; i < shorter; i++) {
            if (program[i] != desired[i]) {
                fitness += 1.0;
            }
        }

        return fitness;
    }

    unique_ptr<ContextFreeMapper> mapper;
    string desired;
};
```

Listing B.2: Example of a Gram project

```

// selection
unsigned long size = 5;
auto numGen1 = make_unique<XorShiftNumberGenerator>();
auto comparer = make_unique<LowFitnessComparer>();
auto selector = make_unique<TournamentSelector>(size, move(numGen1), move(comparer));

// crossover
auto numGen2 = make_unique<XorShiftNumberGenerator>();
auto crossover = make_unique<OnePointCrossover>(move(numGen2));

// mutation
Probability probability(0.1);
auto numGen3 = make_unique<XorShiftNumberGenerator>();
auto stepGen = make_unique<BernoulliStepGenerator>(probability, move(numGen3));
auto numGen4 = make_unique<XorShiftNumberGenerator>();
auto mutation = make_unique<FastCodonMutation>(move(stepGen), move(numGen4));

// reproduction
auto reproducer =
    make_shared<PassionateReproducer>(move(selector), move(crossover), move(mutation));

// mapping
string bnfGrammar =
    "<word> ::= <word> <char> | <char>\n"
    "<char> ::= \"g\" | \"r\" | \"a\" | \"m\"";
BnfRuleParser parser;
auto grammar = parser.parse(bnfGrammar);
unsigned long wrapLimit = 3;
auto mapper = make_unique<ContextFreeMapper>(grammar, wrapLimit);

// evaluation
string target = "gram";
auto evaluator = make_unique<StringDiffEvaluator>(move(mapper), target);
auto evaluationDriver = make_unique<SingleThreadDriver>(move(evaluator));

// logging
auto logger = make_unique<NullLogger>();

// initialization
auto numGen5 = make_unique<XorShiftNumberGenerator>();
unsigned long genotypeLength = 100;
RandomInitializer initializer(move(numGen5), genotypeLength);
unsigned long populationSize = 200;
Population initial = initializer.initialize(populationSize, reproducer);

// evolution
Evaluation evolution(move(evaluationDriver), move(logger));

// run
Population result = evolution.run(initial, [](Population& current) -> bool {
    return current.lowestFitness() == 0.0;
});

```

Appendix C

Used PHP Grammar

Listing C.1: Subset of PHP's grammar used by GE in the task of automating TDD

```
<program> ::= <?php function array_filter($input, $condition) { <stmts> }

<stmts> ::= <stmt> <stmts>
          | <stmt>

<stmt> ::= <array-initialization>
          | <array-push>
          | <foreach-loop>
          | <if-condition>
          | <return>

<array-initialization> ::= <array-var> = [];
<array-push> ::= <array-var> [] = <item-var> ;
<foreach-loop> ::= foreach ( <array-param> as <item-var> ) { <foreach-stmts> }
<if-condition> ::= if ( <logical-expression> ) { <if-statements> }
<return> ::= return <array-var> ;
<logical-expression> ::= <callable-param> ( <item-var> )

<foreach-stmts> ::= <array-initialization>
                  | <array-push>
                  | <if-condition>
                  | <return>

<if-stmts> ::= <array-initialization>
            | <array-push>
            | <foreach-loop>
            | <return>

<array-var> ::= $var
<item-var> ::= $item

<array-param> ::= $input
<callable-param> ::= $condition
```