Bachelorthesis

Artificial Art: Image Generation using Evolutionary Algorithms

An der Fachhochschule Dortmund
im Fachbereich Informatik
Studiengang Informatik
Vertiefungsrichtung Praktische Informatik
erstellte Bachelorthesis

von
Johannes Rückert

Betreuer:
Prof. Dr.-Ing. Christoph M. Friedrich

Dortmund, September 16, 2013

Abstract

This work continues where (Rueckert 2013) left off: Expression-based genetic programming is used in the context of evolutionary art to create and evolve images. Expression-based genetic programming is an evolutionary optimization technique using hierarchical expression trees as representation of the genotype. Evolutionary art is a field in which evolutionary algorithms are used to create works of art; it is based on the works of (Dawkins 1986) and (Sims 1991). Different ways of creating images (evolving fractal and arithmetic functions) and evaluating them (interactive and automatic evaluation) are implemented, analyzed and compared. JGAP, an evolutionary computation framework supporting genetic programming, will be used throughout this work for the task of evolving the genotypes.

Kurzfassung

Diese Arbeit knüpft an (Rueckert 2013) an: Ausdrucksbasierte genetische Programmierung wird im Kontext evolutionärer Kunst eingesetzt, um Bilder zu erstellen und evolvieren. Ausdrucksbasierte genetische Programmierung ist eine evolutionäre Optimierungstechnik, die hierarchische Ausdrucksbäume nutzt, um Genotypen zu repräsentieren. Evolutionäre Kunst ist ein Anwendungsgebiet, in dem evolutionäre Algorithmen zur Erstellung von Bildern genutzt werden; begründet wurde das Feld durch (Dawkins 1986) und (Sims 1991). Verschiedene Techniken zur Bilderstellung (fraktale und arithmetische Funktionen) und -bewertung (interaktive und automatische Bewertung) werden implementiert, analysiert und verglichen. JGAP, ein genetische Programmierung unterstützendes evolutionary computation Framework, wird in der gesamten Thesis zur Evolution der Genotypen eingesetzt.

Contents

Contents i

List of Figures iv

1 Introduction 1
1.1 Introduction . . . 1
1.2 Overview . . . 1

2 Evolutionary Algorithms 3
2.1 Introduction . . . 3
2.1.1 Evolutionary computation and evolutionary algorithms . . . 3
2.1.2 No-free-lunch theorem . . . 4
2.1.3 Machine Learning . . . 4
2.2 The evolutionary algorithm . . . 5
2.3 Techniques . . . 7
2.3.1 Evolution strategies . . . 7
2.3.2 Evolutionary programming . . . 8
2.3.3 Genetic algorithms . . . 9
2.4 Genetic programming . . . 10
2.4.1 Representation . . . 11
2.4.2 Population initialization . . . 12
2.4.3 Selection . . . 14
2.4.4 Genetic Operators . . . 18
2.4.5 Primitive set . . . 21
2.4.6 Strongly-typed GP . . . 22
2.4.7 Reflection and object-oriented GP . . . 23

3 Evolutionary art 25
3.1 Introduction . . . 25
3.2 Representation . . . 25
3.2.1 Expression-based representation . . . 26
3.2.2 Other representations . . . 27

3.3 Fitness function 28
3.3.1 Interactive evaluation 29
3.3.2 Automatic evaluation 31
3.3.3 Multi-objective evolutionary art 38

4 Reflection-based Genetic Programming (ReGeP) 39
4.1 Introduction 39
4.2 Overview 40
4.3 Extending ReGeP 40
4.4 Multi-threaded function set creation 41
4.5 Non-static function set elements 42

5 Java package for evolutionary art (Jpea) 44
5.1 Introduction 44
5.2 Overview 44
5.3 Extending Jpea 45
5.4 Image evaluation 46
5.4.1 Interactive evaluation 46
5.4.2 Automatic evaluation 46
5.4.3 Multi-objective image evaluation 52
5.4.4 Multi-threaded image evaluation 52
5.5 Image creation 53
5.5.1 Fractal image creation 53
5.5.2 Multi-threaded image creation 54
5.5.3 Improving the genotype-phenotype-mapping 55
5.5.4 The primitive sets 57

6 Results and analysis 59
6.1 Introduction 59
6.1.1 Run configuration 59
6.2 Comparing image creation methods 61
6.2.1 Automatically defined functions 61
6.3 Interactive image evolution 65
6.4 Automatic image evolution 67
6.4.1 Global contrast factor 67
6.4.2 Image complexity 67
6.4.3 Fractal dimension 71
6.4.4 Behavior for increased population and generation parameters 73
6.5 Multi-objective image evolution 75
6.5.1 Bloat control 75
6.5.2 Multiple aesthetic measures 77
6.5.3 Comparing evaluation speeds 77

7 Conclusion and future prospects **81**

A The software environment **83**

A.1 Frameworks and libraries 83

A.1.1 JGAP 83

A.1.2 Other libraries 85

A.2 Reflection-based Genetic Programming (ReGeP) 87

A.2.1 Processing 88

A.2.2 FunctionDatabase 89

A.3 Java package for evolutionary art (Jpea) 92

A.3.1 Image creation 92

A.3.2 Image evaluation 93

A.3.3 The application framework 94

List of Figures

2.1 The four steps of the evolutionary process. 7
2.2 GP syntax tree representing $max(x+x, x+3*y)$. Source: (Poli, Langdon, and McPhee 2008) 12
2.3 Creation of a tree using the *full* method and a maximum depth of 2 (t = time). Source: (Poli, Langdon, and McPhee 2008) 13
2.4 Creation of a tree using the *grow* method and a maximum depth of 2 (t = time). At t = 2, a terminal is chosen, closing off that branch of the tree before it has reached the maximum depth. Source: (Poli, Langdon, and McPhee 2008) 14
2.5 Example of Pareto optimality in two dimensions. Solution A and B do not dominate each other, solutions 1, 2 and 3 dominate B, and A is dominated by solution 1. Source: (Poli, Langdon, and McPhee 2008) 18
2.6 Example for one-point subtree crossover, illustrating also that subtrees in the offspring are copies of the original subtrees, leaving the parent individuals unchanged. Source: (Poli, Langdon, and McPhee 2008) 19
2.7 Example for subtree mutation where a leaf is selected as the mutation point. Source: (Poli, Langdon, and McPhee 2008) 20

3.1 Examples for simple expressions. Source: (Sims 1991) 26
3.2 Example for a more complex image. Source: (Sims 1991) 28

3.3 (a) © 2005 D. A. Hart, (b) © Tatsuo Unemi (image taken from (Lewis 2008)) 30
3.4 (a) © Derek Gerstmann, (b) © David K. McAllister, (c) © Tim Day, (d) © Ashley Mills (images taken from (Lewis 2008)). 30
3.5 Example results from independent evolutionary runs using automatic evaluation. Source: (Machado and Cardoso 2002) 32
3.6 Image examples of automatic image evaluation using the Ralph aesthetic measure. (a) and (b) were evolved using the same color target, while (c) and (d) are example results for higher DFN values. Source: (Ross, Ralph, and Zong 2006) 33
3.7 Example results from independent evolutionary runs using fractal dimension automatic evaluation. Source: (Heijer and Eiben 2010a) . 34
3.8 Example results for image evolution using Benford's law as aesthetic measure. Source: (Heijer and Eiben 2010b) 35
3.9 Example results of GCF-guided image evolution. Source: (Heijer and Eiben 2010b) 36
3.10 Example results with enhanced contrast using information theory as the aesthetic measure. Source: (Heijer and Eiben 2010b) 37

5.1 The local contrast is calculated using the neighboring pixels. Source: (Matković et al. 2005) 51
5.2 Creation of superpixels at different resolutions. Source: (Matković et al. 2005) 51
5.3 Mapping of genotype to phenotype. Source: (Ekárt, Sharma, and Chalakov 2011) 56

6.1 Phenotype creation time for function image creation and fractal image creation. 62
6.2 Image examples of fractal ((a) and (b)) and function ((c) and (d)) image creation. 63

6.3 Genotype complexity (a) compared to phenotype creation time (b) and fitness development (c) when using ADFs and GCF evaluation. 64

6.4 Image examples of interactive image evolution at different stages of the evolution: Initial population (a), after five generations (b), after ten generations (c) and after 15 generations (d). 66

6.5 Example results of GCF-guided image evolutions. 68

6.6 Development of genotype complexity for GCF-guided image evolution. 68

6.7 Development of fitness values (a) and image evaluation times (b) for the GCF-guided image evolution. 69

6.8 Example results of IC-guided image evolutions. 70

6.9 Development of genotype complexity in the IC-guided image evolution. 70

6.10 Development of fitness values (a) and image evaluation times (b) in the IC-guided image evolution. 71

6.11 Example results of FD-guided image evolutions. 72

6.12 Development of genotype complexity. 72

6.13 Development of average fitness values (a) and image evaluation times (b). 73

6.14 Development of average fitness values (a) and genotype complexity (b) for an increased population size. 74

6.15 Development of average fitness values (a) and genotype complexity (b) for an increased generation count. 75

6.16 Development of GCF/IC fitness and genotype complexity. 76

6.17 Example results of multi-objective image evolutions. 77

6.18 Development of overall and individual fitness for multi-objective image evolution. 78

6.19 Comparison of evaluation times. 80

A.1 Processing principle. Source: (Rueckert 2013) 89

A.2 Processing in `regep-core`. Source: (Rueckert 2013) 90

A.3 Processing in regep-jgap. Source: (Rueckert 2013) 91
A.4 Object nesting of the GP functions. 91
A.5 Important classes and interfaces of the phenotype creation package. 93
A.6 Overview of the important classes and interfaces of the application package. 95

Chapter 1

Introduction

1.1 Introduction

Evolutionary computation is a field that covers techniques simulating different aspects of evolution and was formed during the 1990s in an effort to bring together researchers from the fields of evolutionary programming, evolution strategies and genetic algorithms (Bäck, Fogel, and Michalewicz 1997). Genetic programming is another evolutionary algorithm, first formally described in (Koza 1992), in which executable computer programs are evolved. Evolutionary art is a field in which evolutionary algorithms are applied to evolve works of art. The field was inspired by (Dawkins 1986) and one of its pioneers was Sims, who used genetic programming to evolve LISP expressions which were then used to generate images in (Sims 1991). In this thesis, the above-mentioned techniques and fields will be described, and different ways of creating and evaluating images will be implemented, tested and compared.

The printed versions are black and white, a digital color version[1] should have accompanied any printed version.

1.2 Overview

This thesis is structured into seven chapters and an appendix.

[1] Also available on `http://saviola.de` (visited on 08/25/2013)

The chapter serves as an introduction to and general overview of the thesis. Chapter 2 provides an introduction to evolutionary algorithms and genetic programming in more detail, while chapter 3 will give an overview of evolutionary art and important works in this field. Chapter 4 and 5 will shortly introduce the two software package ReGeP and Jpea and describe in which ways they were extended during the work on this thesis. Chapter 6 will present results of different image creation and evaluation approaches, as well as analyze and compare them, and chapter 7 will lastly draw a conclusion and point out future prospects. Appendix A contains a more detailed description of ReGeP and Jpea, plus other software libraries and frameworks that were used in this thesis.

Chapter 2

Evolutionary Algorithms

This chapter serves as an introduction to evolutionary algorithms, especially genetic programming (GP). After giving a short overview of evolutionary computation (EC) and evolutionary algorithms in general (section 2.1), the basic evolutionary algorithm will be described in section 2.2, before the three main evolutionary algorithms are introduced in section 2.3. GP will be explored in depth in section 2.4.

2.1 Introduction

2.1.1 Evolutionary computation and evolutionary algorithms

Evolutionary computation is a field that covers techniques simulating different aspects of evolution. According to (Bäck, Fogel, and Michalewicz 1997, p. vii), "The field began in the late 1950s and early 1960s as the availability of digital computing permitted scientists and engineers to build and experiment with various models of evolutionary processes." However, it was formed only in the 1990s to bring together various subgroups of different EC paradigms which had emerged during the previous decades, most importantly evolutionary programming (EP) (see section 2.3.2), evolution strategies (ESs) (see section 2.3.1) and genetic algorithms (GAs) (see section 2.3.3). Also during the 1990s, (Bäck, Fogel, and Michalewicz 1997) was produced as the "first clear and cohesive description of the field" (Bäck, Fogel, and Michalewicz 1997, p. vii).

Neither (Bäck 1996) nor (Bäck, Fogel, and Michalewicz 1997) offer a clear distinction between EC and evolutionary algorithms–it seems that evolutionary algorithms refer to algorithms used in the field of EC.

What really connects the three EC paradigms of EP, ES and GA, is the simulation of an evolutionary process (see section 2.2), i.e. “the reproduction, random variation, competition, and selection of contending individuals in a population” (Bäck, Fogel, and Michalewicz 1997, p. A1.1:1), to solve an optimization problem.

Reasons for the simulation of evolutionary processes come primarily from the fields of optimization, robust adaptation, machine intelligence and biology and “The ultimate answer to the question ’Why simulate evolution?’ lies in the lack of good alternatives.” (Bäck, Fogel, and Michalewicz 1997, p. A1.1:2)

2.1.2 No-free-lunch theorem

When comparing possible solutions for optimization problems, one comes across the no-free-lunch (NFL) theorem, first formally introduced in (Wolpert and Macready 1995) and (Wolpert and Macready 1997). It states that there is no one optimization algorithm which outperforms all others for all possible optimization problems. Instead, it is claimed “that all algorithms that search for an extremum of a cost function perform exactly the same, when averaged over all possible cost functions” (Wolpert and Macready 1995, p. 1).

Hence, there are cases in which evolutionary algorithms perform better than other algorithms but just as many cases in which they perform worse. By no means do evolutionary algorithms offer a universal solution to all optimization problems, as some expected them to in their early days. That said, there are many different evolutionary algorithms and most of them can be tailored to specific problems via configuration of parameters or operators.

2.1.3 Machine Learning

Machine learning (ML), a term, according to (Banzhaf et al. 1998), coined by Samuel in 1959, is generally used to describe a field of computer science in which

the computer is given the ability to learn. Samuel initially used the word to mean computers programming themselves, but as this proved to be too difficult the focus shifted towards algorithms which automatically improve through experience, also called self-adapting or self-optimizing algorithms. Thus, ML is a much broader term that includes EC and all other kinds of self-adapting or self-optimizing algorithms.

With GP and its ability to evolve complex computer programs, the field has returned to Samuel's initial understanding of ML, even if computers are still far away from programming themselves (Banzhaf et al. 1998).

2.2 The evolutionary algorithm

Evolution, as defined by the neo-Darwinian paradigm, is "the inevitable outcome of the interaction of four essential processes: reproduction, competition, mutation and selection" (Bäck, Fogel, and Michalewicz 1997, p. A2.1:1).

Reproduction, in terms of EC, happens by transferring an individual's genetic information to progeny, either asexually, just copying the individual, or sexually, by performing *recombination*. Genetic information is usually subject to *mutation*, which serves to (re)introduce greater genetic diversity. Competition is simulated by using a fixed population size and *selection* is used to determine which individuals survive and/or reproduce.

Individuals are defined by their genetic program, or *genotype*. The genotype is translated into the *phenotype*, which is used to determine the *fitness* of individuals, which in turn is used in the selection process. The genotype-phenotype mapping is often subject to *pleiotropy* (i.e., a single gene influences several phenotypic traits) and *polygeny* (i.e., several genes influence a single phenotypic trait) so that there is no *strong causality* for changes in the genotype and in the phenotype (Bäck 1996).

Selection is performed, as mentioned before, only on the phenotypes, while reproduction is performed only on the genotypes of individuals.

Evolution is an optimization process, and, as stated by the NFL (see section 2.1.2), is more useful than other optimization processes for some problems, and less useful for others; it is, like other optimization processes, not a process leading to

perfection (for most problems) (Bäck, Fogel, and Michalewicz 1997).

In the following, we will look at the most basic process shared by all evolutionary algorithms. The algorithm consists of four steps (Atmar 1994):

Step 1: An initial population of predefined size is randomly created. Alternatively, the initial population may be *seeded* with specific individuals. Seeding can be used to continue an earlier evolutionary process or to accelerate the optimization process by providing individuals which are known to have a higher fitness score than the average, randomly generated individual or to ascertain a higher genetic diversity in the initial population than completely random initialization offers.

Step 2: In this step, the population is replicated. Replication happens by applying one or more genetic operators on the existing population. Examples for such operators are mutation, recombination or reproduction (explained in more detail in section 2.4.4).

Step 3: The individuals of the population are evaluated by a fitness function and selection is performed. Sometimes, selection is performed before replication and sometimes the genotype has to be translated into a phenotype. Through selection, the new population is determined. In some algorithms, new individuals are chosen from both parents and children, in others only the children are considered. More in selection in section 2.4.3.

Step 4: Repetition. Steps 2 and 3 are repeated until one of potentially several termination criteria is met. Criteria might be a maximum number of iterations or the quality of the solution.

Figure 2.1 shows the four steps of the evolutionary process as explained above.

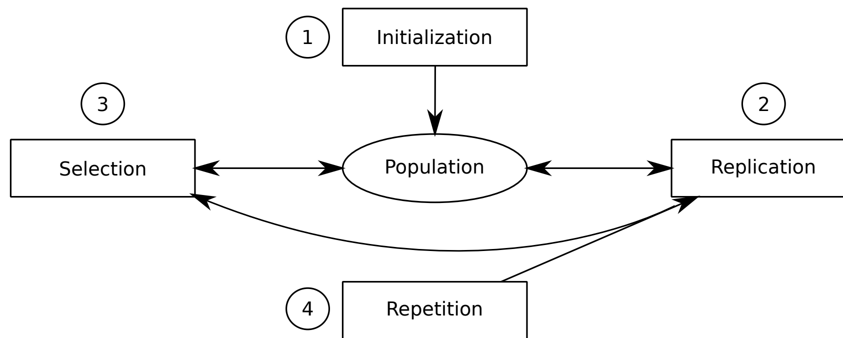


Figure 2.1: The four steps of the evolutionary process.

2.3 Techniques

This section will shortly explain the three techniques of evolution strategies, evolutionary programming and genetic algorithms, before section 2.4 will introduce genetic programming in greater detail.

2.3.1 Evolution strategies

Evolution strategies (ESs) were first developed in the 1960s by Bienert, Rechenberg and Schwefel. They were created to cope with the "impossibility to describe and solve [certain] optimization problems analytically or by using traditional methods" (Bäck 1996, p. 67) by introducing random mutations that were meant to imitate mutations found in nature. The first ES was the so called $(1+1)$-ES, in which only one individual exists which is mutated and evaluated (one of the first applications was in (Rechenberg 1965)). If the new individual performs better than the previous one, it replaces the old individual. This process is repeated until a satisfying solution has been found or some other termination criterion is met. As representation, a real-valued vector of object variables is used, mutation happens by adding normally distributed random numbers (Schwefel 1965). The first multi-membered ES, denoted as $(\mu+1)$-ES, was later introduced by Rechenberg and was the first to use the population principle. Here, two parents are involved in the creation of one child, allowing for the imitation of sexual reproduction, for which a new genetic operator is introduced: recombination. For each vector component of the child, one of its parents'

components is randomly chosen. The parents involved in the recombination process are chosen randomly from the population. The selection operator then removes the worst individuals form the population–regardless of whether they are parents or children–to restore the population size of μ. Mutation happens in the same way as in the $(1+1)$-ES. Lastly, $(\mu+\lambda)$-ES and (μ,λ)-ES were introduced "to make use of [...] parallel computers, and [...] to enable self-adaptation of strategic parameters like the [...] standard deviation of the mutations" (Bäck, Hoffmeister, and Schwefel 1991). They are based on the $(\mu+1)$-ES, which was never widely used and were described in (Schwefel 1975; Schwefel 1977; Schwefel 1981). In the $(\mu+\lambda)$-ES, the recombination process produces λ children, and the new population is again selected from both parents and offspring. To counter the effects of individuals surviving several generations (as described in (Bäck, Hoffmeister, and Schwefel 1991, p. 4)), (μ,λ)-ES was investigated by Schwefel. Here, the lifetime of individuals is limited to one generation and selection is performed only on the offspring, thus avoiding the long survival of misadapted (in terms of their strategy parameters) individuals and stagnations of the population fitness. In both strategies, strategic parameters were added to the evolving variables to allow for self-adaptation and evolution of better strategy parameters over time (Schwefel 1987). Evolution strategies, especially (μ,λ)-ES, are still widely used today.

More on ES can be gathered from (Bäck, Hoffmeister, and Schwefel 1991; Bäck 1996; Bäck, Fogel, and Michalewicz 1997) and the literature referenced therein.

2.3.2 Evolutionary programming

Introduced by Fogel in 1964, evolutionary programming (EP) is based on finite-state-machines (FSMs) which are used to match an input sequence to a certain output sequence. The fitness of an individual is measured by the percentage of correct output values. The individuals (first a (1+1)-selection was used which was later extended to a $(\mu+\lambda)$-selection to avoid becoming stuck in local optima) were mutated slightly and selection was performed among parents and offspring. No recombination is used.

EP did not receive much acknowledgment until about a decade later, when GAs and ES were developed. According to (Bäck 1996), reasons for this included missing computing power and earlier disappointments in the field, which ultimately resulted in a stagnation of development in the field of evolutionary algorithms until the 70s.

It was not until the 1980s, when the initial creator's son, D. B. Fogel, continued the development of EP into a direction that made it turn out very similar to ES: normally distributed mutations and self-adaptation by encoding mutation parameters in the genotypes.

For selection, tournament selection (see section 2.4.3) is used to reduce the population size from 2μ to μ (2μ because each individual is mutated to create one child) probabilistically.

For more information on EP, see (Bäck 1996; Bäck, Fogel, and Michalewicz 1997; Bäck, Fogel, and Michalewicz 1999).

2.3.3 Genetic algorithms

Genetic algorithms (GAs) were first formally introduced in (Holland 1975). They differ from other evolutionary algorithms primarily in three ways: Their representation (bitstrings), their selection method (proportionate selection) and their primary genetic operator (crossover).

There have, however, been many variations of GAs since, which have used different forms of selection and representations other than bitstrings, so that only the focus on crossover remains as uniquely identifying (Bäck, Fogel, and Michalewicz 1997, B1.2:1).

In its original form as proposed by Holland, GAs use bit strings of fixed length as representation. When evaluating individuals, segments of bits (usually of equal length) are interpreted as integers which are then used in the context of the problem which is to be solved. The algorithm works mostly as described in section 2.2. After evaluation, individuals are chosen for crossover probabilistically based on their fitness value. The idea behind this it to combine beneficial traits of parents by creating offspring who is basically a mix of both parents' genes. There are different

kinds of crossover: the simple one-point crossover, which exchanges all bits to the right of the crossover point; and the more powerful multi-point crossover, which exchanges bits between two points–several segments can be formed and exchanged this way. Uniform crossover randomly decides for each bit from which parent it is taken. Mutation is mostly used as a background operator to reintroduce traits that were lost through convergence in the population (Bäck 1996).

2.4 Genetic programming

> "Genetic programming (GP) is an evolutionary computation (EC) technique that automatically solves problems without requiring the user to know or specify the form or structure of the solution in advance. At the most abstract level GP is a systematic, domain-independent method for getting computers to solve problems automatically starting from a high-level statement of what needs to be done" (Poli, Langdon, and McPhee 2008, p. 1).

In GP, computer programs are evolved using the basic evolutionary concepts and processes of population initialization (see section 2.4.2), selection (see section 2.4.3) and reproduction (see section 2.4.4). It is very similar to GA regarding the basic algorithm: a random population is generated, individuals are evaluated, crossover (and, optionally, mutation) is performed and selection determines which individuals may reproduce.

The fundamental difference between GP and GAs lies in the representation: While GAs have been implemented using many different representations (bit strings only being the first), most of these representations encode domain knowledge into the genotype by using a fixed size and / or structure. These limitations make exploring the theoretical foundations easier, but do not permit an application to different problems–for each problem a new representation has to be devised.

John Koza's idea was that "for many problems in machine learning and artificial intelligence, the most natural representation for a solution is a computer program (i.e., a hierarchical composition of primitive functions and terminals) of indeterminate

size and shape, as opposed to character strings whose size has been determined in advance."

The initial–and still widely used–representation for such program structures were trees (see section 2.4.1). Other representations have been used in Linear GP (linear lists of commands, (Banzhaf et al. 1998)) and Cartesian GP (graphs, (Miller 2011)), but we will focus on and use tree-based GP in the implementations presented in this thesis.

2.4.1 Representation

Genetic information is stored in genotypes on which genetic operations like crossover, mutation and reproduction (see section 2.4.4) can be performed. These are then translated into phenotypes. An example for this process could be images (phenotype) that are generated from an arithmetic expression or function (genotype).

There are different ways to represent programs in GP, the most common of which is the tree representation, where hierarchical structures are encoded in syntax trees. An example for such a syntax tree is shown in figure 2.2. Other, less common representations are linear lists, in which a sequence of commands is stored and more general graph representations that allow for even more flexible structures than trees (used, for example, in cartesian genetic programming (Miller 2011)).

When programs are represented by syntax trees, the leaves consist of variables, constants and functions that do not take arguments. They are called *terminals*. Other operations and commands are called *functions*, and together they form the *primitive set* (see section 2.4.5).

In more complex scenarios, several such trees are used in one program, they are then called *branches*. The *architecture* of a program is defined by the number and types of such branches (Poli, Langdon, and McPhee 2008).

Automatically defined functions (ADFs) as first introduced in (Koza 1994) offer a method of evolving reusable components in GP systems. When using ADFs, the programs are split into function-defining branches (ADFs) and result-producing branches (RPB). The ADFs can then be called in the RPB (and they can also call each

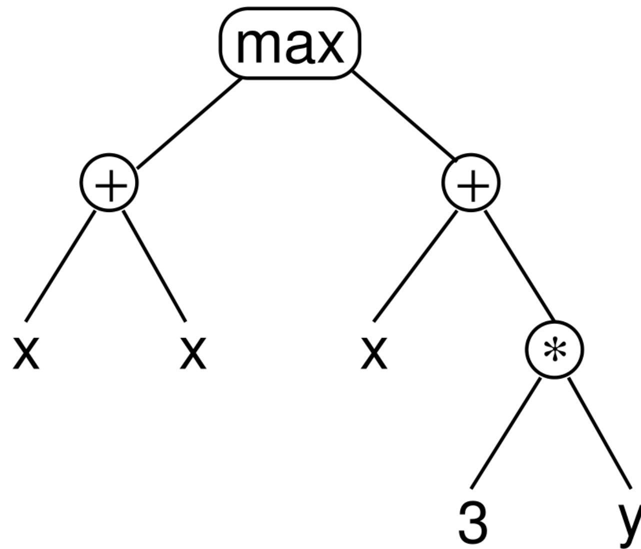


Figure 2.2: GP syntax tree representing $max(x+x, x+3*y)$. Source: (Poli, Langdon, and McPhee 2008)

other) and are evolved in the same way as the RPB. There are different approaches as to where ADFs are stored. While Koza stored ADFs separately for each individual, other approaches offer a global pool of ADFs which are evolved alongside the individuals.

2.4.2 Population initialization

Generally, population initialization can be random, or it can be seeded (or a combination of both). Seeding can happen in different ways. It can be the initialization of a random number generator with a seed to repeat an earlier evolutionary run, or it can mean that the initial population is (partly) filled with existing individuals. It is done for example to continue earlier evolutionary runs, or to ensure a suitable genetic diversity is given.

Population initialization in GP usually happens randomly, similar to other evolutionary algorithms. However, it is not as trivial as in GAs, where a coin can be thrown for each bit string position to generate random individuals, or as in ES, where random number generators can directly be used to initialize random vectors for individuals.

In GP, trees have to be generated. An initialization algorithm usually expects at least one configuration parameters: The maximum initial tree depth. In the following, three simple and commonly used initialization methods shall be described: *full*, *grow*, and their combination, *ramped half-and-half*.

The *full* method randomly picks functions from the primitive set until the maximum tree depth is reached. After that, only terminals can be chosen. As a consequence, all leaves are at the same depth, somewhat limiting the possible shapes and structures of generated trees. Figure 2.3 shows an example of a tree generated by the full method.

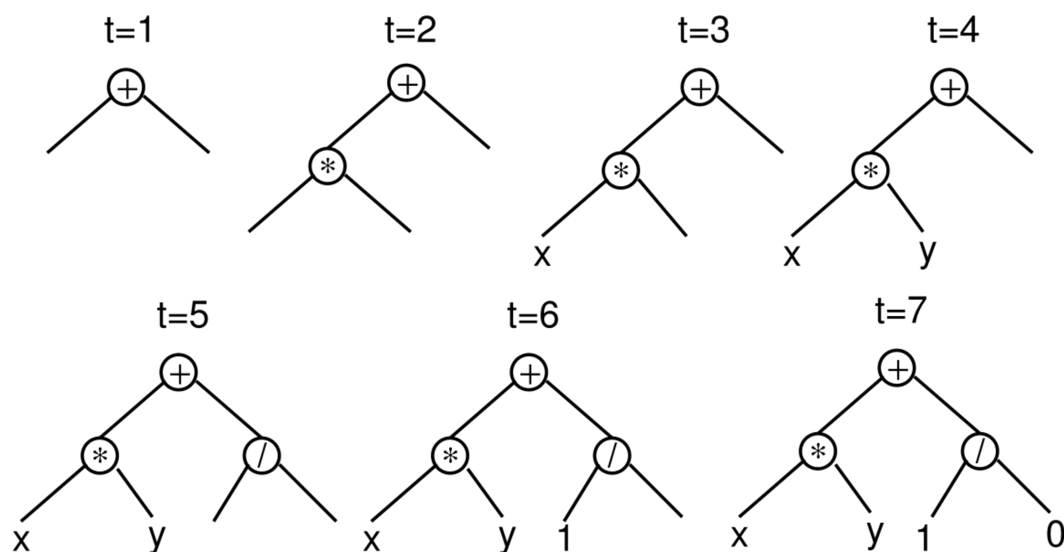


Figure 2.3: Creation of a tree using the *full* method and a maximum depth of 2 (t = time). Source: (Poli, Langdon, and McPhee 2008)

The *grow* method provides a little more variety in the generated trees by randomly choosing from the entire primitive set until the maximum tree depth is reached, and then, just like the full method, only from the terminals. Thus, terminals can appear at different depths of the tree, and trees might not reach the maximum depth at all. Figure 2.4 shows an example of a tree generated by the grow algorithm.

(Koza 1992) introduced a third method, which combines the full and grow methods and is called *ramped half-and-half*. On their own, neither method provides enough variety in the generated trees, ramped half-and-half uses the grow method

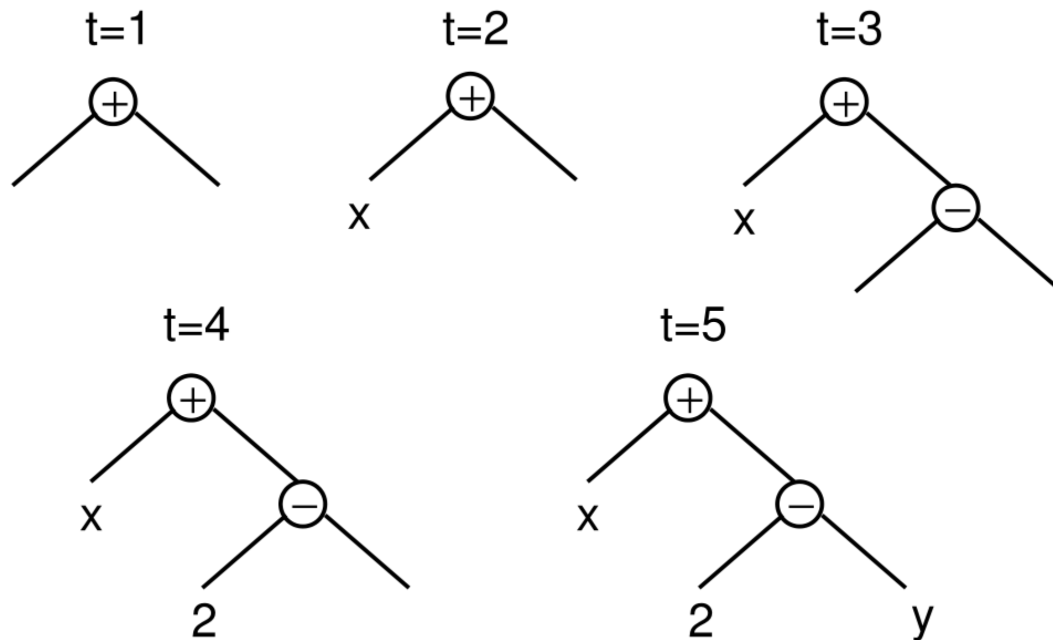


Figure 2.4: Creation of a tree using the *grow* method and a maximum depth of 2 (t = time). At t = 2, a terminal is chosen, closing off that branch of the tree before it has reached the maximum depth. Source: (Poli, Langdon, and McPhee 2008)

for one half of the population, and full for the other. To ensure varying tree depths, different depth limits are used.

Additional initialization methods as well as *seeding* are described in (Poli, Langdon, and McPhee 2008).

2.4.3 Selection

Selection is an integral part of every evolutionary algorithm, for without it, no evolution could happen, there would be no "survival of the fittest" and no improvement in the fitness values of the population.

Normally, selection (using one of the below-mentioned selection strategies) is performed for each free spot in the new population and stochastically selects individuals based on their fitness value. When an individual has been chosen, the genetic operator (see section 2.4.4) to be used on it will be determined according to configuration parameters. Examples would be recombination (at least two individuals

are involved in creating offspring), where selection would again be performed to determine the other participant(s), mutation, where the individual is randomly changed, or reproduction, where an individual is copied to the new population without changes (Poli, Langdon, and McPhee 2008).

Elitism is a selection mode in which a certain number of individuals are not stochastically chosen but instead purely by their fitness value, effectively avoiding to lose the best individuals.

In *steady-state GP*, only one population is held in memory and newly generated individuals are directly integrated into the population and are available for reproduction instantly (Banzhaf et al. 1998, pp. 134).

Selection strategies

Selection is usually performed on phenotypes (the process of creating phenotypes from genotypes is explained in section 2.4.1). The evaluation is based on a fitness value that represents how well a certain program performs a certain task and it affects the chance of the individual being chosen for reproduction or recombination, i.e., the chance of the individual having offspring (or, in the case of reproduction, of its survival).

The two most commonly used selection strategies are *tournament selection* and *roulette wheel selection* (also called fitness proportionate selection). Tournament selection randomly picks a predefined number of individuals from the population. Their fitness values are compared and the best one is chosen and inserted into the mating pool. This selection tournament is repeated as often as the genetic operator requires (for recombination, two selection tournaments are held to choose the two parent individuals). An increased number of tournament participants increases the selection pressure (i.e., the likelihood of weaker individuals being chosen decreases). Roulette wheel selection ensures that individuals are chosen proportional to their fitness value. Each individual is assigned a part of an imaginary roulette wheel the size of which is proportional to its relative fitness value. Thus, individuals with a higher fitness value receive a bigger part of the roulette wheel and have a higher chance of being chosen. The main difference between these two strategies is

that roulette wheel selection imposes a greater selection pressure, i.e., individuals with a much higher fitness value will quickly conquer the whole population and greatly reduce diversity that way. Tournament selection, on the other hand, keeps the selection pressure constant and introduces a noise due to the random selection of tournament participants (Miller and Goldberg 1995). Furthermore, tournament selection ignores absolute fitness values, i.e. it does not matter how much better an individual is compared to another individual. For roulette wheel selection, absolute fitness values are relevant because individuals receive a portion of the roulette wheel according to their fitness value. Great differences in fitness values can lead to undesired results as described above (Bäck 1996).

Fitness function

It was mentioned that selection is an integral part of an evolutionary algorithm, and that it depends on a “fitness value” to select individuals. But how is this fitness value determined?

For this, every optimization problem needs a fitness function or fitness measure. The fitness function determines the quality of a possible solution and a good fitness function is crucial to the success of an evolutionary optimization process. Fitness values are mostly numeric and either represent the success (i.e., higher is better) or the error (i.e., lower is better) of a solution. A fitness function can be visualized as a fitness landscape over the search space (Poli, Langdon, and McPhee 2008).

Fitness functions normally do not operate on the genotype of an individual (i.e., the syntax tree), but on the phenotype (e.g., executable program). This translation from genotype to phenotype has to be performed prior to the fitness evaluation of individuals. When dealing with genotypes that directly represent executable programs, the translation process contains only compiling and running or interpreting the programs, but some GP systems incorporate a hierarchical translation process similar to that found in nature, in which the genotypes are (sometimes in several steps of varying complexity) transformed into phenotypes. The fitness measure then only takes the phenotypes and their behavior into account. A mapping between phenotype and genotype is maintained to allow the assignment of fitness values to

genotypes, which are then used during selection.

Depending on the translation process, problems can arise when dealing with changes to the genotype which cause unexpected changes to the phenotype (*weak causality*). E.g., small changes to the genotype cause big changes in the phenotype and vice versa. Such problems hamper evolutionary progress as they create discontinuous fitness landscapes (Koza 1994; Poli, Langdon, and McPhee 2008).

Multi-objective GP

> "In a multi-objective optimization (MOO) problem, one optimises with respect to several goals or fitness functions f1, f2, ... The task of a MOO algorithm is to find solutions that are optimal, or at least acceptable, according to all the criteria simultaneously" (Poli, Langdon, and McPhee 2008, p. 75).

There are different ways of considering several fitness functions. The first and easiest is an *aggregate scalar fitness function* (Poli, Langdon, and McPhee 2008, p. 75), which combines several fitness measures using a weighted sum. The advantage of this approach is the easy integration into any single-objective optimization as the result is a scalar.

There is another, more complex, way of combining several objectives into one fitness measure: using *pareto dominance*. "Given a set of objectives, a solution is said to Pareto dominate another if the first is not inferior to the second in all objectives, and, additionally, there is at least one objective where it is better" (Poli, Langdon, and McPhee 2008, p. 77). The task then is to find non-dominated solutions which exist on the *pareto front* (see figure 2.5).

There are several algorithms to incorporate pareto ranking into the selection process of an evolutionary algorithm, one of them is the *nondominated sorting genetic algorithm II* (NSGA II), which is described in (Deb et al. 2002). It provides a selection operator which selects individuals based on fitness and spread (to avoid favoring certain fitness measures) and provides a fast pareto sorting algorithm. Furthermore, dominance is redefined to speed up the sorting process and elitism is

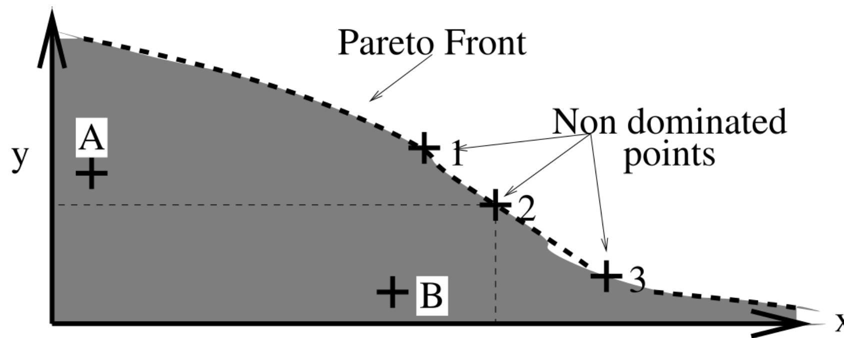


Figure 2.5: Example of Pareto optimality in two dimensions. Solution A and B do not dominate each other, solutions 1, 2 and 3 dominate B, and A is dominated by solution 1. Source: (Poli, Langdon, and McPhee 2008)

supported.

Multi-objective GP has often been used to counter *bloat* (Bleuler et al. 2001). Bloat describes the phenomenon of increased program sizes without any gain in functionality (and, by extension, fitness) in later stages of evolution. This increased incorporation of *introns* (units without functionality) can be explained by the decreased risk of losing functionality upon crossover (fewer introns mean a higher chance of destroying functionality through crossover). Thus, many systems have included the size of programs as an additional optimization objective (Veldhuizen and Lamont 2000).

2.4.4 Genetic Operators

The most important genetic operators in GP are crossover and mutation. For crossover, random parts of the parent programs are combined into a new child program. Mutation randomly changes parts of a program to create a new child program. In GP, crossover is more important than mutation and even though mutation helps the evolutionary process (by reintroducing lost functions), it is not strictly necessary if the initial population contains all or most functions and terminals. Systems introduced in (Koza 1992; Koza 1994), for example, do not employ mutation.

Reproduction simply copies an individual to the new population without changing it.

Crossover

Crossover is, similarly to GAs, the central genetic operator of GP. Essentially, it combines several (usually two, but more are possible) parent individuals to one or more children. To avoid positional changes of genes (i.e., subtrees) inherent to random *subtree crossover*, so called *homologous* crossover operators have been introduced, which analyze the structure of parent individuals and only perform crossover on the *common region*, where the individuals have the same shape.

One-point subtree crossover selects a common crossover point within the common region of parent individuals and swaps the subtrees at that position (see figure 2.6).

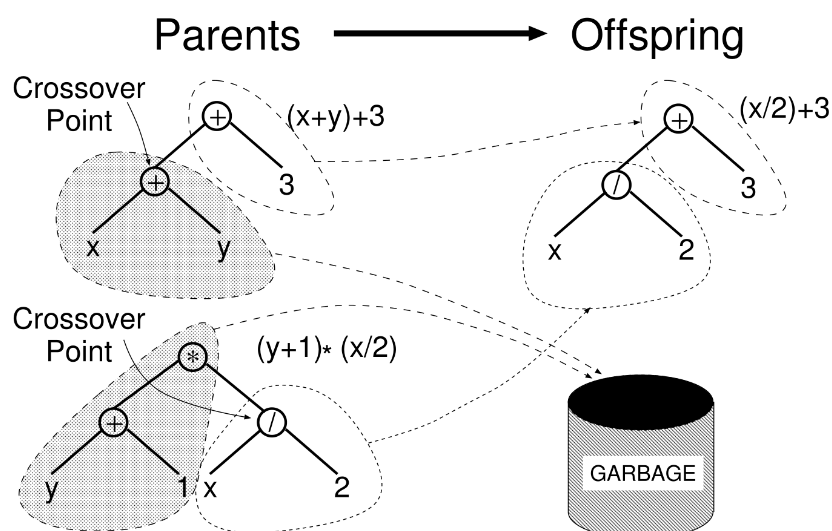


Figure 2.6: Example for one-point subtree crossover, illustrating also that subtrees in the offspring are copies of the original subtrees, leaving the parent individuals unchanged. Source: (Poli, Langdon, and McPhee 2008)

Uniform crossover randomly decides for each node in the common region from which parent it is taken (similar to uniform crossover in GAs).

Other crossover operators exist but they are usually just more sophisticated versions of one of the above (Poli, Langdon, and McPhee 2008).

Mutation

As opposed to GAs, where mutation simply inverts random bits in the bit strings, in GP trees have to be mutated. Some of the most common ways of achieving that shall shortly be described here (Poli, Langdon, and McPhee 2008, p. 42-44).

Subtree mutation simply replaces a random subtree of a solution with another, newly generated, subtree (Koza 1992, p. 106). Figure 2.7 shows an example for subtree mutation.

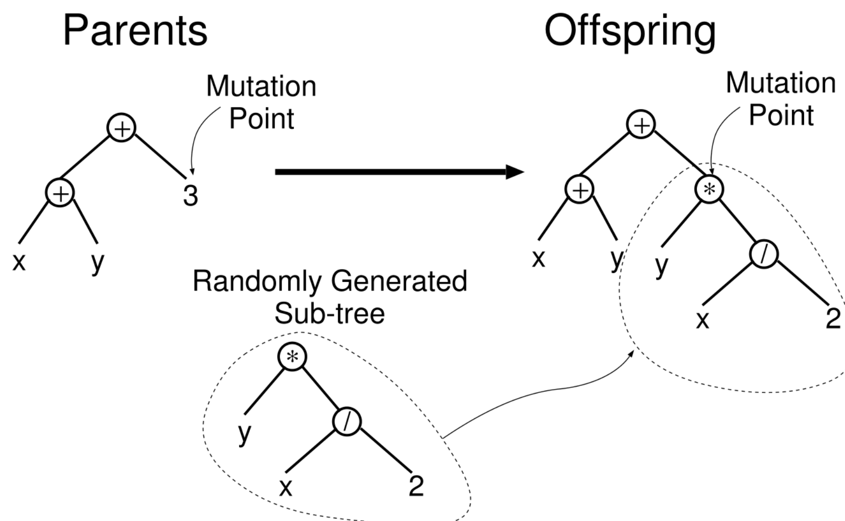


Figure 2.7: Example for subtree mutation where a leaf is selected as the mutation point. Source: (Poli, Langdon, and McPhee 2008)

Node replacement mutation (or *point mutation*) replaces a random node in the program with a compatible (i.e., having the same number of arguments) random node.

Mutating constants has been achieved by adding Gaussian distributed random noise or by systematically trying to optimize constants in a program. Other approaches have been to replace constants with variables and vice versa.

Generally, mutation is used in most GP systems and is seen as advantageous by most, even though it has been shown that GP can work without mutation.

2.4.5 Primitive set

The primitive set of a GP system consists of terminals and the function set. It defines the search space for the solutions and it is important to choose an appropriate primitive set (i.e., it has to contain functions and terminals which enable it to find desired solutions) or else the search might take too long (if the primitive set contains too many elements, most of which may be irrelevant or unhelpful for the problem at hand) or not find a satisfying solution (if the primitive set does not contain the appropriate set of terminals and functions to find such a solution).

The terminal set usually consists of *variables* (which can be changed from the outside of the program, so they are basically the program's inputs), *functions with no arguments* and constants. Sometimes random functions are part of the terminal set, so that each program execution wields a different result, but in most GP systems a program is supposed to return the same result for every execution with the same input values. Random functions are then included in the form of *ephermeral random constants*, a random number that is generated once and stays the same after that.

GP function sets can essentially contain any kind of function usable in a computer program and it usually is defined by the problem domain. Arithmetic operations and programming language structures like conditionals and loops are often found in function sets.

Two important properties of function sets are *closure* and *sufficiency*.

Closure can be broken down into *type consistency* and *evaluation safety*. Type consistency is often required because crossover can create all possible combinations of functions and arguments, i.e. every function must be usable as an argument for any other function. Thus, it is required to either use the same type for all arguments and return values, or provide automatic conversion mechanisms. A different and more flexible approach is the extension of the GP system to use available type information and automatically avoid impossible combinations of nodes. Evaluation safety is necessary because among all possible combinations of functions and arguments there might be (and will be) some which cause errors or exceptions of some kind (e.g., division by 0). One way to avoid such exceptions is using protected versions

of arithmetic operations (which return some default value instead of causing an exception) or catching runtime exceptions and strongly reducing the fitness of the solution (which can be problematic when the relative amount of solutions causing exceptions is high). Another solution would be for the GP framework to discard defect individuals directly.

Sufficiency, on the other hand, goes into the direction of what was stated in the first paragraph of this section: With a sufficient primitive set, it is possible to express a solution to the problem. In most cases, sufficiency is hard to ascertain because no solutions are known beforehand, so it comes down to testing and extending the primitive set as needed (Poli, Langdon, and McPhee 2008, p. 19-23).

2.4.6 Strongly-typed GP

As described in section 2.4.5, type consistency, which is part of closure, is one of the properties a primitive set should have. One way to ensure type consistency is to use only one data type, but that is unreasonably restrictive for most problems. Another solution is to make any function accept all possible types as arguments (possibly handling them in different ways or just throwing exceptions for unsupported types) but this may result in many defect solutions or solutions which contain meaningless combinations of functions and terminals.

One alternative to such approaches is to make the GP system aware of data types and to let it consider them when building parse trees. Such GP systems are *strongly-typed* GP (STGP) systems (Montana 1995).

The first thing STGP introduces is explicit type-awareness. This means every terminal is assigned a type and each function is assigned types for its arguments and return value. Furthermore, the root node's return type is defined by the problem and "each non-root node returns a value of the type required by the parent node as an argument" (Montana 1995, p. 9). Upon generation of new individuals (and also when performing crossover or mutation), the return and parameter types of functions and terminals are considered so that only legal parse trees are created. For crossover, this means that only compatible (in terms of their return type) nodes can be chosen

as crossover points.

Additionally, STGP introduces generic functions and data types which serve as templates for GP functions and data types initially supporting several arguments, return and data types. When they are used they have to be instantiated after which their types are definite.

2.4.7 Reflection and object-oriented GP

When talking about object-oriented programming (OOP), there are two ways to incorporate it into GP. Firstly, one can use it to create GP programs, as done in (Abbott 2003) and ReGeP (described in appendix A.2, and in greater detail in (Rueckert 2013)). Secondly, one can use GP to generate object-oriented code, as done in (Bruce 1995) and (Lucas 2004). Of course, these approaches can be combined to create a completely object-oriented GP system.

In (Abbott 2003) simple Java programs were generated and–via reflection–executed. These programs were object-oriented (in the sense that every Java program is object-oriented), but did not really facilitate object-oriented mechanisms like classes or methods calling each other. Additionally, reflection was used to explore the class library. (Bruce 1995) focused on generating one object and its methods (simultaneously and separately), staying close to the standard GP approaches in terms of representation. (Lucas 2004) used reflection to explore the runtime environment and aimed at generating classes and interfaces instead of the simple programs that GP was able to generate so far.

> “Reflection in an OO language is where a program is able to discover things about itself at run time, such as the class of an object, the fields associated with it, its superclass, the set of interfaces it implements, its set of constructors, and the set of methods that can be invoked on it” (Lucas 2004).

Reflection does not necessarily have to be used in the context of object-oriented GP, but rather in the context of object-oriented programming languages. With it,

methods of classes which are loaded can be determined at runtime and used in the GP process.

Chapter 3

Evolutionary art

3.1 Introduction

"Evolutionary art is a research field where methods from Evolutionary Computation are used to create works of art [. . .]" (Heijer and Eiben 2010a, p. II).

A good overview of the field is given in (Lewis 2008). There have been numerous approaches at both representation and fitness function in evolutionary art (EA), some of which will be introduced in this chapter.

Generally, EA is just another application of evolutionary computation, and it follows the evolutionary process as described in section 2.2.

EA has its origin in (Dawkins 1986) and (Sims 1991) was the first to focus solely on expression-based EA. After that, many artists and scientists have followed them and explored the possibilities of EA.

This chapter will first give an overview of the different representations used in the generation of 2D imagery (section 3.2) and then explore the different ways to determine the fitness of images (section 3.3), with special focus on automatic evaluation (section 3.3.2).

3.2 Representation

In the context of EA, the image is the phenotype that is evaluated. Genotype representations, on the other hand, can take any form, usually involving either a fixed

size set of numbers (as in GAs) or a hierarchy of expressions (as in GP). Genotypes are then transformed to phenotypes through expression, as described in section 2.4.1.

In the following, different kinds of genotype representations will shortly be introduced.

3.2.1 Expression-based representation

(Sims 1991) was among the first to use an expression-based genotype for evolutionary art. He evolved trees of LISP expressions, the function set included “various vector transformations, noise generators, and image processing operations, as well as standard numerical functions” (Sims 1991, p. 2).

In this way, he was able to generate complex images from relatively simple expression trees. Figures 3.1 (listing 3.1 show the genotypes) and 3.2 show some examples of his work.

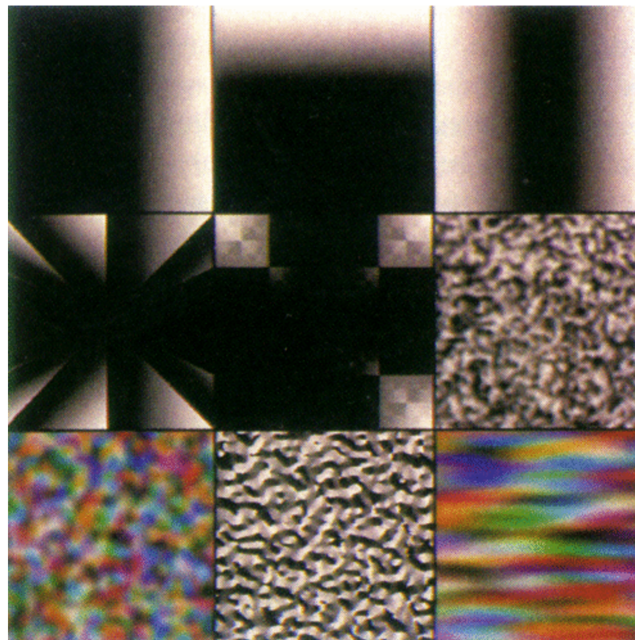


Figure 3.1: Examples for simple expressions. Source: (Sims 1991)

Through the 1990s, many adopted his approach and tried to create EA using different function sets and expression techniques.

Listing 3.1: Genotypes for the images in figure 3.1 (reading left to right, top to bottom). Source: (Sims 1991)

```
a. X
b. Y
c. (abs X)
d. (mod X (abs Y))
e. (and X Y)
f. (bw-noise .2 2)
g. (color-noise .1 2)
h. (grad-direction (bw-noise .15 2) .0 .0)
i. (warped-color-noise (* X .2) Y .1 2)
```

Expression-based genotype representation is one of the most flexible representations, as it does not limit the size or structure of genotypes. It is, however, depending on the function set and parameters, slow in evolving aesthetic images, for the very same reason.

Most expression-based systems using Sims' approach use mathematical functions and only local information to determine pixel colors; it is often possible to recognize from the generated images which types of functions were used in their creation (some examples of such systems are introduced in section 3.3.1.

More details in (Lewis 2008).

3.2.2 Other representations

There have been other representations, including Fractals (using iterated function systems), most notably used in the Electric Sheep project[1], Neural networks, images evolved by processing existing images, and lines and shapes evolved using ant or swarm algorithms (Lewis 2008).

[1] http://www.electricsheep.org/ (visited on 08/25/2013)

Figure 3.2: Example for a more complex image. Source: (Sims 1991)

3.3 Fitness function

There are two basic approaches to fitness functions in EA: First, there is the interactive evaluation, described in section 3.3.1. It defers evaluation to a human user who can, e.g., pick preferred images or evaluate images with a numeric value. These evaluations are then used to evolve the images. This approach was used in many early implementations of EA, because it is easy to implement and the evaluation takes no computation time at all.

Then again, the evaluation takes a lot of time, because humans are slow, quickly fatigued and/or bored, so new ways to evaluate images automatically had to be found. But an automatic aesthetic measure is not a trivial problem, because it has to model a part of human behavior or preference which is not sufficiently understood yet. Quite a few automatic image evaluation algorithms will be described in section 3.3.2, all of which result in a slightly different “kind” of resulting image. The generated images are not quite the same, but they share some traits identifying the algorithm which helped evolve them.

In (McCormack 2005), where open problems in evolutionary music and art are discussed, one of these problems is identified to be the design of “formalized fitness functions that are capable of measuring human aesthetic properties of pheno-

types. These functions must be machine representable and practically computable" (McCormack 2005, p. 5).

Section 3.3.3 will shortly elaborate on multi-objective EA, with which combinations of automatic evaluations can be used or hybrid systems using human and automatic evaluations can be created.

3.3.1 Interactive evaluation

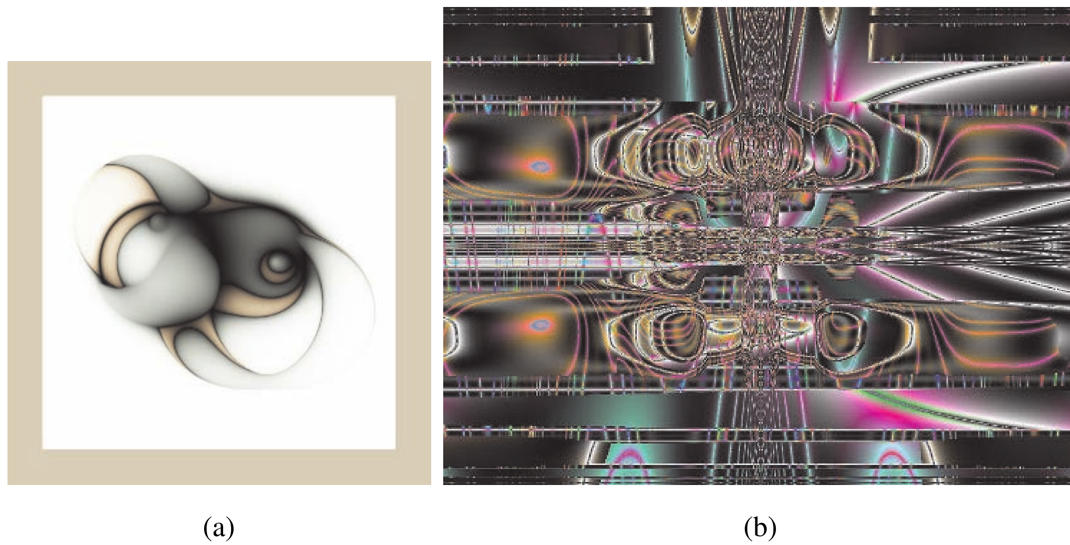
Interactive image evaluation was the first and simplest image evaluation method. Generated images are presented to the user who then has to evaluate them. Evaluation can take different forms, from binary–where the users choose images they like–to arbitrarily stepped evaluation scales (in Jpea, for example, 5 different evaluation levels exist). These evaluations serve as fitness values and are then used to evolve the image population.

Obvious advantages of an interactive evaluation is that there is no computing power needed to evaluate the images (this was of concern especially in the early days of EA) and one does not have to try to model human preferences in an evaluation algorithm.

There are, however, severe disadvantages of interactive evaluation as well: human fatigue and memory capacity limits. Human fatigue means that any interactive evolutionary run can usually only consist of 10-20 generations. The human memory capacity limits only allow for quite small population sizes which have negative impacts on, e.g., genetic diversity.

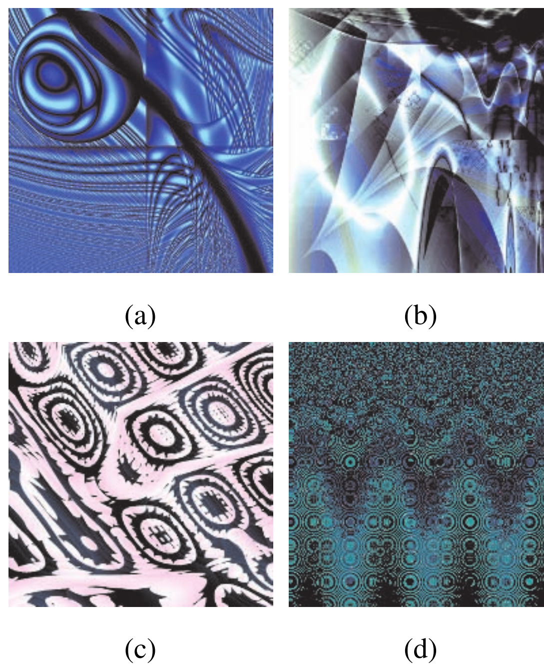
Some examples of systems that use interactive image evaluation include those of Sims (already introduced in section 3.2), Hart, Unemi (see figure 3.3), Gerstmann, McAllister, Day and Mills (see figure 3.4, all of whom used some form of expression-based genotype (as described in section 3.2).

Additionally, systems using interactive evaluation only in support of an automatic evaluation have been developed, and they will shortly be introduced in section 3.3.2 (Takagi 2001).



(a) (b)

Figure 3.3: (a) © 2005 D. A. Hart, (b) © Tatsuo Unemi (image taken from (Lewis 2008))



(a) (b)

(c) (d)

Figure 3.4: (a) © Derek Gerstmann, (b) © David K. McAllister, (c) © Tim Day, (d) © Ashley Mills (images taken from (Lewis 2008)).

3.3.2 Automatic evaluation

The limitations of interactive evaluation (not only in the field of EA) quickly brought on research in the field of automatic evaluation. One of the early approaches at automatic evaluation in EA was introduced in (Baluja, Pomerleau, and Jochem 1994), where neural networks were trained through interactive evaluation and could later perform the evaluation without the user, even on images never encountered before. Several problems were encountered using this approach and the results were not satisfying to a degree where the neural network could be used as a replacement for the human evaluation.

Later approaches tried to generalize aesthetic features in art and incorporate them into aesthetic models which could then be used to determine an image's aesthetic fitness numerically. Some of these algorithms are described in the following sections.

Machado & Cardoso

In (Machado and Cardoso 1998), a system to create visual art was developed, the focus being on the automatic evaluation of grayscale images based on the understanding of human aesthetic judgment at the time.

The authors identified several problems which make the evaluation of visual art challenging. Among them, the fact that visual art theory is not as advanced as in other fields of art (like music) and there is no underlying notation system for visual art.

After going over the emergence and origins of art, they arrive at the conclusion "that the assessment of the visual aesthetic value of an image is directly connected to the visual image perception system" (Machado and Cardoso 1998, p. 4).

They go on to break down the visual value of an image into two components: Processing complexity (PC; where lower is better) and image complexity (IC; where higher is better). For both methods, image compression is used as a measure. For image complexity, JPEG image compression is used. The compressed image is

compared to the original using the quotient

$$IC(I) = \frac{RMSE(I)}{Compression_ratio(I)} \tag{3.1}$$

where RMSE is the root mean square error between the images, and the compression ratio is the ratio between the image sizes before and after compression. The less predictable the pixels of the image are, the greater the IC measure. Fractal image compression, which basically explores self-similarities of an image, is used to measure the processing complexity as it resembles the human image processing mechanism more closely than other compression techniques. Additionally, for PC, the human perception process (i.e., more details are observed the longer the image is being processed) is imitated by compressing and evaluating the image twice–with different compression quality settings. This is not done for IC, as JPEG compression degrades too much for high compression ratios (Machado and Cardoso 1998; Machado and Cardoso 2002; Heijer and Eiben 2010a).

In (Ekárt, Sharma, and Chalakov 2011), different estimates for image complexity and processing complexity are used: Image complexity is defined only through the compression ratio, i.e. the predictability of the pixels, while processing complexity is expressed as the compression ratio of the genotype, so it is rather a measure of genotype complexity.

Figure 3.5 shows a set of example results of the automatic evaluation from (Machado and Cardoso 2002).

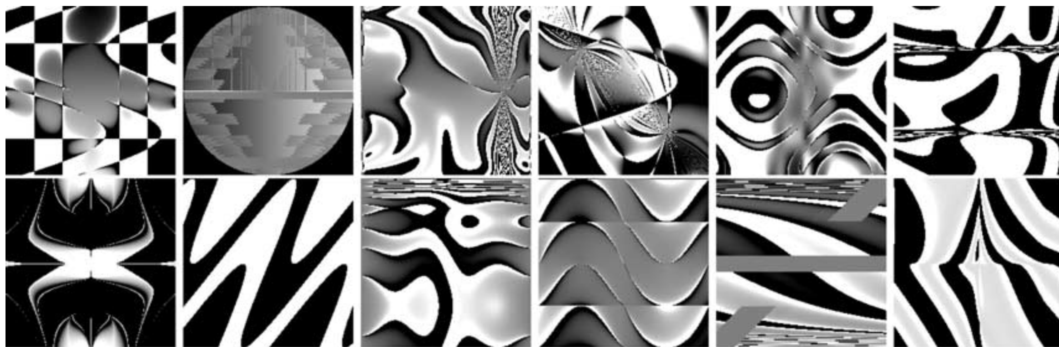
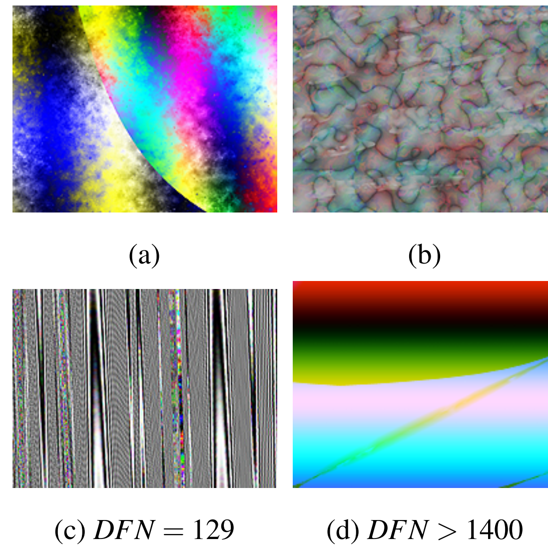


Figure 3.5: Example results from independent evolutionary runs using automatic evaluation. Source: (Machado and Cardoso 2002)



(a) (b)

(c) $DFN = 129$ (d) $DFN > 1400$

Figure 3.6: Image examples of automatic image evaluation using the Ralph aesthetic measure. (a) and (b) were evolved using the same color target, while (c) and (d) are example results for higher DFN values. Source: (Ross, Ralph, and Zong 2006)

More on the implementation in section 5.4.2.

Ross & Ralph

In (Ralph 2006), a mathematical model of aesthetics was introduced, which "found that many works consistently exhibit functions over colour gradients that conform to a normal or bell curve distribution" (Ross, Ralph, and Zong 2006). It analyzes color gradients and mimics human reaction to stimuli by treating measurements logarithmically.

For every pixel, a stimulus and response value is calculated. From the latter, parameters of a normal distribution are derived and the actual distribution of response values is compared to this normal distribution. The smaller this deviation, named *deviation from normality* (DFN), the better distribution of colors in the image and the higher its aesthetic value. Normally, this measure is used as a multi-objective evolution, with the two parameters of the calculated normal distribution and the DFN as objectives.

Figure 3.6 shows some example images.

More details on the mathematic model and the implementation in section 5.4.2.

Fractal dimension

The fractal dimension determines the complexity of an image with a value between 1 (one dimensional) and 2 (two dimensional). In (Spehar et al. 2003), a fractal dimension around 1.35 was determined to be the best in terms of human preference, so in (Heijer and Eiben 2010a) the following aesthetic measure was used:

$$M(I) = max(0, 1 - |1.35 - d(I)|) \tag{3.2}$$

The fractal dimension of images is determined using a box-counting technique. Figure 3.7 shows some examples of fractal dimension evaluation.

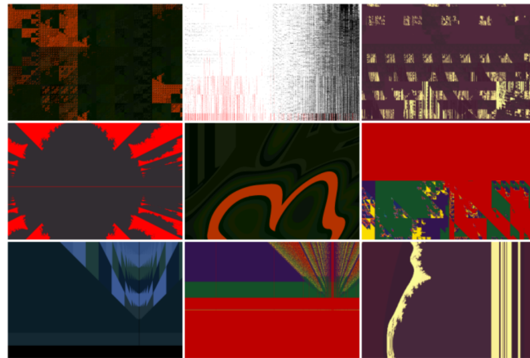


Figure 3.7: Example results from independent evolutionary runs using fractal dimension automatic evaluation. Source: (Heijer and Eiben 2010a)

Benford's law

> "The Benford's law has been proposed in the very past in order to modelize the probability distribution of the first digit in a set of natural numbers" (Jolion 2001).

The law can be used to detect altered data, and will in this context, be used to evaluate the color gradient magnitude against the expected Benford distribution.

In (Heijer and Eiben 2010b), Benford's law was used by creating a histogram of pixel luminance using 9 bins and then comparing the difference between the expected distribution according to the Benford's law and the actual distribution.

Figure 3.8 shows a set of example images evolved using Benford's law.

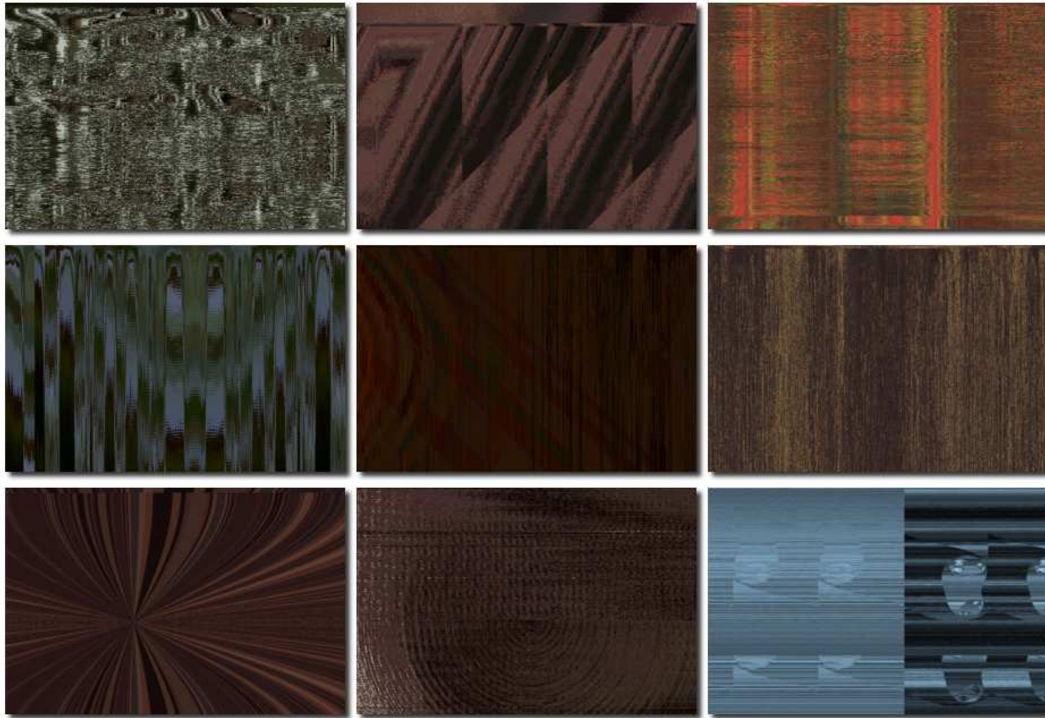


Figure 3.8: Example results for image evolution using Benford's law as aesthetic measure. Source: (Heijer and Eiben 2010b)

Global contrast factor

The global contrast factor (GCF) is a contrast measure which tries to model the human perception of contrast when looking at an image. It computes an overall contrast value by inspecting local contrast factors at different resolutions and building a weighted average. Local contrast is the average difference between neighboring pixels, and in each iteration several pixels are combined into one superpixel, this is repeated until only a few superpixels are left. The local contrasts are then summed up using weight factors (Matković et al. 2005).

Images of lower contrast are considered less interesting than images with higher contrast.

Figure 3.9 shows some examples of GCF-guided image evolution.

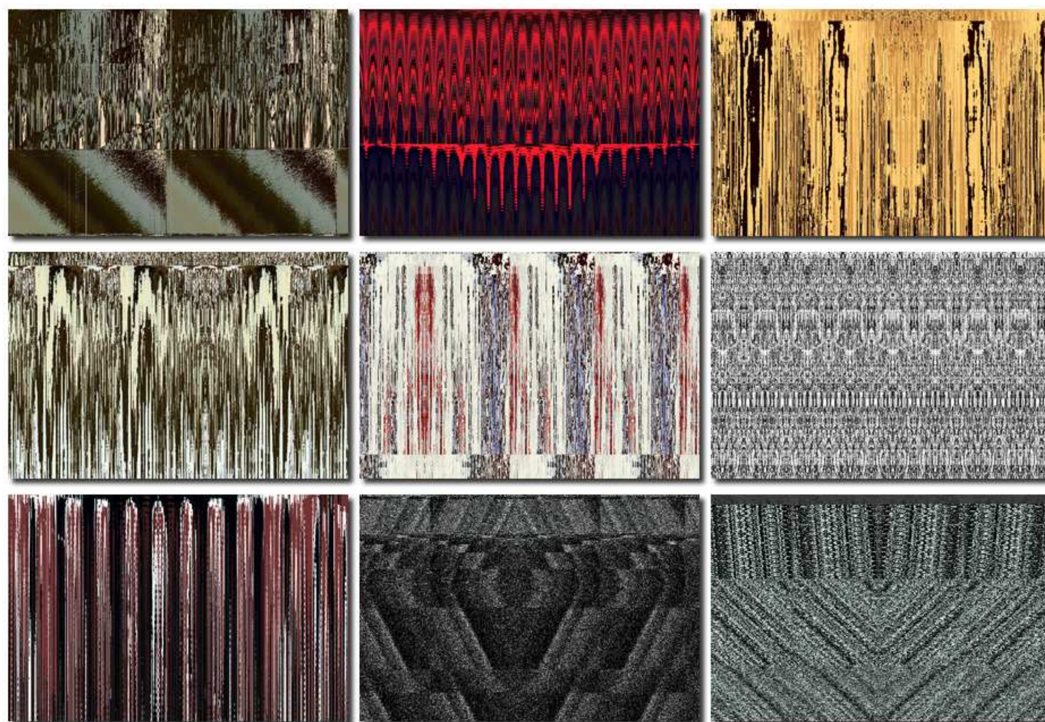


Figure 3.9: Example results of GCF-guided image evolution. Source: (Heijer and Eiben 2010b)

More details on this algorithm and its implementation in section 5.4.2.

Birkhoff measure

The Birkhoff measure generally defines aesthetic value as the quotient between order and complexity. Since its introduction, there have been numerous approaches at defining measures for order and complexity. Some of these approaches, as shortly introduced in (Ekárt, Sharma, and Chalakov 2011) and in more detail in (Rigau, Feixas, and Sbert 2008), tried to used information theory, more specifically Shannon entropy and Kolmogorov complexity, to measure order and complexity.

Figure 3.10 shows some image examples using information theory as the aesthetic measure.

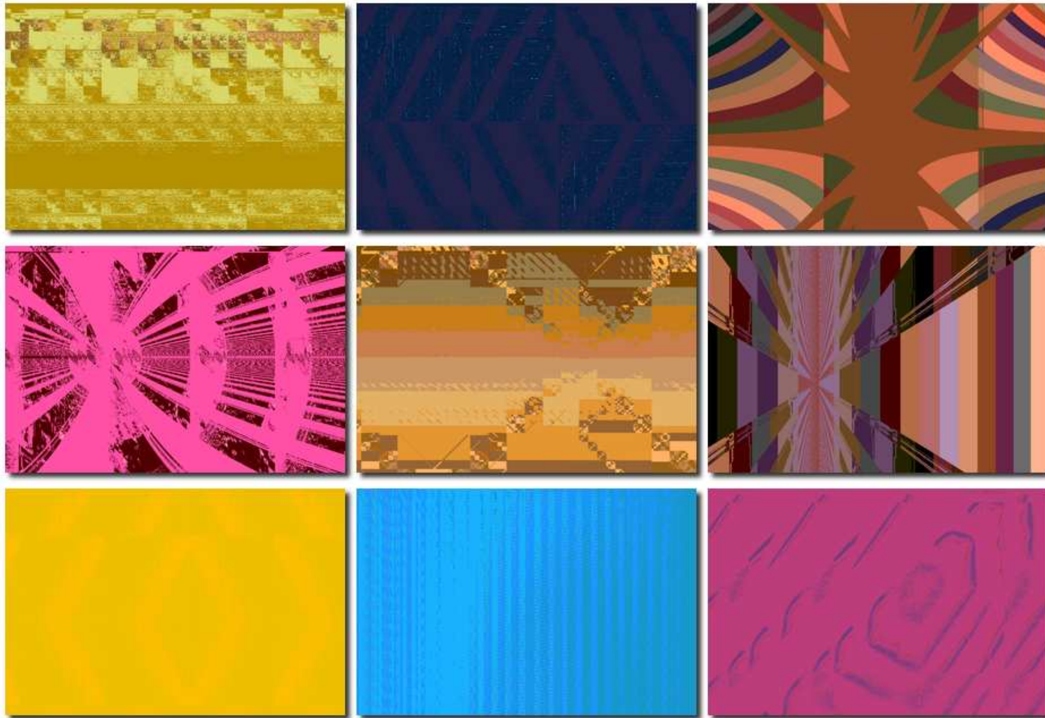


Figure 3.10: Example results with enhanced contrast using information theory as the aesthetic measure. Source: (Heijer and Eiben 2010b)

Target image evolution

The first attempt at target image evaluation in (Rueckert 2013) was a simple pixel color comparison, and it did not–as expected from such a simple solution–produce acceptable results. Generally, one can try to exactly match a target image (which is then a compression problem), or try to create a similar image by comparing specific features of images. In any case, multi-objective optimization (see section 3.3.3) is in order, as measures tend to vary in efficiency at different stages of the target image evolution (e.g., pixel comparisons are ineffective at the beginning but become more usable when the images have a certain degree of similarity, while other measures, which compare more general features, may create images which are indeed similar according to the compared features, but do not look similar).

The problem with pixel comparisons (RMSE or otherwise) is that the structure of an image is not sufficiently compared. E.g., if an image has a dominant color, images which contain large amounts of this color (or even only this color) are favored over

images that may have a different dominant color but similar patterns or structures. This can be alleviated by comparing grayscale version of the images or performing edge detection or similar feature extracting algorithms.

A further problem with this approach is the color space: When using the RGB color space, the distance between colors may cause unexpected color developments. E.g., the distance between green $(0,255,0)$ and black $(0,0,0)$ is the same as between green and red $(255,0,0)$ but much smaller than the distance between green and white $(255,255,255)$. A color space that would work better for pixel comparison is the HSV color space.

Basically, any combination of the aforementioned measures could be used to implement a target image evolution, i.e. the target image would be evaluated using one of the algorithms and the fitness value would be used as the target fitness value for the evolution. (Ross and Zhu 2004; Ross, Ralph, and Zong 2006).

3.3.3 Multi-objective evolutionary art

When optimizing towards multiple objectives, one can either use a weighted sum (or product), or, which is in most cases more suitable, pareto ranking (see section 2.4.3) (Heijer and Eiben 2011; Ross and Zhu 2004; Ross, Ralph, and Zong 2006).

In the context of this work, we will limit ourselves to weighted sums and products when incorporating multiple optimization criteria into the image evolution.

Chapter 4

Reflection-based Genetic Programming (ReGeP)

4.1 Introduction

Reflection-based Genetic Programming (ReGeP) is a software library, intended as an extension to GP frameworks, which was developed and documented in (Rueckert 2013).

Its main purpose is the construction of the primitive set of a GP system by providing

- a wrapping layer which allows decoupling of functions and terminals from the GP system in which they are used
- an infrastructure which offers gradual processing of elements–starting from package or class names and ending with GP function ready for use
- validation of the GP function set so that no unusable functions clutter the primitive set.

Reflection (see section 2.4.7) is used to automatically discover functions usable in a GP system, and there is no need to adjust or re-implement functions before using them.

The following section will give a short system overview and sum up the important classes, interfaces and their association. In section 4.3 extensions of ReGeP will be documented that have been implemented in the context of this thesis.

A more detailed introduction and documentation of ReGeP is given in section A.2 of appendix A.

4.2 Overview

ReGeP consists of two parts: first, the core module `regep-core`. It contains most of the processing infrastructure and will wrap access to elements of the function set (e.g., methods, attributes, and constructors of classes) in an instance of the `Function` interface which represents a general GP function.

As the frameworks in which the functions will be used usually has its own interface or abstract class that GP function have to implement or extend, framework-specific modules are needed to finish the processing. Currently, only the `regep-jgap` module for the JGAP framework (see section A.1.1) exists. It further processes the `Function` instances by wrapping them in a `JgapFunction-Adapter`, which implements the `CommandGene` class of JGAP.

After this, GP functions are stored in a `FunctionDatabase` (see section A.2.2), through which the validated primitive set can be accessed.

4.3 Extending ReGeP

In (Rueckert 2013), ReGeP was given individual classes out of which static methods were extracted for use in the GP function set.

In the context of this thesis, ReGeP has been further developed to work multi-threaded, allow for complete packages to be processed and constructors and instance methods to be used in the GP primitive set in addition to static methods.

The following sections will shortly document and describe reasons for, details of and problems with these extensions.

4.4 Multi-threaded function set creation

Turning the function set creation from single-threaded to multi-threaded first requires independent units of work that can be executed concurrently. This unit of work is the processing of a single object in a single processor. I.e., after the first input object has been accpeted by the processor chain, a task is created to process it. Similarly, tasks are spawned to process each output object, recursively. These tasks are then submitted to a thread pool of configurable size and will be processed concurrently. Doing that, two questions arise:

- How to prevent endless recursions?
- When to terminate the thread pool?

The first question, prevention of endless recursion, had been solved using a set of classes in which every input object's class was recorded and only removed after all the output object had been processed. If, during the processing of one of the output objects, an input object of the same class was encountered, the endless recursion prevention would be triggered, stopping the processing.

This was not a perfect mechanism, because it might happen that object with the same class appear on different levels of the processing hierarchy without causing an endless recursion, but it was a sufficiently simple and, in the case at hand, working mechanism.

However, when processing objects multi-threadedly, this mechanism no longer works, as several branches of the recursion tree are processed in parallel, thus requiring several sets of objects which would have to be attached to the current processing branch–something that is not possible with the current design.

To avoid overcomplication, the easy way out was taken: There is no endless recursion prevention in the multi-threaded function set creation.

The second question is more of an implementation problem. A thread pool is a service which accepts tasks until it is shut down. These tasks are then executed on one of the threads in the thread pool. When processing an object, initially a new thread pool is created and the recursive process is started. After that, the method has

to wait for all tasks to be spawned, shut down the thread pool (after which no new tasks are accepted) and await its termination (which it will only after the currently queued tasks are completed). But how to determine when to shut the thread pool down when there is no way of knowing beforehand how many tasks will be spawned (as the process is recursive)?

The solution is a global spawn counter. This counter is incremented just before a task is spawned (signaling that this task might spawn new child tasks). On the thread of this new task, it is made sure that all new child tasks (i.e., tasks to process the output objects) are spawned before the spawn counter is decremented. Not since all new subtasks have been spawned on the same thread on which the spawn counter is decremented, we know that the spawn counter has been incremented for each newly spawned subtask. Thus, it can only reach 0 after all tasks have been spawned.

4.5 Non-static function set elements

When generating object-oriented programs, it makes sense to be able to create objects and call its methods, which requires constructors and instance methods to be part of the primitive set.

There are, however, disadvantages to incorporating non-instance elements into the function set, the most important being the low performance when using Java reflection. There are libraries like reflectasm[1] which greatly improve the performance for non-static reflection access.

Processors for non-static elements were implemented but are not used in this thesis. They were implemented by using the object on which a method is called as the first argument to the GP function and calling the method using reflection for instance methods. Constructors are GP functions which return the object created by the constructor.

The reason why non-static methods and constructors are not used in this thesis is twofold: First, no object-oriented programs are generated, and most arithmetic functions in the function set (see section 5.5.4) are static functions. Second, the weak

[1] `https://code.google.com/p/reflectasm/` (visited on 08/25/2013)

performance of non-static reflection access (as mentioned above) makes it more interesting to wrap the few cases, in which non-static methods or constructors have to be used, in static methods (as happened with the `Complex` class for fractal image creation).

Chapter 5

Java package for evolutionary art (Jpea)

5.1 Introduction

The Java package for evolutionary art (Jpea) is, primarily, a toolbox for evolutionary art applications. It provides a framework for image creation, evaluation and framework integration, and, on a higher level, for easy application creation.

It was, just like ReGeP, developed in (Rueckert 2013) and has been extended for this thesis. Extensions are documented in section 5.3, a short overview is given in the next section, and a more detailed documentation is given in section A.3.

5.2 Overview

Jpea consists of three major packages: image creation, centered around the `PhenotypeCreator` interface; image evaluation, centered around the `Evaluator` interface and the application package, which offers tools for easier application creation. Additionally, the classes `Individual` and `Population` provide ways of storing individual information (genotype, phenotype, evaluation) and individuals of the current generation.

Image creation, or, more general, phenotype creation, is handled by the `PhenotypeCreator` interface, which is defined by the expected genotype and

the created phenotype. An example for a concrete image creator is the `VectorFunctionImageCreator`, which expects the genotype to return a vector upon execution. The first three components of this vector are treated as RGB components. The genotype is executed for each pixel of the resulting image, the variables inside the genotype are set to the corresponding pixel position (or, more precisely, to a scaled down pixel position) prior to execution.

Image evaluation is handled by the `Evaluator` interface, which is defined by the expected genotype and / or phenotype and the evaluation type (it might just evaluate the phenotype, in this case the genotype is of no relevance). Two examples of image evaluators are the `ComparingImageEvaluator`, which implements a naive pixel comparison, and the `InteractiveImageEvaluator`, which defers evaluation to the human user.

The application package provides so called scenarios (represented by the `Scenario` interface) which consist of an `EvolutionEngine` (containing a `PhenotypeCreator` and `Evaluator` and delegating image creation and evaluation to them) and a `FrameworkDriver` which handles configuration of and communication with the framework.

As in ReGeP, there is a core module (`jpea-core`) and a framework-specific module (`jpea-jgap`) to keep the dependency on a single framework to a minimum.

A more detailed overview can be found in section A.3 of appendix A.

5.3 Extending Jpea

In (Rueckert 2013), image evolution was approaches from two different directions: target image evolution, which, quite frankly, failed, because the fitness function was inadequate; and interactive evolution in which the user guides the evolution.

Neither approach was satisfying. The first for obvious reasons, the second because the population size and generation number is very limited when implementing a user-guided evolution.

In this thesis Jpea will be extended to investigate different possible improvements to these approaches:

- How well do different automatic evaluation algorithms (see section 5.4.2) perform and what kind of images do they produce?
- How does the choice of the GP primitive set affect the generated images and general image evolution success (see section 5.5.4)?
- Are there better ways of representing or interpreting the image genotype and could an improved genotype-phenotype mapping yield better image evolution results (see section 5.5.3)?

In the following sections the new features needed to answer these questions will be documented, before the questions themselves will be answered in chapter 6.

5.4 Image evaluation

5.4.1 Interactive evaluation

In (Rueckert 2013), interactive image evolution was performed on a population of 10 individuals. This resulted in low genetic diversity, an effect that was intensified by the few number of newly generated individuals that could compete with the existing population. To improve the interactive image evolution, several changes have been made.

Firstly, the GUI was redesigned and improved to allow for a population of 20 images which can be shown and evaluated on one screen, increasing the possible genetic diversity in the population.

Secondly, the general genotype-phenotype mapping was improved (see section 5.5.3) and the primitive set tuned to create aesthetic images quicker and more often (see section 5.5.4).

5.4.2 Automatic evaluation

Here, the different automatic evaluation algorithms that were implemented will be described with focus on the implementation. The general principles were described in section 3.3.2.

Machado & Cardoso

As described in section 3.3.2, the aesthetic measure first introduced in (Machado and Cardoso 1998) uses image complexity and processing complexity to evaluate an image:

$$\frac{IC^a}{PC^b} \tag{5.1}$$

Where a and b can be used as weights. Image complexity using JPEG compression was implemented without problems, however, calculating the processing complexity proved to be more difficult.

Image complexity is calculated as follows:

$$IC = \frac{RMSE}{Compressionratio} \tag{5.2}$$

Fractal image compression is an extremely computationally expensive operation, the library *Fractal Image Compression* (F.I.C, see section A.1.2) was used for compression and decompression. Compressing one image at the default settings takes about 3 seconds–too much to be usable in this context. Changing the settings led to faster compression times, but the resulting images were too degraded to be used in an RMSE comparison.

The processing complexity measure used in (Ekárt, Sharma, and Chalakov 2011) (compression ratio of the genotype's string representation) was used instead, although it was simplified by using the number of nodes in the expression tree. To create a smoother genotype complexity measure, PC^b was replaced by

$$\frac{PC}{1000} + 1 \tag{5.3}$$

where PC is the number of nodes in the genotype.

Ross & Ralph

The general concepts of Ralph's aesthetic measure was introduced in section 3.3.2, this section will focus on the mathematical model and implementations as described in (Ross, Ralph, and Zong 2006).

The process can be broken down into three steps as follows.

Step 1: The color gradient or stimulus for each pixel of an image is determined by first calculating for each pixel:

$$|\Delta r_{i,j}|^2 = \frac{\left(r_{i,j} - r_{i+1,j+1}\right)^2 + \left(r_{i+1,j} - r_{i,j+1}\right)^2}{d^2} \tag{5.4}$$

Where i and j are pixel coordinates, r is the red color component and d is 0.1% of half the diagonal length. The same is done for the other color components. The stimulus S is then computed using

$$S_{i,j} = \sqrt{|\Delta r_{i,j}|^2 + |\Delta g_{i,j}|^2 + |\Delta b_{i,j}|^2} \tag{5.5}$$

from which the response R can be calculated with

$$R_{i,j} = log\left(\frac{S_{i,j}}{S_0}\right) \tag{5.6}$$

where S_0 is the detection threshold, which is taken to be 2 in (Ross, Ralph, and Zong 2006).

Step 2: Now the distribution of response values is calculated. For that we first determine the mean (μ) and standard deviation (σ^2) of the normal distribution of R:

$$\mu = \frac{\sum_{i,j}\left(R_{i,j}\right)^2}{\sum_{i,j} R_{i,j}} \qquad \sigma^2 = frac\sum_{i,j} R_{i,j}\left(R_{i,j} - \mu\right)^2 \sum_{i,j} R_{i,j} \tag{5.7}$$

The actual distribution of all pixels $R_{i,j}$ is then calculated by creating a histogram in which each Pixel updates the corresponding bin with its weight $R_{i,j}$.

Step 3: In the last step, the `deviation from normality` (DFN, which is basically the relative entropy) is calculated as the difference between the bell curve distribution and the actual distribution of response values:

$$DFN = 1000\sum p_i log\left(\frac{p_i}{q_i}\right) \tag{5.8}$$

Here, p_i is the observed probability in the ith bin of the histogram and q_i the expected probability according to the normal distribution with the above mean and

standard deviation. Lower DFN means better accordance of the colors to a normal distribution, and, in the scope of this algorithm, a higher aesthetic value of images.

Fractal dimension

The fractal dimension of grayscale images can be estimated using a so called "box counting algorithm" (Li, Du, and Sun 2009). The algorithm was not reimplemented for this thesis, instead an existing implementation was adapted. ImageJFractalDimension[1] is a plugin for ImageJ[2] and it offers fractal dimension estimates based on the SDBC and SBC algorithms (Chen et al. 2001).

Global contrast factor

The general principles of the global contrast factor evaluation algorithm are described in section 3.3.2. Here we will look at the algorithm and the implementation in more detail.

The algorithm can be separated into five steps:

1. Calculate the linear luminance of the original image.
2. The perceptual luminance is computed for the current resolution.
3. Determine the local contrast for each pixel and the average local contrast at the current resolution.
4. Calculate the superpixels for the next resolution.
5. Repeat steps 2-4 for different resolutions and build a weighted sum.

Step 1: The linear luminance l is the gamma corrected ($\gamma = 2.2$) original pixel value k ($k \in \{0, 1, \ldots, 254, 255\}$) scaled to a value between 0 and 1,

$$l = \left(\frac{k}{255}\right)^{\gamma}. \tag{5.9}$$

[1] `https://github.com/perchrh/ImageJFractalDimension` (visited on 08/25/2013)

[2] `http://rsb.info.nih.gov/ij/` (visited on 08/25/2013)

The algorithm was designed for grayscale images, so in our case the formula becomes

$$l = 0.299\left(\frac{r}{255}\right)^{\gamma} + 0.587\left(\frac{g}{255}\right)^{\gamma} + 0.114\left(\frac{b}{255}\right)^{\gamma} \tag{5.10}$$

with r, g and b being the RGB color components. The RGB conversion parameters are used in the grayscale filter (see section A.1.2) and taken from (Union 2011).

Step 2: Now the perceptual luminance L is calculated using

$$L = 100\sqrt{l}. \tag{5.11}$$

Step 3: The local contrast for each pixel is calculated next. For this formula, the image is expected to be "organized as a one dimensional array of row-wise sorted pixels" (Matković et al. 2005, p. 3), w and h are width and height of the image. The local contrast lc_i for pixel i is then:

$$lc_i = \frac{|L_i - L_{i-1}| + |L_i - L_{i+1}| + |L_i - L_{i-w}| + |L_i - L_{i+w}|}{4} \tag{5.12}$$

For pixels at the edges this formula is reduced to the available neighboring pixels (see figure 5.1).

The average local contrast C_i for the current resolution is then

$$C_i = \frac{1}{w \cdot h}\sum_{i=1}^{w \cdot h} lc_i \tag{5.13}$$

Step 4: To compute a new resolution, superpixels are used to combine several pixels into one, using the average linear luminance (which is then converted to perceptual luminance). Figure 5.2 shows the process of creating superpixels at different resolutions.

Step 5: Now the global contrast factor GCF can be calculated using the weighted sum over the local contrasts at N different resolutions with

$$GCF = \sum_{i=1}^{N} w_i \cdot C_i \tag{5.14}$$

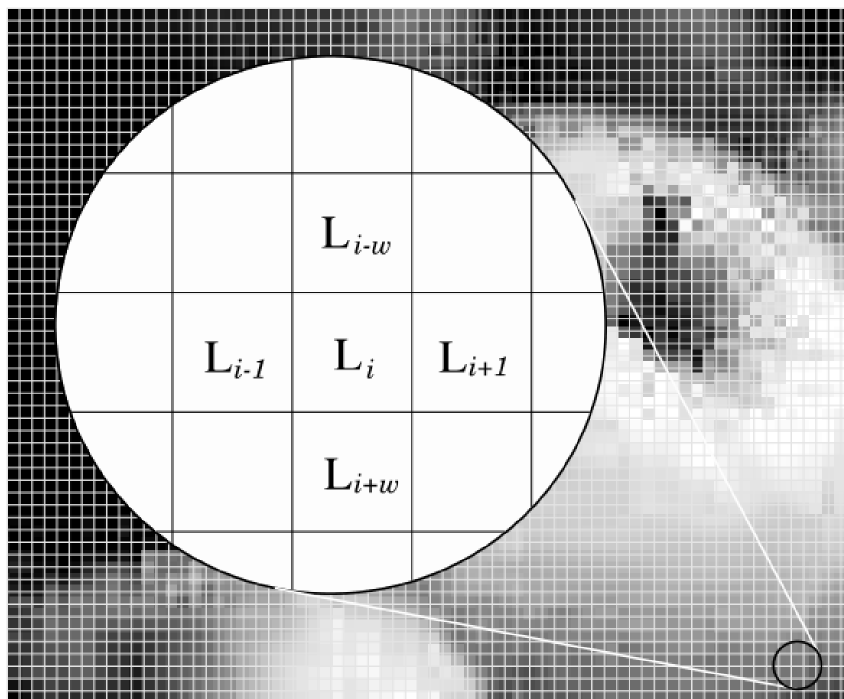


Figure 5.1: The local contrast is calculated using the neighboring pixels. Source: (Matković et al. 2005)

The weight factors w_i ($i \in \{1, 2, \ldots, 8, 9\}$) were approximated in (Matković et al. 2005) as

$$w_i = \left(-.406385\frac{i}{9} + .334573\right) \cdot \frac{i}{9} + .0877526 \tag{5.15}$$

Results of the global contrast factor evaluation are presented in section 6.4.1.

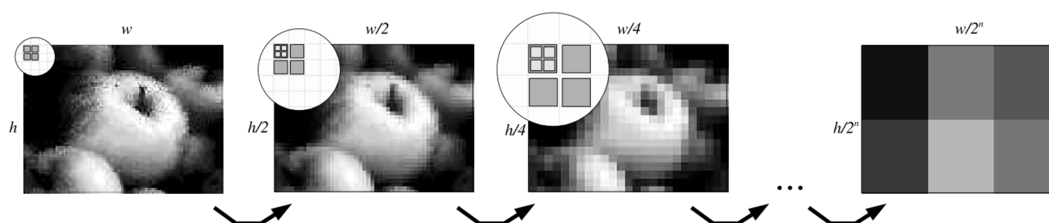


Figure 5.2: Creation of superpixels at different resolutions. Source: (Matković et al. 2005)

Birkhoff measure

The Birkhoff measure was implemented similar to to the BZ aesthetic measure in (Ekárt, Sharma, and Chalakov 2011), i.e.,

$$M_{BZ} = \frac{H_p}{K} \tag{5.16}$$

where H_p is the Shannon entropy which is defined as

$$H_p = -\sum_i p_i \cdot log p_i. \tag{5.17}$$

This formula operates on the luminance histogram of an image, where p_i is the probability in the ith bin of the histogram. K represents the Kolmogorov complexity, which can be seen as the genotype complexity. In (Ekárt, Sharma, and Chalakov 2011) it was expressed as the length of the compressed string of the expression tree representing the genotype. In this thesis, it is simply the number of nodes in the genotype (scaled to avoid crippling the evolutionary process by preventing genotype complexity to increase beyond the absolute minimum).

5.4.3 Multi-objective image evaluation

For multi-objective image evolution, a weighted product is used. To allow for fine-grained tuning of individual fitness measures, a couple of variables are introduced, so that for each fitness measure the fitness values M are transformed to M_t using the following formula:

$$M_t = (M \cdot a + b)^c \tag{5.18}$$

Where a, b and c can be configured for each evaluator.

5.4.4 Multi-threaded image evaluation

Multi-threaded image evaluation is implemented in the abstract superclass `ConcurrentPopulationEvaluator` which spawns a task for every individ-

ual in the `evaluatePopulation(Population)` method and submits it to a thread pool of configurable size.

Additionally, there are several hooks to collect evaluation statistics and support for multiple evaluators.

Usually, the image evaluation depends on nothing but the image which is being evaluated, so multi-threading is straight-forward and unproblematic.

5.5 Image creation

This section describes the different extensions in regards to image creation.

5.5.1 Fractal image creation

Other than in the arithmetic image creation, the images are created within the primitive set. This has the advantage of better encapsulation and simpler genotype structure (the genotype only returns the image, otherwise the genotype would have to return all necessary parameters for the fractal image creation). For fractal image creation, a more general Mandelbrot set recursion is used (also called Multibrot set):

$$z_{i+1} \mapsto z_i^d + c \tag{5.19}$$

The parameters d (any double), the starting value of c (complex number that is taken to be the coordinates of the upper left corner of the generated image) and the step sizes in x and y direction (i.e., the zoom) are parameters to the image creation function, z_0 is set to c.

Instead of the more widely used point escape algorithm (where a point is iterated through the formula until it either leaves a defined stopping radius or the maximum number of iterations is reached; the number of iterations is used to determine the color of the point) for rendering the fractal image, the Lyapunov exponent is plotted instead (Shirriff 1993). This algorithm explores the behavior of points when iterated through the fractal formula, without depending on the stopping radius used in the

point escape algorithm. It uses the Lyapunov exponent, which is calculated here using a simpler version than in (Shirriff 1993):

$$\lambda = \frac{1}{N} ln|z| \tag{5.20}$$

Where N is the number of iterations. “A positive Lyapunov exponent indicates chaotic behavior, and a negative Lyapunov exponent indicates stable behavior” (Shirriff 1993, p. 3).

Example images of this rendering algorithm can be seen in section 6.2.

5.5.2 Multi-threaded image creation

Multi-threaded image creation, similarly to the multi-threaded image evaluation (see section 5.4.4), is implemented in the abstract superclass `ConcurrentPhenotypeCreator` which spawns a task for every individual in the `createForPopulation(Population)` method and submits it to a thread pool of configurable size.

On the genotype-level, it was implemented using thread-local variables to allow several programs using the same variables to be executed at the same time on multiple threads. Thread-local variables basically work as wrappers providing access to underlying instances of the actual variables. For each thread, a separate instance of this underlying variable is created, and all threads access the same wrapper object.

It is a common pattern used to make originally single-threaded software support multi-threading, and the class `ThreadLocal`, which provides a wrapper as described above, is part of the Java standard library.

One problem with multi-threading in this context is that seeding (see section 2.4.2) is no longer possible when the order in which random numbers are generated changes. Seeing how random ephermeral constants are created when they are first needed, this problem arises as soon as multiple images are created parallely, as the order in which the commands are executed on parallel threads, and therefore the order in which random numbers are taken from the random number generator, is not deterministic.

Therefore, Jpea currently does not support seeding. This problem could be solved by providing each thread with its own random number generator, which would have to be reset after each finished task. This, however, would lead to a very limited set of random numbers which would be reused throughout the evolutionary run.

5.5.3 Improving the genotype-phenotype-mapping

The genotype (not including the fractal image genotype) is an expression tree producing a vector with three double valued components. In this expression tree, the variables x and y, representing the coordinates of the pixel for which the expression tree is evaluated, have a special meaning, since they are, on the one hand, the only thing that changes during the creation of an image, thus determining color changes in the image, and, on the other hand, determine the absolute colors depending on their values and the changes of values between pixel coordinates.

The phenotype is an RGB bitmap (possibly encoded into PNG format later). The first approach at mapping genotype to phenotype, as it was implemented in (Rueckert 2013), included the following steps for each pixel (see figure 5.3):

1. Set the variables x and y to the current pixel's coordinates (values from 0..499).
2. Evaluate the expression tree.
3. Interpret the resulting vector's components directly as RGB components, truncating them to the range 0..255.

Both the direct coordinate input and the treatment of the output values have proved problematic when looking at phenotypes in the early stages of evolution–it often took quite a number of generations of aesthetically "boring" (single-colored, very simple color gradients, etc.) images before input and output values were transformed in the expression trees to allow more interesting images.

Generally, for most evolutionary algorithms any kind of scaling or transformation is something that can be discovered through evolution at some point. But considering interactive image evaluation, where user fatigue (Takagi 2001) has to be taken into

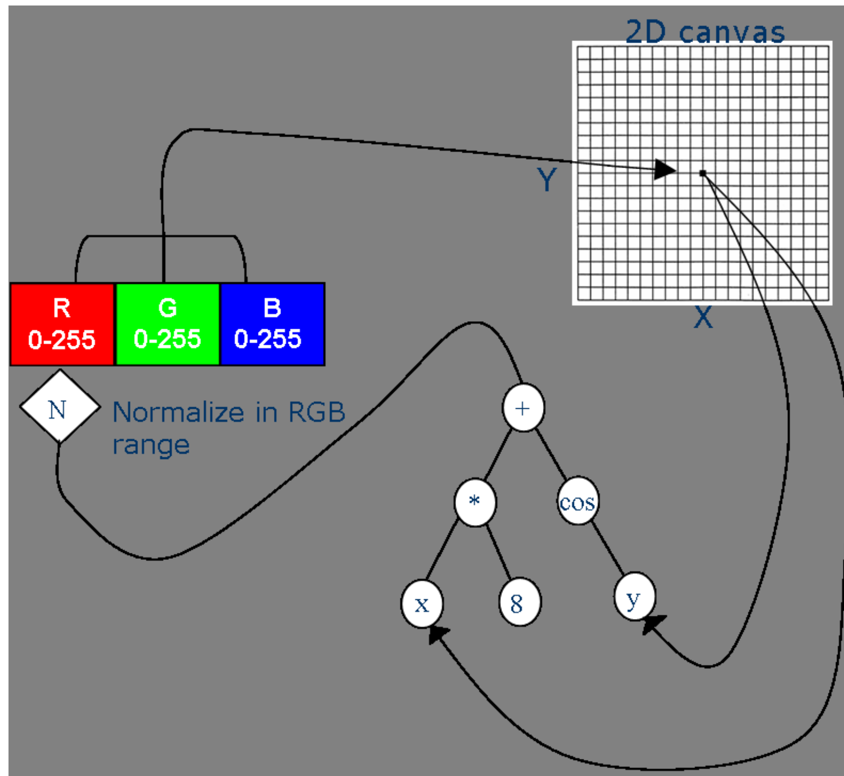


Figure 5.3: Mapping of genotype to phenotype. Source: (Ekárt, Sharma, and Chalakov 2011)

account, these changes can be important to generate aesthetically pleasing images quicker and more often. Thus, two things were changed (Hart 2006):

- **Scaling input variables:** Previously, input variables were taken to be 0..499 for x and y–the unscaled pixel coordinates. Now both variables are scaled to be between $-1..1$.

- **Scaling function output:** Previously, function output was taken as is and just truncated to the range of 0..255, which resulted in many black images (function outputs all close to 0). Now the function output is expected to be in the range of $-1..1$ and is scaled to the color range of 0..255.

As a result of these changes, fewer black or white images are encountered, and aesthetically pleasing images are evolved quicker (because many functions in the primitive set work better with the scaled input values).

Example images that were evolved with the new interactive image evolution are presented in section 6.3.

5.5.4 The primitive sets

Here, the different primitive sets are described.

Arithmetic operations

The arithmetic primitive set consists of the library classes `java.lang-.Math`, `com.google.common.math.IntMath`, `com.google.common-.math.LongMath`, `com.google.common.math.DoubleMath` and the class `MathOperations`, which was written as part of the `jpea-jgap` module. The adjacent list contains the different types of functions in these classes.

- **Standard operations:** Like addition, subtraction, multiplication, division and modulo.
- **Trigonometric functions:** Like sine, cosine and tangent.
- **Number conversion functions:** Conversion functions between the types `double`, `float`, `integer`, `long` and `boolean`.
- **Conditional functions:** Simple conditional functions like if-statements, greater than and lower than.
- **Rounding functions:** Rounding functions supporting several rounding modes.
- **Other functions:** Other common functions like square root, power, logarithm and others.

Fractals functions

The fractal primitive set incorporates the type `org.apache.commons-.math3.complex.Complex` for complex numbers and contains several complex and double functions as listed below. All complex functions are gathered in the class

`ComplexFunctions` of the `jpea-jgap` module, mostly to wrap non-static functions in static functions and provide a limited set of double functions, additionally the double functions used in the arithmetic primitive set are available here as well:

- **Complex functions:** Several complex functions like addition, multiplication, division, negation, power, square root, sine, cosine, tangent and others are available.
- **Double functions:** All of the above double functions are available.

Chapter 6

Results and analysis

6.1 Introduction

In this chapter, the results of different experiments and tests will be presented. First, section 6.1.1 will give an overview of the default configuration parameters used during the evolutionary runs. After that, results of the two different image creation methods are introduced in section 6.2.

Then, the results of the interactive image evolution will be described in section 6.3 before section 6.4 will present in greater detail the results of automatic image evolution using different evaluation measures.

6.1.1 Run configuration

The following table shows the run configuration which is, if not stated otherwise, used throughout the runs executed to achieve the results presented in this chapter. Most of these values are the default values of JGAP, some were tuned for performance.

Parameter	Value	Explanation
Population size	50	Generally, 50 is a rather small population size, but it has been chosen in this case due to performance limitations and it offers an acceptable degree of genetic diversity.
Population initialization	Ramped half-and-half	A version of ramped half-and-half is used, where the max. depth is varied according to the current individual's index in the creation loop.
Generation count	50	Number of generations evolved during one run (only for automatic evaluation).
Crossover function probability	90%	The probability of a function being chosen as crossover point instead of a terminal.
Percentage of new individuals	30%	Percentage of completely new individuals per generation. Serves to keep the genetic diversity at an acceptable level.
Minimum initial tree depth	2	The minimum tree depth of newly generated individuals.
Maximum initial tree depth	7	The maximum tree depth of newly generated individuals.
Crossover probability	90%	The probability of crossover being chosen as genetic operator for an empty slot in the new generation. Otherwise, reproduction is chosen (i.e., an individual is copied to the new generation).
Mutation probability	10%	Probability of a node being mutated during crossover or reproduction.
Selection	Tournament selection	Tournament selection with three participants is used for selection (see section 2.4.3).

6.2 Comparing image creation methods

Two different image creation methods have been used in this thesis. For most of the tests and experiments, arithmetic expression-based function image creation has been used. Alternatively, basic fractal expression-based image creation (see section 5.5.1) has been implemented. In this section, they will be compared.

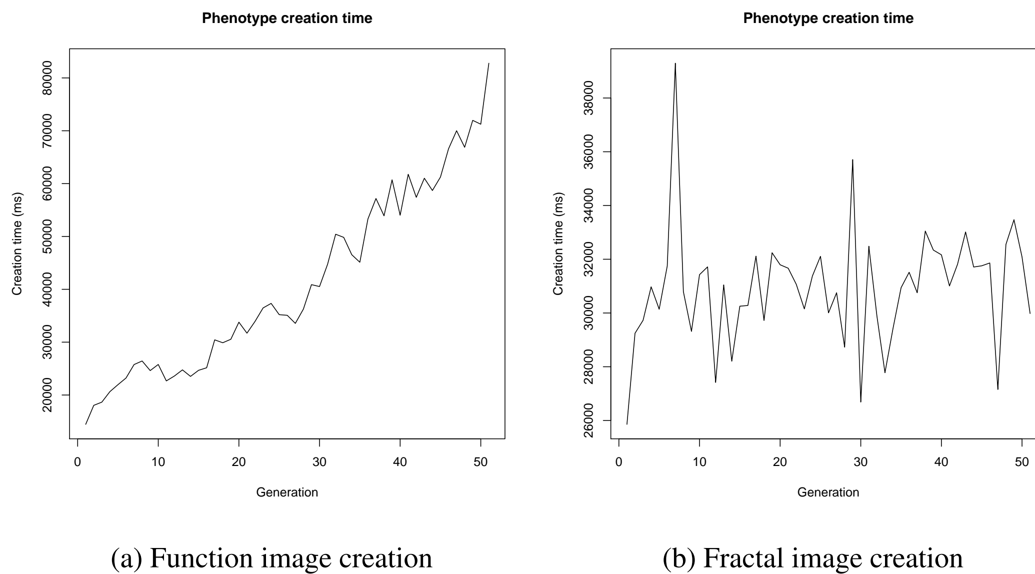
On the genotype level, both methods use an expression tree. In the case of the function image creation, the return value of the expression tree is a color vector and variables are incorporated into the tree which can be changed from the outside to generate different results for multiple executions of the expression tree. The fractal image creation works differently: The expression tree directly returns an image upon evaluation, encapsulating the whole fractal image creation into the primitive set. For this reason, the expression tree has to be evaluated only once.

Figure 6.1 shows a comparison of image creation times. While the function image creation depends on the depth of the expression tree, the fractal image creation is independent of the genotype and always takes roughly the same time (this would change if, for example, the number of iterations when iterating a point through the fractal formula, would be subject to evolution).

Figure 6.2 shows example images of function and fractal image creation. Fractal image creation, with a fixed number of iterations and a fixed formula, shows less phenotype diversity, while function image creation (more examples can be found in the later section of this chapter) can compete in image complexity but is overall slower than fractal image creation.

6.2.1 Automatically defined functions

JGAP offers support for automatically defined functions (which are shortly introduced in section 2.4.1). All ADFs have to be explicitly declared and receive their own function set–they basically form a second, parallelly evolved expression tree with its own root node. However, ADFs do not seem to offer better fitness values or any other advantages in the case of arithmetic function image creation, as can be seen in figure 6.3, which is why they will not be used for any of the following



(a) Function image creation

(b) Fractal image creation

Figure 6.1: Phenotype creation time for function image creation and fractal image creation.

tests. Also, the phenotype creation time is much higher (when put in relation with genotype complexity) when using ADFs.

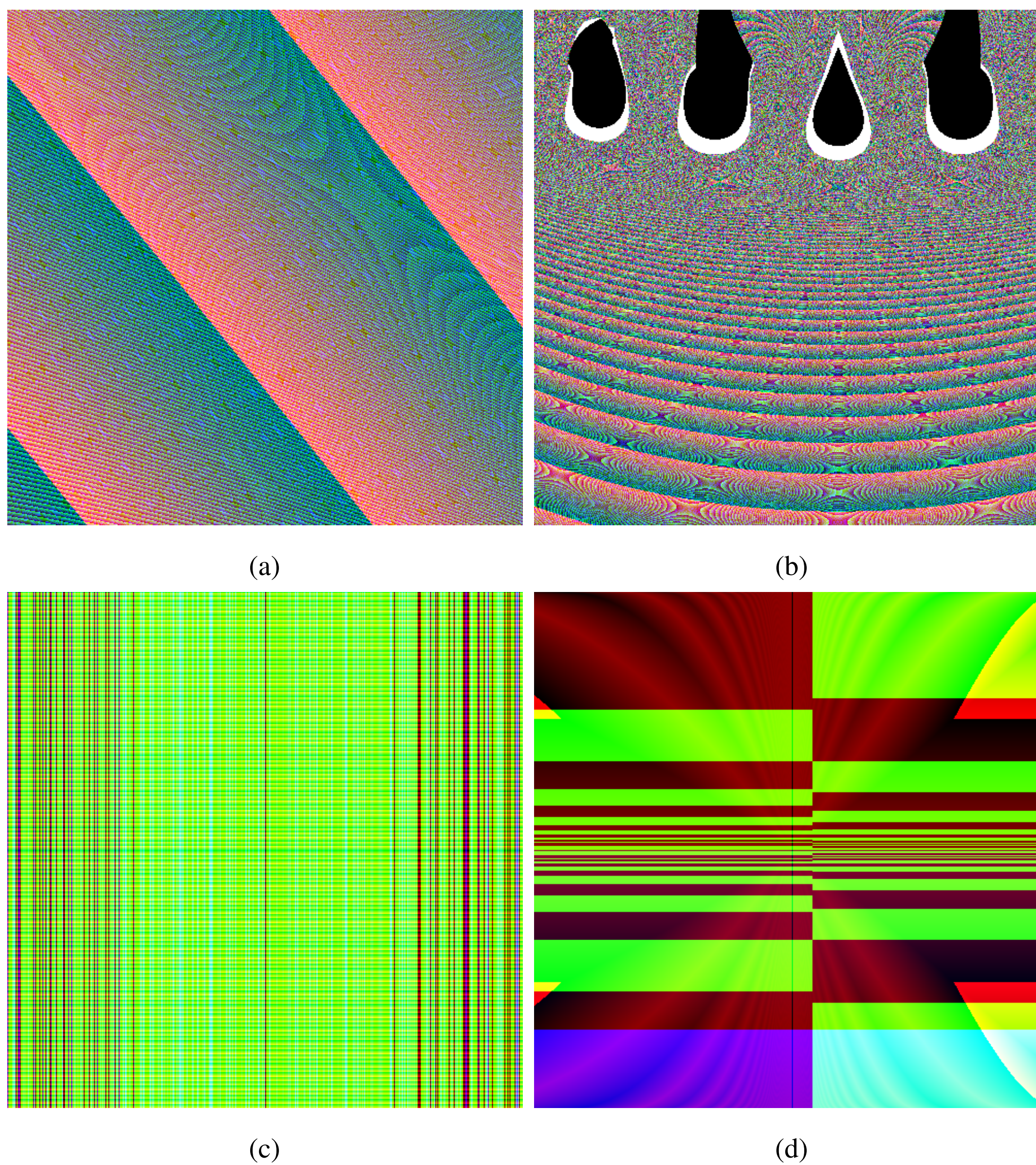


Figure 6.2: Image examples of fractal ((a) and (b)) and function ((c) and (d)) image creation.

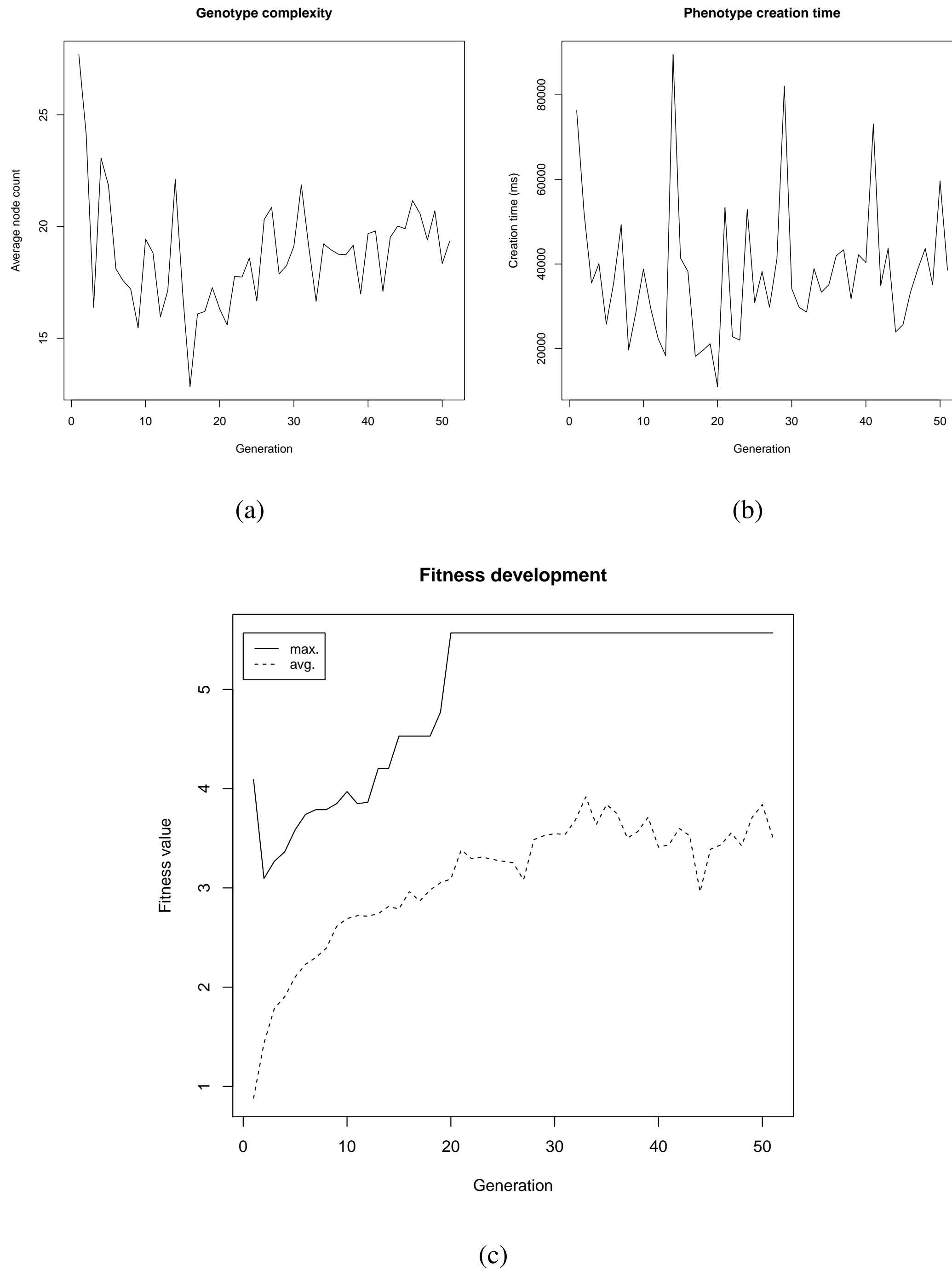
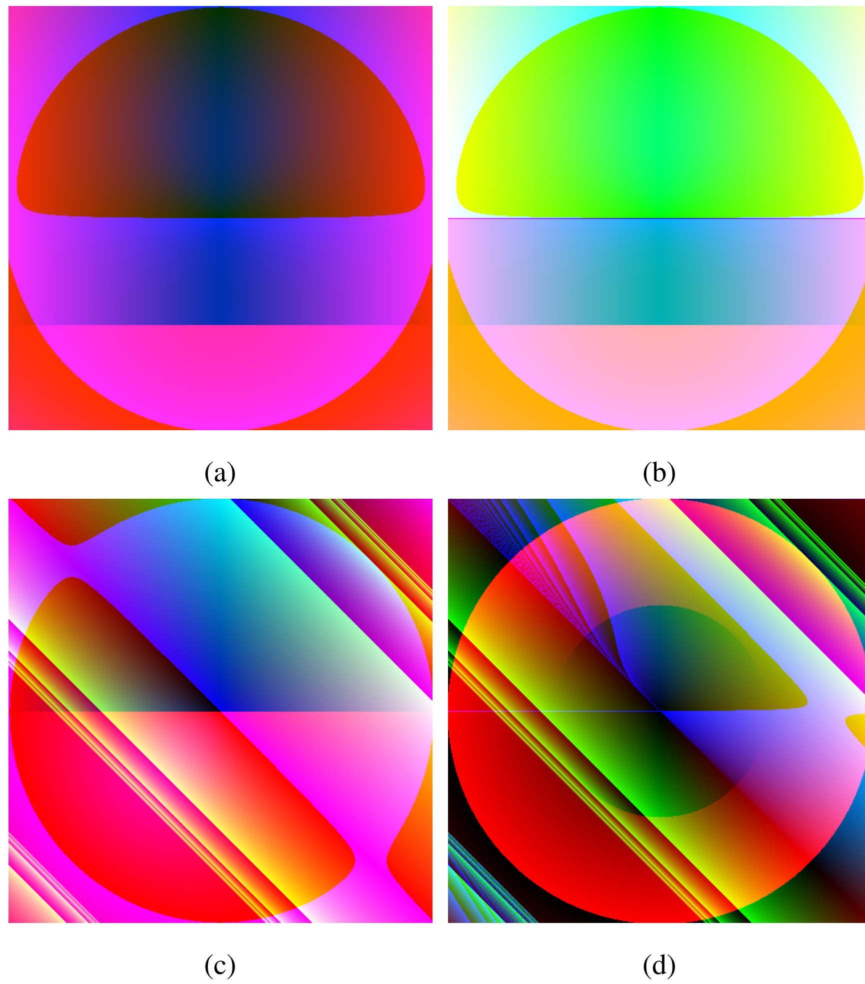


Figure 6.3: Genotype complexity (a) compared to phenotype creation time (b) and fitness development (c) when using ADFs and GCF evaluation.

6.3 Interactive image evolution

After the changes that were made to the interactive image evolution as well as to the genotype-phenotype mapping (see section 5.5.3), this section will present some images that were generated at different stages of the evolutionary process. Figure 6.4 shows four example images that were encountered at different stages of the evolution.



(a) (b)

(c) (d)

Figure 6.4: Image examples of interactive image evolution at different stages of the evolution: Initial population (a), after five generations (b), after ten generations (c) and after 15 generations (d).

6.4 Automatic image evolution

The following sections will present evolution results for different automatic evaluation algorithms.

6.4.1 Global contrast factor

The GCF, as described in sections 3.3.2 and 5.4.2, is an algorithm which evaluates the amount of detail (as perceived by a human viewer) in an image.

Figure 6.5 shows two example results of evolutionary runs using GCF as an evaluation measure. The results are similar in that they exhibit many sharp edges and no smooth color gradients–as expected of an algorithm that prefers local contrasts, i.e., many details.

On to some statistics: Figure 6.6 shows the development of genotype complexity (exposing clear patterns of bloat in the later stages, see section 2.4.3), while figure 6.7 shows the development of fitness values and evaluation times over the course of the evolutionary process. Unexpectedly, the evaluation time decreases throughout the evolution for no apparent reason–possible explanations include recompilation and optimization of code paths after their first execution, threading optimization and color distributions which allow for faster calculations in the algorithm. Fitness quickly develops towards 7, after which the fitness improvement continues at a slower pace, eventually reaching 8.

Overall, the GCF works well as a primary aesthetic measure as it favors images with many details without producing images which are as distorted or "noisy" as the image complexity measure (see section 6.4.2).

6.4.2 Image complexity

The image complexity measure, as described in sections 3.3.2 and 5.4.2, is an algorithm which evaluates image complexity by looking at the performance of JPEG compression of an evolved image–compression size and the error in the compressed image (RMSE) are taken into account. It is part of the Machado & Cardoso aesthetic

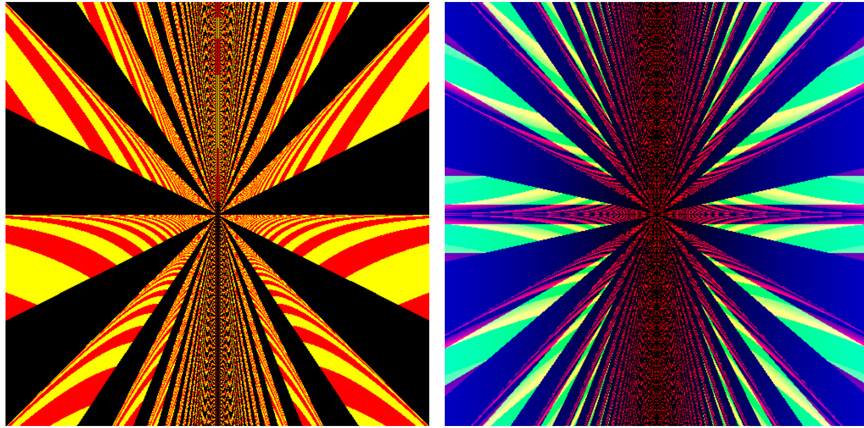


Figure 6.5: Example results of GCF-guided image evolutions.

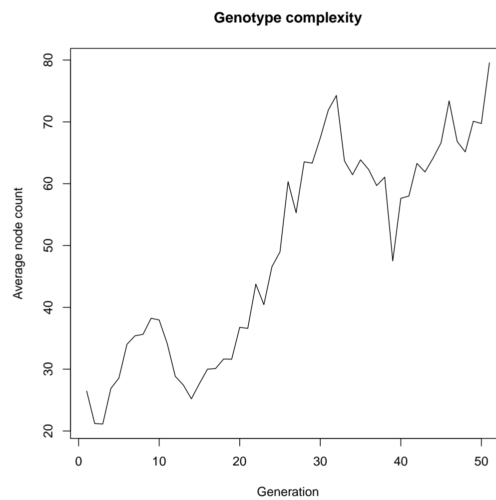
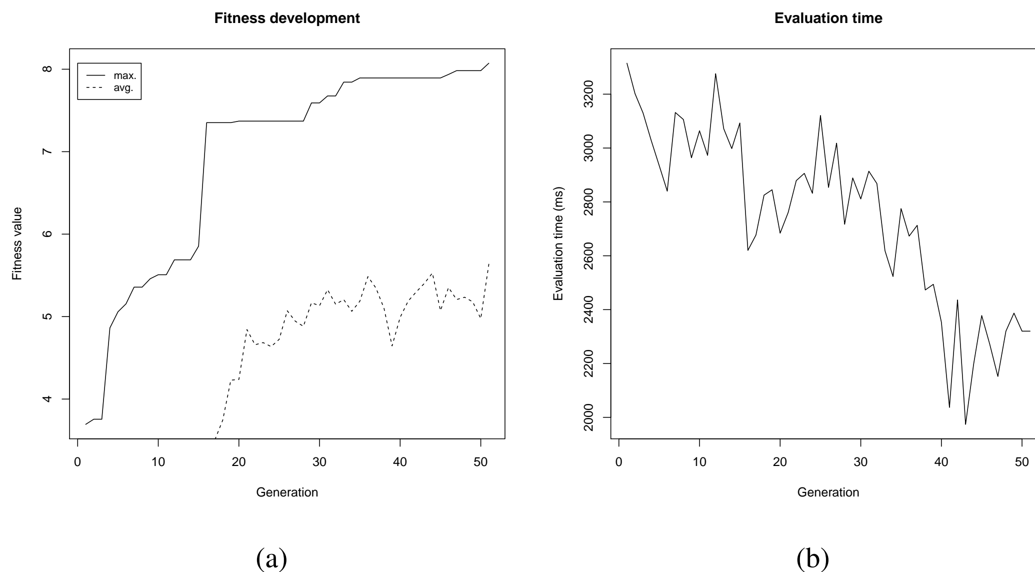


Figure 6.6: Development of genotype complexity for GCF-guided image evolution.



(a) (b)

Figure 6.7: Development of fitness values (a) and image evaluation times (b) for the GCF-guided image evolution.

measure and will, at this point, only take the image complexity into account, ignoring the processing complexity.

Figure 6.8 shows two example results of the IC-guided evolution. It can be seen that IC prefers images with a lot of color distortion or “noise”, i.e., big (and hard to predict) changes in color between neighboring pixels, as these are hard to compress and produce images with higher complexity.

Just like in the case of GCF, bloat can be observed, even though the increase in genotype complexity is quicker, hinting at the expected relation between genotype and phenotype complexity (see figure 6.9). The fitness values are quickly growing until generation 30, where stagnations sets in and even growing genotype complexity can no longer produce images with much higher complexity. The evaluation times very roughly correspond to the fitness values, meaning more complex images take longer to evaluate.

Image complexity works well as a primary evolutionary objective and produces aesthetically pleasing images, although, in later stages of the evolution, the subjective image quality degrades because of the color distortion and noise.

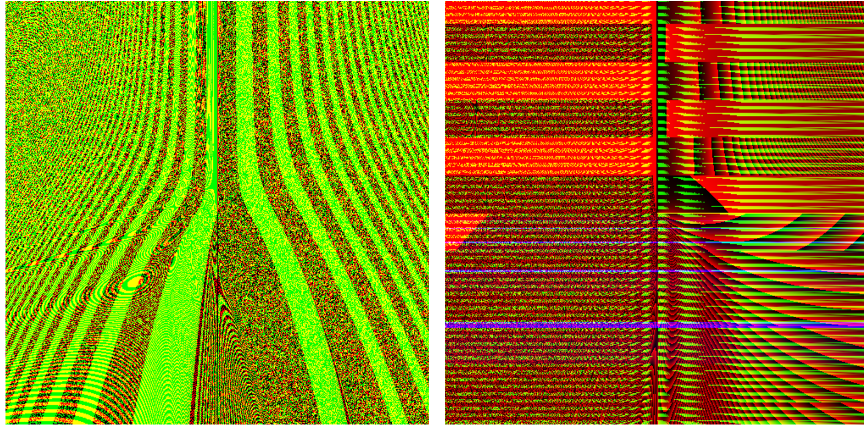


Figure 6.8: Example results of IC-guided image evolutions.

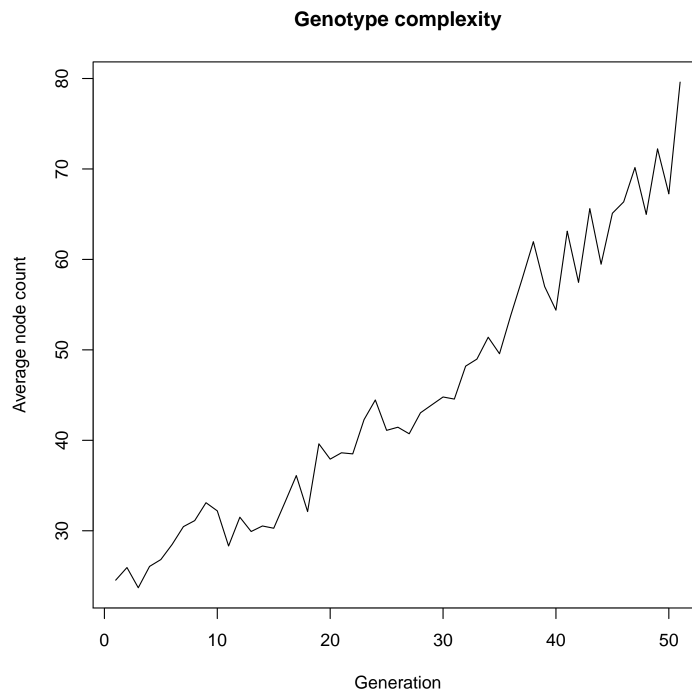
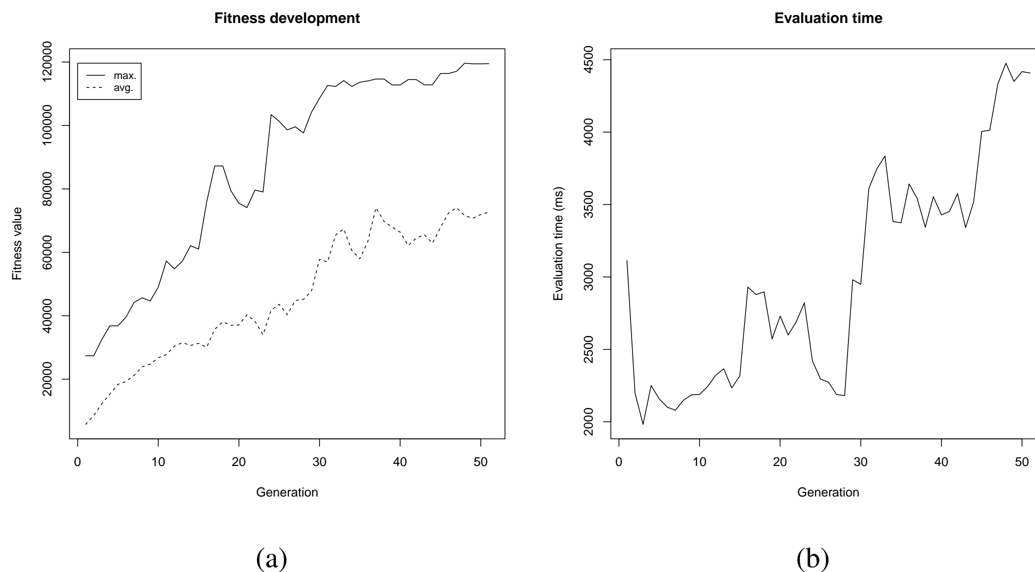


Figure 6.9: Development of genotype complexity in the IC-guided image evolution.



(a) (b)

Figure 6.10: Development of fitness values (a) and image evaluation times (b) in the IC-guided image evolution.

6.4.3 Fractal dimension

The fractal dimension evaluation is described in sections 3.3.2 and 5.4.2; it's an algorithm that uses "box-counting" to explore self-similarities of an image.

Figure 6.11 shows two example images of the FD-guided evolution. The target FD was set to 1.35 (as explained in 3.3.2), and figure 6.13 shows the development of the average fitness value as it first moves towards that value and then, towards the end, slightly away from it as the best fitness value stagnates close to 1.35. There are several images sharing similar fitness values that do not expose obvious patterns or similarities as it was the case for GCF or IC. Simple color gradients and patterns are preferred, but bloat still occurs (see figure 6.12). The fact that more simple patterns are preferred makes FD more interesting as a secondary aesthetic measure with IC or GCF as the primary measure.

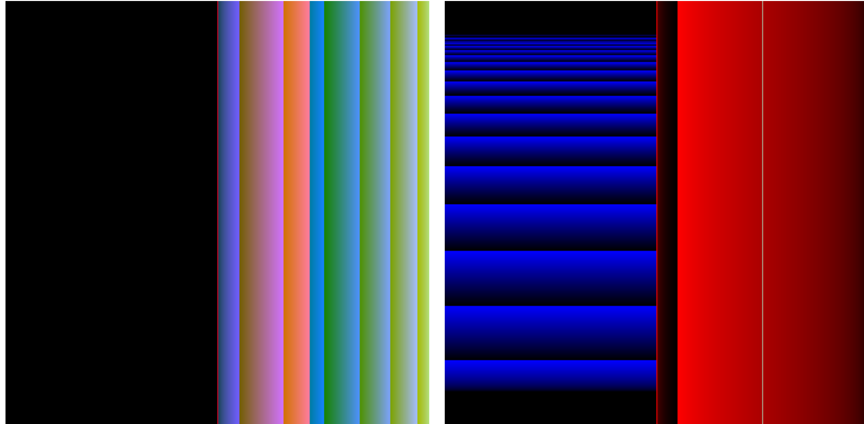


Figure 6.11: Example results of FD-guided image evolutions.

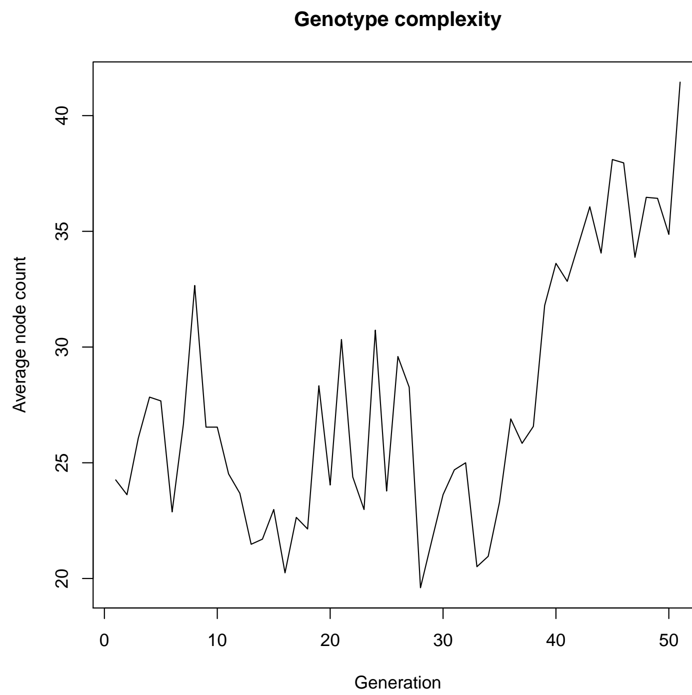
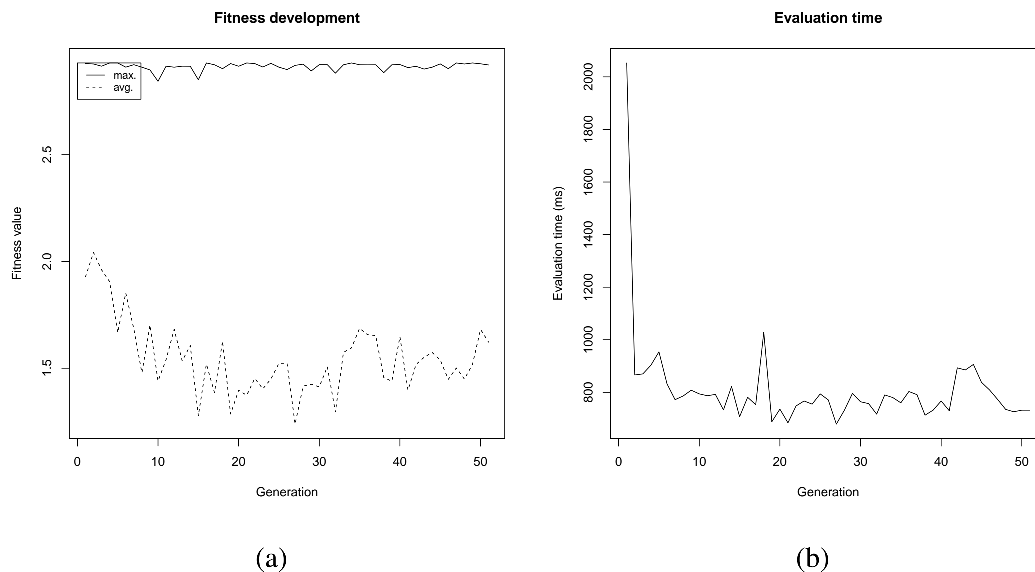


Figure 6.12: Development of genotype complexity.



(a) (b)

Figure 6.13: Development of average fitness values (a) and image evaluation times (b).

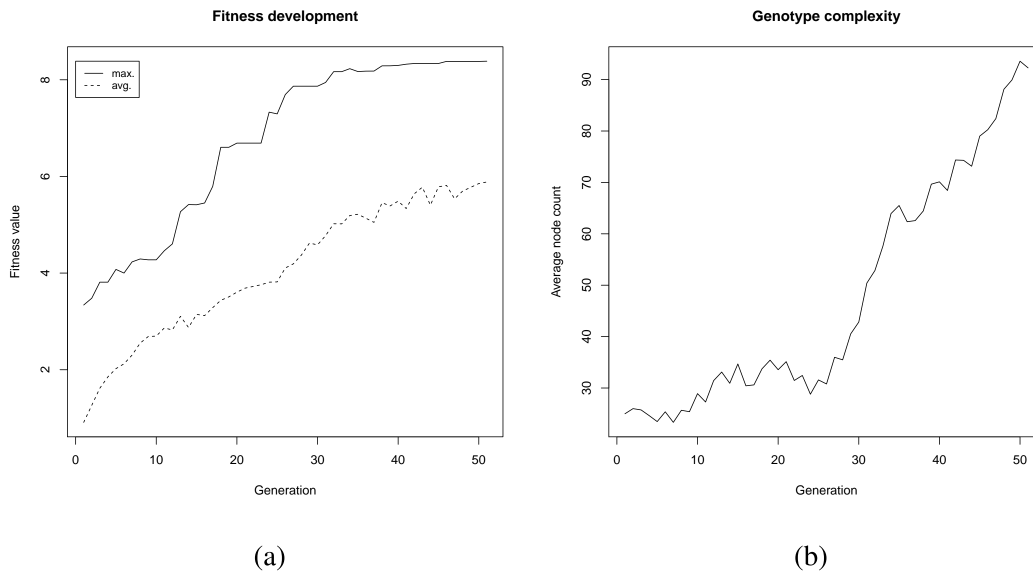
6.4.4 Behavior for increased population and generation parameters

In this section, two configuration parameters are changed to analyze the behavior of the image evolution (especially bloat and fitness values) with different configuration values.

Population size

The population size has usually been set to 50 individuals due to performance restrictions–usually, higher values are used. In this run, the population size is increased to 100. Figure 6.14 shows the development of fitness values and bloat using GCF as evaluation.

No real surprises here, the average fitness values and node counts develop a little smoother due to the greater population size and the maximum fitness value moves up in more, smaller steps and the 8 is hit earlier, but does not really improve much beyond that (see figure 6.7 for comparison).



(a) (b)

Figure 6.14: Development of average fitness values (a) and genotype complexity (b) for an increased population size.

Generation count

The generation count was set to 50 for previous automatic evaluation tests. Again, not the highest number, but performance restrictions did not realistically allow for higher numbers for all tests. In this run, the number is increased to 100 to observe the development of fitness values and bloat beyond the point of 50 generations. Again, GCF evaluation is going to be used, so see figure 6.7 for comparison. Figure 6.15 shows the results of the new tests over 100 generations.

The more interesting statistic here is the genotype complexity development: It continues (and even accelerates) beyond generation 50 and only slows down after generation 80 but never comes to a halt. The fitness value, on the other hand, reaches the peak value in generation 60 and improves by very little after that, while the average fitness value stays below 6 after about 50 generations.

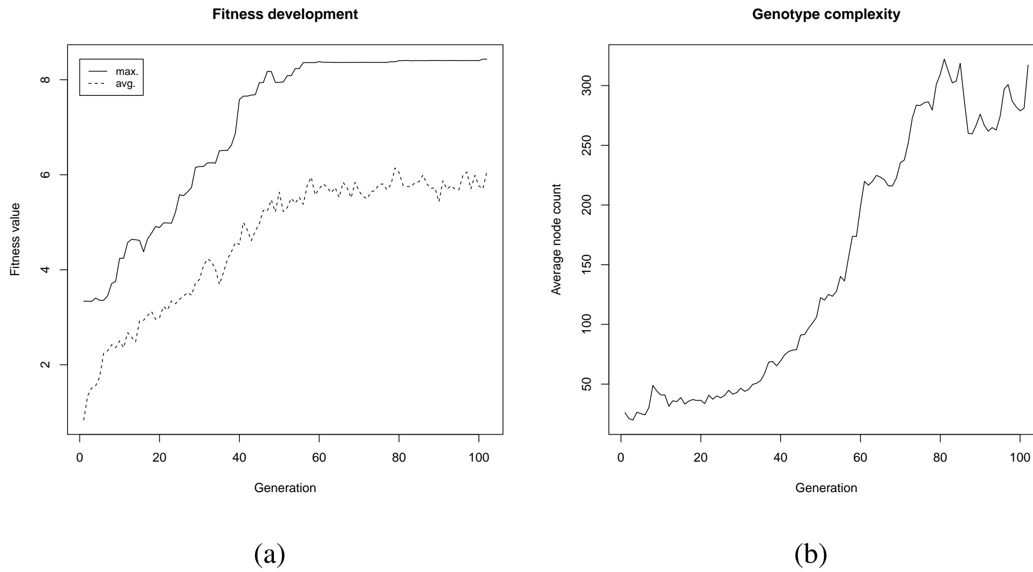


Figure 6.15: Development of average fitness values (a) and genotype complexity (b) for an increased generation count.

6.5 Multi-objective image evolution

This works well in scenarios when there is one primary evaluator and one or more secondary evaluators (e.g., for bloat control).

6.5.1 Bloat control

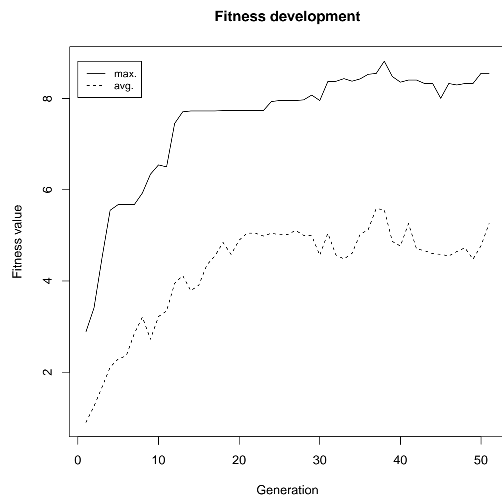
For this experiment, two of the above tests (GCF and IC) are performed again, this time with a second fitness measure, which is integrated using the formula

$$M = \frac{M_P}{M_{BC}} \tag{6.1}$$

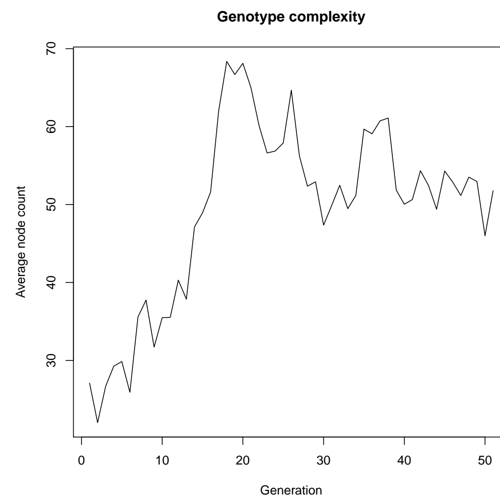
where M_P is the primary aesthetic measure and M_{BC} is the bloat control measure defined as

$$M_{BC} = \sqrt{1 + \frac{N}{1000}} \tag{6.2}$$

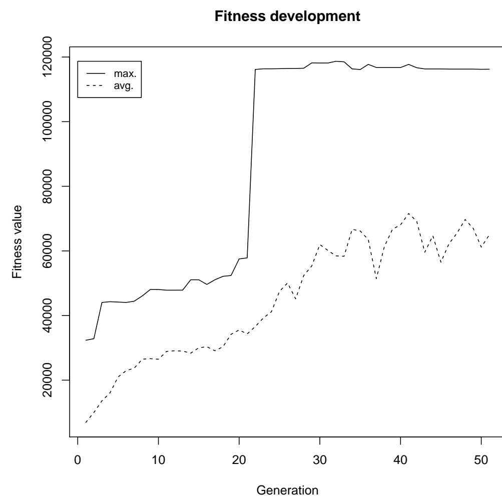
with N being the number of nodes in the genotype. Figure 6.16 shows the development of the actual fitness measure M and the genotype complexity.



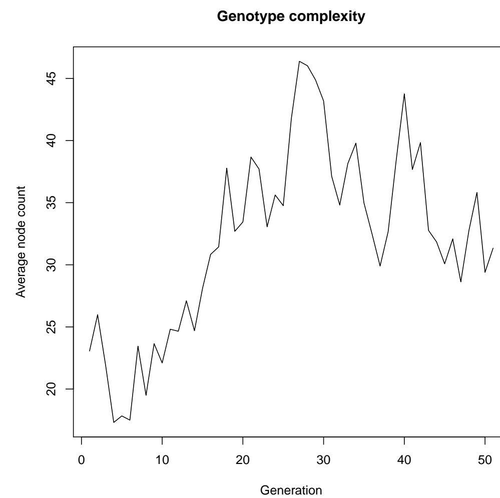
(a) GCF fitness



(b) GCF genotype complexity



(c) IC fitness



(d) IC genotype complexity

Figure 6.16: Development of GCF/IC fitness and genotype complexity.

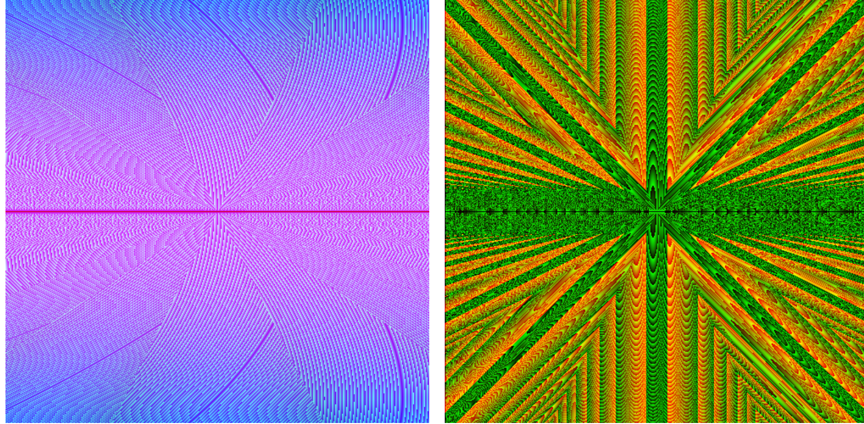


Figure 6.17: Example results of multi-objective image evolutions.

It can be concluded that bloat control (with the parameters used here) does not negatively affect either GCF or IC fitness values, when seen over the whole evolution. In times of overall fitness stagnation, individuals are optimized towards smaller genotypes, thereby slightly decreasing the fitness value of the primary aesthetic measure.

6.5.2 Multiple aesthetic measures

In this section, multi-objective image evolution will be performed using the two aestehtic measures IC and GCF, as well as bloat control. the overall formula used is

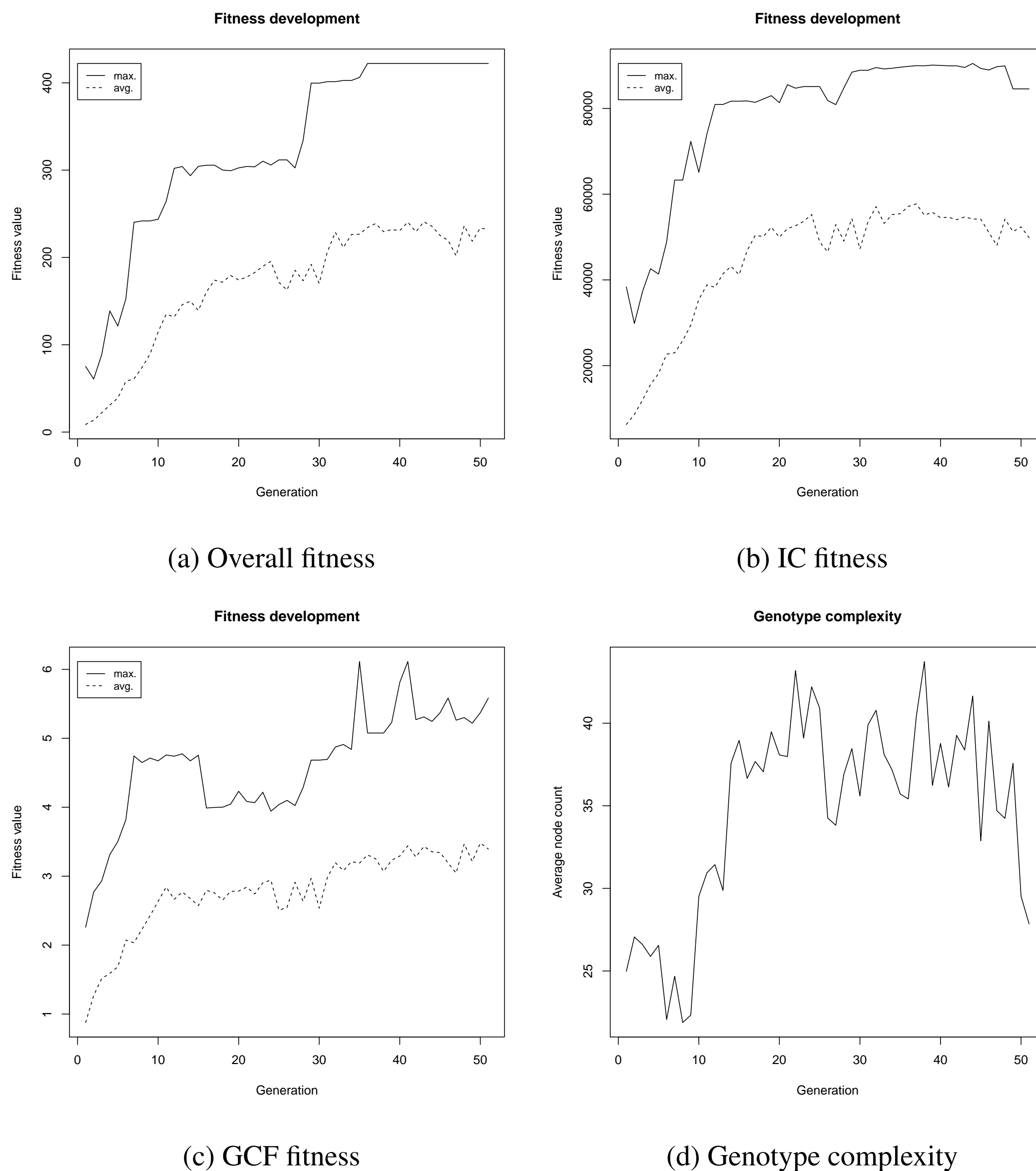
$$M = \frac{M_{GCF} \cdot M_{IC}}{1000 \cdot M_{BC}} \tag{6.3}$$

with M_{BC} being the bloat control measure as introduced in equation 6.2 in section 6.5.1. Figure 6.17 shows two examples of the multi-objective image evolution. IC is overall the stronger measure because it has greater absolute values, but the influence of the GCF evaluation can still be seen in the generated images.

Figure 6.18 shows the overall and individual fitness developments throughout the run.

6.5.3 Comparing evaluation speeds

In this section, the evaluation speeds of GCF, IC and FD evaluation will be compared.



(a) Overall fitness

(b) IC fitness

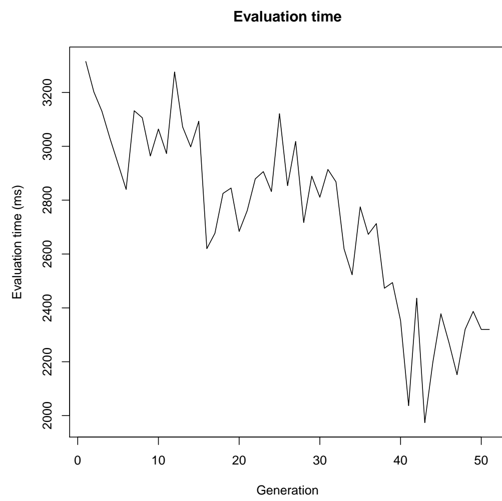
(c) GCF fitness

(d) Genotype complexity

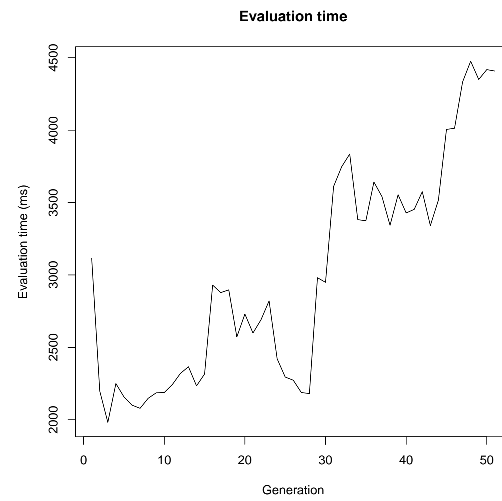
Figure 6.18: Development of overall and individual fitness for multi-objective image evolution.

Figure 6.19 shows the evaluation times for the three automatic evaluation algorithms (as also shown before in each respective section). The fractal dimension evaluation (6.19c) is generally the fastest, with evaluation times between 700 and 900*ms* per generation; it is also largely independent of the fitness value. Next is global contrast factor (6.19a), taking between 2000 and 3200*ms* to evaluate one generation of images. Here, a clear overall decrease of evaluation time can be noticed over the course of the evolution, indicating a negative correlation between fitness value and evaluation time (reasons for this are not entirely clear, some possibilities are listed in section 6.4.1). Lastly, image complexity (6.19b) is the slowest evaluation algorithm, taking about 2000*ms* in the early stages of the evolution and up to 4500*ms* in the later stages, clearly exposing a strong relation between fitness value and evaluation time. This can easily be explained by looking at the algorithm: The JPEG compression takes longer for more complex images, resulting in overall higher evaluation times.

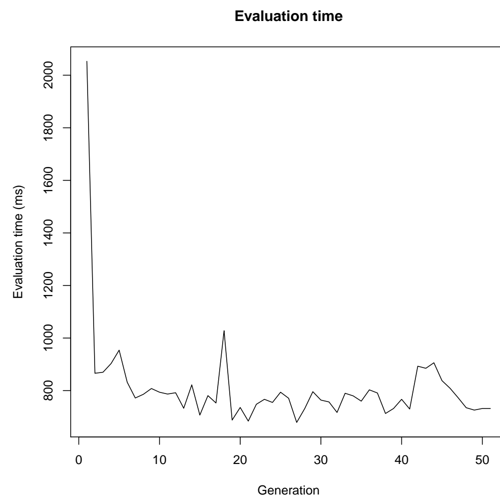
Comparing image evaluation times with image creation times (see section 6.2), image evaluation is generally much faster (about an order of magnitude) and thus less critical when looking at the overall performance of the evolutionary system.



(a) Global contrast factor



(b) Image complexity



(c) Fractal dimension

Figure 6.19: Comparison of evaluation times.

Chapter 7

Conclusion and future prospects

This thesis has given an overview of the field of evolutionary art and the techniques involved in generating and evolving the genotype as well as the images and their evaluation. Two applications (ReGeP and Jpea) that were developed in (Rueckert 2013) were extended and extensive tests have been performed using automatic image evolution with different combinations of evaluation algorithms. Arithmetic function image creation was the primary image creation method, though fractal image creation has been implemented and tested as well. Additionally, interactive image evolution has been improved to allow for better and quicker aesthetically pleasing image generation.

The most successful automatic image evaluation algorithms introduced here were global contrast factor evaluation and image complexity evaluation. Arithmetic function image creation provides a wide variety of different kinds of images, even though it is limited in that it only incorporates local information when generating images. Fractal dimension has proved to be less successful when evaluating images.

Fractal image generation has potential to generate beautiful image, but has many drawbacks, including slow generation speed, difficult coloring algorithms that need to be tuned and a limited generalization. In short, fractal image generation has been more successfully used in GA systems than in GP systems, because mostly certain formulas are used, in which different parameters can be changed. When trying to generate fractals or iterated function systems from scratch, the number of interesting solutions is completely lost in the vast amount of uninteresting ones.

Concluding, different existing image evaluation and creation algorithms (using different primitive sets) have been implemented and their results have been presented, it can be said that evolutionary art is still a field with much potential, especially when looking at future processor speeds and increasing number of processor cores. Automatic evaluations are useful but still do not really capture human aesthetic preferences–a fact that might not change in the near future, seeing how complex of a topic it is.

Appendix A

The software environment

This chapter will shortly introduce the software environment used in this thesis. First, different frameworks and libraries will be listed in section A.1. Then, an overview of ReGeP (section A.2) and Jpea (section A.3) will be given.

A.1 Frameworks and libraries

A.1.1 JGAP

> “JGAP (pronounced "jay-gap") is a Genetic Algorithms and Genetic Programming component provided as a Java framework. It provides basic genetic mechanisms that can be easily used to apply evolutionary principles to problem solutions.“[1]

JGAP is used in Jpea (see section A.3) to evolve the genotypes for the images using the primitive set which is created using ReGeP (see section A.2). This section will shortly introduce the most important components and configuration directives of JGAP and is, in parts, a translation of the JGAP introduction in (Rueckert 2013).

For a more exhaustive introduction to JGAP, see (Rueckert 2013).

[1] `http://jgap.sourceforge.net/` (visited on 08/25/2013)

Overview

GP programs (interface `IGPPogram`) consist of a number of *branches* (interface `IGPChromosome`), which, taken individually, are *syntax tress* made up of *nodes* (class `CommandGene`) and *leaves* (also of class `CommandGene`). The nodes are *functions*, having a return value and an arbitrary number of parameters (called *arity*). Together with the *terminals* (functions without parameters), they form the *primitive set*. For simple GP problems (class `GPProblem`) there is often only one syntax tree, the *root node* of which has a certain return value.

Configuration

JGAP offers configuration directives through the class `GPConfiguration`, the most important of which will be described here.

Fitness function

Using the method `setFitnessFunction(GPFitnessFunction)`, an object can be set which is responsible for the fitness values.

Fitness evaluation

Using the method `setGPFitnessEvaluator(IGPFitness-Evaluator)`, an object can be set which is responsible for interpreting the fitness values (e.g., whether lower fitness values are better).

Selection

Using the method `setSelectionmethod(INaturalGPSelector)`, an object can be set which is responsible for the selection process (the default is a tournament selection).

Crossover

Using the methods `setCrossoverMethod(CrossMethod)`, `set-CrossoverProb(float)` and `setMaxCrossoverDepth(int)`, the method to be used for crossover, the crossover probability and the maximum resulting individual depth can be set. Additionally, using the method

`setFunctionProb(float)`, the probability of functions being used as crossover points (as opposed to terminals) can be configured.

Mutation

Using the method `setMutationProb(float)`, the probability of a node being mutated during program creation can be set.

Population composition

Using the method `setNewChromsPercept(double)`, the percentage of newly generated individuals in each generation can be set.

Individual complexity

The method `setMaxInitDepth(int)` and `setMinInitDepth(int)` allow the limitation of the initial tree depth for new individuals.

A.1.2 Other libraries

This section will list the most important third-party libraries used in ReGeP and Jpea and explain their function shortly.

Commons Math

> "Commons Math is a library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the Java programming language or Commons Lang."[2]

A math library which is used mainly for its `Complex` class and the functions implemented therein.

The data-exporter library

> "data-exporter is a Java library to export the tabular data (like List of rows) into many output formats."[3]

[2] `http://commons.apache.org/proper/commons-math/` (visited on 08/25/2013)
[3] `http://code.google.com/p/data-exporter/` (visited on 08/25/2013)

The data-exporter library is used to export statistical data into CSV files for further processing.

Fractal Image Compression (F.I.C.)

> "An open source library written in Java, implementing the concepts of fractal image compression, along with a simple implementation–a proof of concept application."[4]

The fractal image compression library was used as an attempt at estimating processing complexity of an image.

Guava

> "The Guava project contains several of Google's core libraries that we rely on in our Java-based projects: collections, caching, primitives support, concurrency libraries, common annotations, string processing, I/O, and so forth."[5]

This general purpose library is used in several places in ReGeP and Jpea to simplify collection handling, primitive wrapping and other processing.

JH Labs Java Image Filters

The Java image filters from JH Labs[6] are used for simple image transformation (grayscale) in Jpea.

Marvin Image Processing Framework

> "Marvin is an extensible, cross-platform and open source image processing framework developed in Java."[7]

---

[4] `http://c00kiemon5ter.github.io/Fractal-Image-Compression/` (visited on 08/25/2013)

[5] `http://code.google.com/p/guava-libraries/` (visited on 08/25/2013)

[6] `http://www.jhlabs.com/ip/filters/` (visited on 08/25/2013)

[7] `http://marvinproject.sourceforge.net/en/index.html` (visited on 08/25/2013)

The Marvin framework is used primarily to handle image presentation in the GUI.

The imgscalr library

> "imgscalr is an very simple and efficient (hardware accelerated) 'best-practices' image-scaling library implemented in pure Java 2D;[...]"[8]

This library is used to scale images in different places of Jpea (population thumbnails, phenotype database and others).

A.2 Reflection-based Genetic Programming (ReGeP)

Reflection-based Genetic Programming[9] (ReGeP) was developed as part of (Rueckert 2013) to automate and decouple the construction of the primitive set from the framework in which it is used. It consists of about 1000 lines of Java code in 34 classes. The basic working steps are as follows:

1. Determine the classes from which to extract GP functions
2. Extract elements from the classes which can be used as GP functions or terminals (e.g., methods, constructors, attributes) through reflection
3. Provide access to these elements through a unified interface (`Function`)
4. Make `Function` instances usable for the GP framework by wrapping them in an adapter class
5. Make the validated GP primitive set available though a `Function-Database`.

Steps 1 through 3 and 5 are realized in the core module `regep-core`, while step 4 is done in the framework-specific module `rege-jgap`.

The following is a shortened and translated version of (Rueckert 2013, p. 26-36).

[8] `http://www.thebuzzmedia.com/software/imgscalr-java-image-scaling-library/` (visited on 08/25/2013)

[9] `https://gitorious.org/regep` (visited on 08/25/2013)

A.2.1 Processing

The processing component of the `regep-core` module mainly consists of the two interfaces `Processor` and `ProcessorChain`. A processor has only one method: `Collection<Object> process(Object input)`. As the method signature suggests, a processor expects one input element and returns a (possibly empty) collection of output elements. Usually, to be able to process an element, a processor expects its input elements to be of a certain class or have some other property or state. For this reason, the `Processor` interface extends the `Applicable` interface that provides the method `boolean isApplicable(Object input)`, which returns `true` if the processor can process the given input object.

The reason why the `Processor` interface is kept so general will become clearer when looking at the `ProcessorChain` and how it works. The `ProcessorChain` interface extends the `Processor` interface and provides a single method `addProcessor(Processor p)` which allows processors to be added to the chain. A processor chain is expected to work as shown in figure A.1: Each input object is processed in each processor. The output objects of each processor are again treated as input objects. In this way (considering that only those processors are active for which the current input object is applicable), a hierarchical processing structure is implemented without the overhead of programmatically describing the hierarchy. The drawback of this approach is the vulnerability to endless recursions if there are no protection mechanisms in place.

There are two implementations of the `ProcessorChain` interface, one is the `RecursiveFilteringProcessorChain`, which additionally implements the `FilterChain` interface to allow fine-grained control over which objects are processed. It works recursive, i.e., if one thinks of the object structure as a tree (an input object being the root node and the corresponding output objects its children), objects are processed in a depth-first manner. The other one, `ConcurrentRecursiveFilterProcessorChain` is an extension of the aforementioned class to support concurrent processing (see section 4.4).

Figures A.2 and A.3 show the concrete processing hierarchy implemented

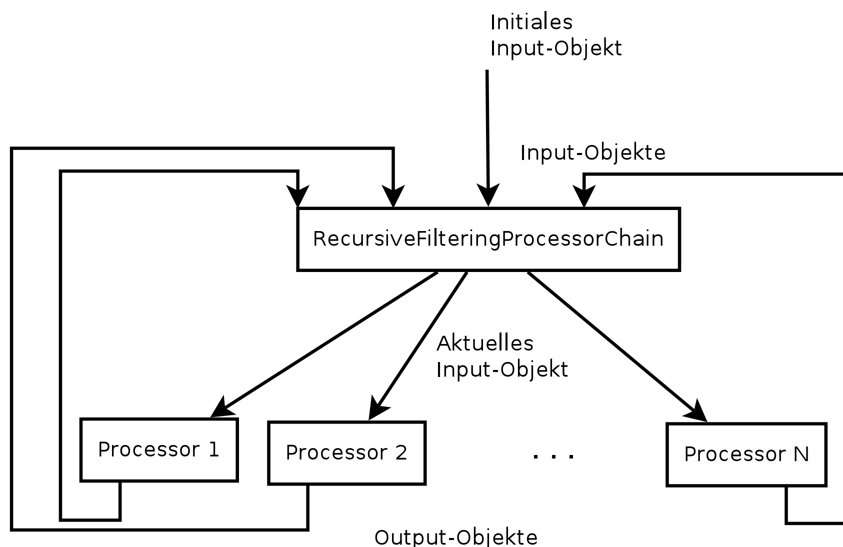


Figure A.1: Processing principle. Source: (Rueckert 2013)

by `regep-core` and `regep-jgap`, respectively, in which the `FunctionDatabase` is created an populated. There are processor classes to extract classes from packages, extract methods and constructors from classes, wrap them in `Function` classes, wrap those in `JgapFunctionAdapter` classes and register those in the `FunctionDatabase`. The resulting nested object structure is shown in figure A.4.

A.2.2 FunctionDatabase

After the processing stage, the function database is populated with possible GP functions. To retrieve the validated primitive set, the return type(s) of the generated programs have to be specified using `addRootType(Class<?> type)`. Using this type information, the `FunctionDatabase` recursively validates all functions of the primitive set, using two criteria:

- Can the function be used as an argument to another function?
- Are there any functions which can serve as arguments to this function?

The second point is more problematic than the first, because it has to be checked recursively. In the beginning, there are no valid functions and the only other “function”

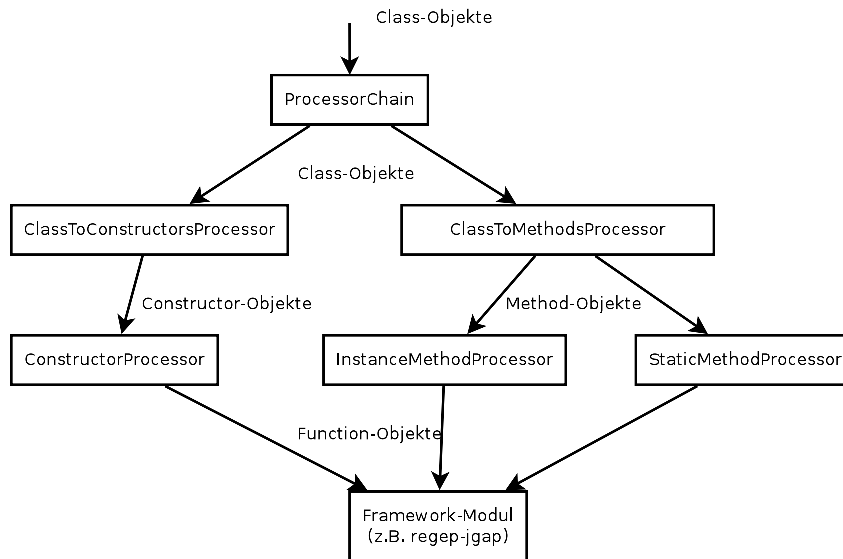


Figure A.2: Processing in `regep-core`. Source: (Rueckert 2013)

to which any validated function can serve as an argument are the root types. So, without further checking, only terminals would be valid, because the second point would never be true (as there are no other valid functions). To validate such functions, the program building process is simulated by recursively searching functions which can be used as arguments of the current function until a terminal is reached. When all leaves are terminals, all functions in the simulated tree are valid.

Most GP frameworks offer similar validation as well.

The validation is automatically executed upon calling `Set<T> getFunctionSet()` which returns the primitive set to be passed on to the GP framework.

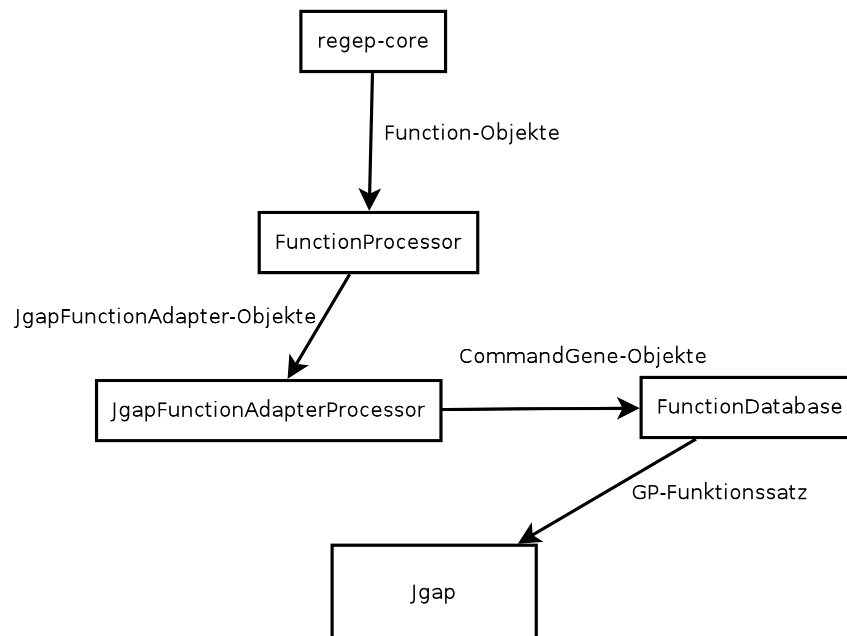


Figure A.3: Processing in `regep-jgap`. Source: (Rueckert 2013)

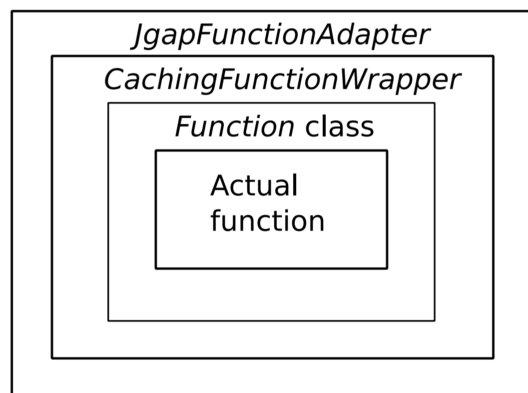


Figure A.4: Object nesting of the GP functions.

A.3 Java package for evolutionary art (Jpea)

The Java package for evolutionary art[10] (Jpea) was developed in (Rueckert 2013). It is a toolbox for evolutionary art applications by providing a framework to easily build and extend such applications. It consists of about 5300 lines of Java code in 112 classes.

Jpea is horizontally divided into image creation, image evaluation and an application package, and vertically divided into a core module (`jpea-core`) and framework-specific modules (`jpea-jgap`).

The following is loosely based on (Rueckert 2013, p. 37-46).

A.3.1 Image creation

Image creation is based on the `PhenotypeCreator` interface which contains two methods for creating phenotypes for an individual or for a population (the latter case is necessary for concurrent implementations (e.g., `ConcurrentPhenotypeCreator`). Figure A.5 shows the classes and interfaces of the phenotype creation package.

The `AbstractPhenotypeCreator` and `ConcurrentPhenotypeCreator` are abstract classes which provide different standard implementations of `createForPopulation(Population p)`, while the abstract `FunctionImageCreator` provides the basis for image creation using an `ImageFunction`. An image function can be evaluated at given `x` and `y` positions (e.g., pixel coordinates) and return a value of some kind.

One concrete `ImageFunctionCreator` is the `VectorFunctionImageCreator` which expects the return value of the associated `ImageFunction` to be a `Vector` containing the RGB color components.

In the `jpea-jgap` module, there are two more classes. First, the `JgapImageFunction`, which executes JGAP programs (`IGPProgram`), and the `JgapImageCreator`, which wraps a `VectorFunctionImageCreator`, takes the

[10] `https://gitorious.org/jpea/` (visited on 08/25/2013)

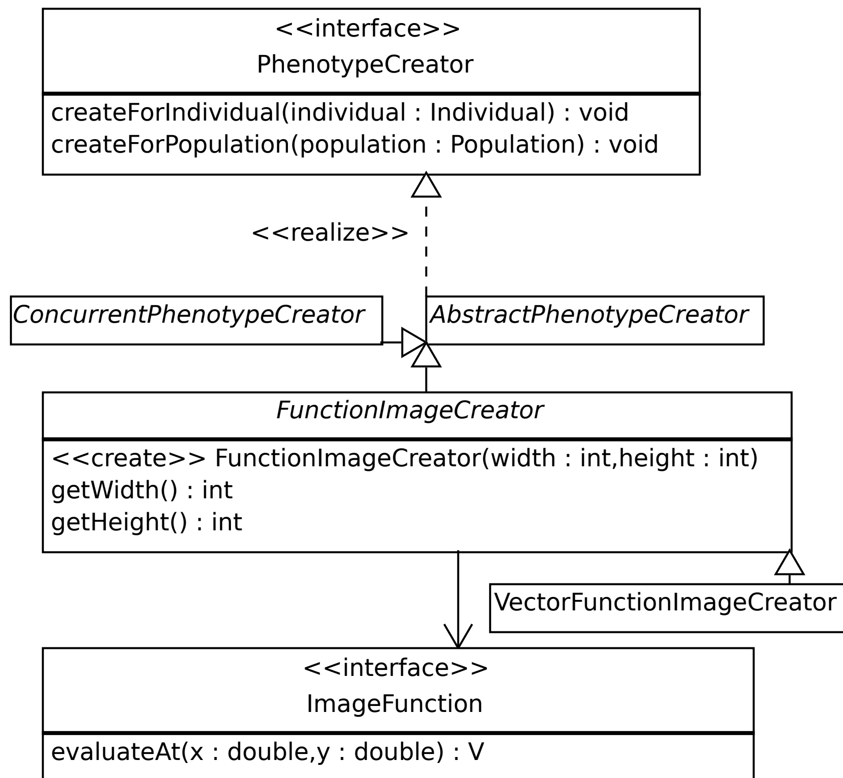


Figure A.5: Important classes and interfaces of the phenotype creation package.

JGAP individuals (the genotypes of which are of type `IGPProgram`) and creates new individuals having `JgapImageFunction` objects as their genotype (these can be used by the `VectorFunctionImageCreator`, as `JgapImageFunction` extends `ImageFunction` and can return any type that the underlying `IGPProgram` generates.

A.3.2 Image evaluation

Image evaluation centers around the two evaluation interfaces `PopulatorEvaluator` and `IndividualEvaluator`, which provide, similar to the `PhenotypeCreator` interface, methods to evaluate individuals and populations. The individual evaluators provide the method `V evaluatorIndividual(Individual)` which returns the evaluation of the individual, while the population evaluators manage statistics and registering evaluations (even combining evaluations from several individual evaluators) and multi-threaded indi-

vidual evaluations.

The two GUIs (one for interactive and one for automatic evaluation) implement the population evaluation interface (the interactive evaluation interface also implements the individual evaluation interface because the individuals are evaluated directly in the GUI by the user).

A.3.3 The application framework

The goal of the application framework is to create reusable and extensible structures which can be used to rapidly develop evolutionary art applications. Figure A.6 gives an overview of the most important classes and interfaces of the application framework. Central is the scenario interface, which basically represents an application in the context of this framework. An abstract implementation of a scenario is provided with `AbstractScenario`. This class consists of a `FrameworkDriver`, which is supposed to configure and communicate with the GP framework which executes the actual evolutionary process, and an `EvolutionEngine`. The `EvolutionEngine` bundles phenotype creation and image evaluation and is called by the `FrameworkDriver` whenever the GP framework asks for evaluations of GP programs. The `Configurator` interface can be used to configure different components used in the scenarios.

The concrete scenarios are all part of the jpea-jgap module: The two main scenarios are `InteractiveScenario` and `AutomaticScenario`, both of which are extending the abstract `JgapScenario` which provides a `JgapDriver`, GUI configuration and general logic needed in most scenarios.

The class `AutomaticScenario` further extends this so that concrete automatic scenarios only have to provide a population evaluator and a title (and can, optionally, override which phenotype creator or which primitive set to use).

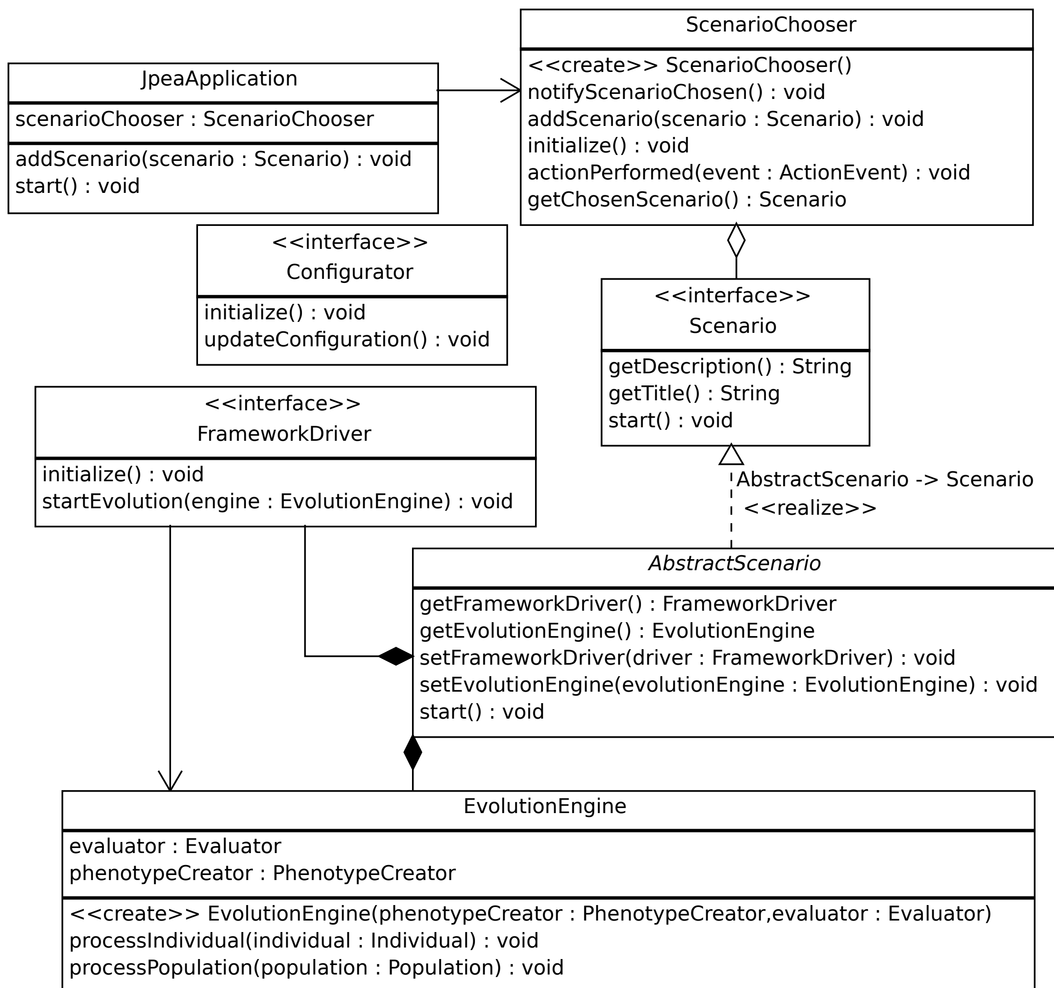


Figure A.6: Overview of the important classes and interfaces of the application package.

Bibliography

Abbott, Russell J. (Sept. 2003). "Object-Oriented Genetic Programming, An Initial Implementation". In: *Proceedings of the Sixth International Conference on Computational Intelligence and Natural Computing*. Vol. 1. Embassy Suites Hotel and Conference Center, Cary, North Carolina USA.

Atmar, W. (1994). "Notes on the simulation of evolution". In: *Neural Networks, IEEE Transactions on* 5.1, pp. 130–147. ISSN: 1045-9227. DOI: 10.1109/72.265967.

Bäck, Thomas (1996). *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford, UK: Oxford University Press. ISBN: 0-19-509971-0.

Bäck, Thomas, David B. Fogel, and Zbigniew Michalewicz, eds. (1997). *Handbook of Evolutionary Computation*. 1st. Bristol, UK: IOP Publishing Ltd. ISBN: 0750303921.

— eds. (1999). *Basic Algorithms and Operators*. 1st. Bristol, UK: IOP Publishing Ltd. ISBN: 0750306645.

Bäck, Thomas, Frank Hoffmeister, and Hans-Paul Schwefel (1991). "A Survey of Evolution Strategies". In: *Proceedings of the Fourth International Conference on Genetic Algorithms*. Vol. 1. Morgan Kaufmann, pp. 2–9.

Baluja, Shumeet, Dean Pomerleau, and Todd Jochem (1994). "Towards Automated Artificial Evolution for Computer-generated Images". In: *Connection Science* 6.2-3, pp. 325–354. DOI: 10.1080/09540099408915729. URL: http://www.tandfonline.com/doi/abs/10.1080/09540099408915729 (visited on 08/25/2013).

Banzhaf, Wolfgang et al. (Jan. 1998). *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. San Francisco, CA, USA: Morgan Kaufmann. ISBN: 1-55860-510-X. URL: http://www.elsevier.com/wps/find/bookdescription.cws_home/677869/description#description (visited on 08/25/2013).

Bleuler, Stefan et al. (2001). "Multiobjective Genetic Programming: Reducing Bloat Using SPEA2". In: *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*. Vol. 1. IEEE Press, pp. 536–543. ISBN: 0-7803-6658-1. DOI: doi:10.1109/CEC.2001.934438. URL: ftp://ftp.tik.ee.ethz.ch/pub/people/zitzler/BBTZ2001b.ps.gz (visited on 08/25/2013).

Bruce, Wilker Shane (Dec. 1995). "The Application of Genetic Programming to the Automatic Generation of Object-Oriented Programs". PhD thesis. School of Computer and Information Sciences, Nova Southeastern University. URL: http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/bruce.thesis.ps.gz (visited on 08/25/2013).

Chen, Wen-Shiung et al. (2001). "Algorithms to estimating fractal dimension of textured images". In: *Acoustics, Speech, and Signal Processing, 2001. Proceedings.(ICASSP'01). 2001 IEEE International Conference on*. Vol. 3. IEEE, pp. 1541–1544.

Dawkins, Richard (1986). *The blind watchmaker: Why the evidence of evolution reveals a universe without design*. WW Norton & Company.

Deb, K. et al. (2002). "A fast and elitist multiobjective genetic algorithm: NSGA-II". In: *Evolutionary Computation, IEEE Transactions on* 6.2, pp. 182–197. ISSN: 1089-778X. DOI: 10.1109/4235.996017.

Ekárt, Anikó, Divya Sharma, and Stayko Chalakov (2011). "Modelling human preference in evolutionary art". In: *Proceedings of the 2011 international conference on Applications of evolutionary computation - Volume Part II*. EvoApplications'11. Torino, Italy: Springer-Verlag, pp. 303–312. ISBN: 978-3-642-20519-4. URL: http://dl.acm.org/citation.cfm?id=2008445.2008480 (visited on 08/25/2013).

Hart, David (2006). “Toward greater artistic control for interactive evolution of images and animation”. In: *ACM SIGGRAPH 2006 Sketches*. Vol. 1. SIGGRAPH ’06. Boston, Massachusetts: ACM. ISBN: 1-59593-364-6. DOI: 10.1145/1179849.1179852. URL: http://doi.acm.org/10.1145/1179849.1179852 (visited on 08/25/2013).

Heijer, E. den and A. E. Eiben (2010a). “Comparing aesthetic measures for evolutionary art”. In: *Proceedings of the 2010 international conference on Applications of Evolutionary Computation - Volume Part II*. EvoCOMNET’10. Istanbul, Turkey: Springer-Verlag, pp. 311–320. ISBN: 3-642-12241-8, 978-3-642-12241-5. DOI: 10.1007/978-3-642-12242-2_32.

— (2011). “Evolving art using multiple aesthetic measures”. In: *Proceedings of the 2011 international conference on Applications of evolutionary computation - Volume Part II*. EvoApplications’11. Torino, Italy: Springer-Verlag, pp. 234–243. ISBN: 978-3-642-20519-4. URL: http://dl.acm.org/citation.cfm?id=2008445.2008473 (visited on 03/27/2013).

Heijer, Eelco den and AE Eiben (2010b). “Using aesthetic measures to evolve art”. In: *Evolutionary Computation (CEC), 2010 IEEE Congress on*. Vol. 1. IEEE, pp. 1–8.

Holland, J (1975). “Adaptation in artificial and natural systems”. In: *Ann Arbor: The University of Michigan Press*.

Jolion, Jean-Michel (Feb. 2001). “Images and Benford’s Law”. In: *J. Math. Imaging Vis.* 14.1, pp. 73–81. ISSN: 0924-9907. DOI: 10.1023/A:1008363415314. URL: http://dx.doi.org/10.1023/A:1008363415314 (visited on 08/25/2013).

Koza, John R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press. ISBN: 0-262-11170-5.

— (1994). *Genetic programming II: Automatic Discovery of Reusable Programs*. Complex adaptive systems. MIT Press, pp. I–XX, 1–746. ISBN: 978-0-262-11189-8.

Lewis, Matthew (2008). "Evolutionary Visual Art and Design". In: *The Art of Artificial Evolution*. Ed. by Juan Romero and Penousal Machado. Natural Computing Series. Springer Berlin Heidelberg, pp. 3–37. ISBN: 978-3-540-72876-4. DOI: 10.1007/978-3-540-72877-1_1. URL: http://dx.doi.org/10.1007/978-3-540-72877-1_1 (visited on 08/25/2013).

Li, Jian, Qian Du, and Caixin Sun (Nov. 2009). "An improved box-counting method for image fractal dimension estimation". In: *Pattern Recogn.* 42.11, pp. 2460–2469. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2009.03.001. URL: http://dx.doi.org/10.1016/j.patcog.2009.03.001 (visited on 08/25/2013).

Lucas, Simon (Apr. 2004). "Exploiting Reflection in Object Oriented Genetic Programming". In: *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*. Ed. by Maarten Keijzer et al. Vol. 3003. LNCS. Coimbra, Portugal: Springer-Verlag, pp. 369–378. ISBN: 3-540-21346-5. URL: http://algoval.essex.ac.uk/rep/oogp/ReflectionBasedGP.pdf (visited on 08/25/2013).

Machado, Penousal and Amílcar Cardoso (1998). "Computing Aesthetics". In: *Advances in Artificial Intelligence*. Vol. 1515. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 219–228. ISBN: 978-3-540-65190-1. DOI: 10.1007/10692710_23. URL: http://dx.doi.org/10.1007/10692710_23 (visited on 08/25/2013).

— (2002). "All the Truth About NEvAr". English. In: *Applied Intelligence* 16.2, pp. 101–118. ISSN: 0924-669X. DOI: 10.1023/A:1013662402341. URL: http://dx.doi.org/10.1023/A%3A1013662402341 (visited on 08/25/2013).

Matković, Krešimir et al. (2005). "Global contrast factor - a new approach to image contrast". In: *Proceedings of the First Eurographics conference on Computational Aesthetics in Graphics, Visualization and Imaging*. Vol. 1. Computational Aesthetics'05. Girona, Spain: Eurographics Association, pp. 159–167. ISBN: 3-905673-27-4. DOI: 10.2312/COMPAESTH/COMPAESTH05/159-167.

URL: http://dx.doi.org/10.2312/COMPAESTH/COMPAESTH05/159-167 (visited on 08/25/2013).

McCormack, Jon (2005). “Open Problems in Evolutionary Music and Art”. In: *Applications of Evolutionary Computing*. Ed. by Franz Rothlauf et al. Vol. 3449. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 428–436. ISBN: 978-3-540-25396-9. DOI: 10.1007/978-3-540-32003-6_43. URL: http://dx.doi.org/10.1007/978-3-540-32003-6_43 (visited on 08/25/2013).

Miller, Brad L. and David E. Goldberg (1995). “Genetic Algorithms, Tournament Selection, and the Effects of Noise”. In: *Complex Systems* 9, pp. 193–212.

Miller, Julian F., ed. (2011). *Cartesian Genetic Programming*. Natural Computing Series. Springer. DOI: doi:10.1007/978-3-642-17310-3. URL: http://www.springer.com/computer/theoretical+computer+science/book/978-3-642-17309-7 (visited on 08/25/2013).

Montana, David J. (1995). “Strongly Typed Genetic Programming”. In: *Evolutionary Computation* 3.2, pp. 199–230. DOI: doi:10.1162/evco.1995.3.2.199. URL: http://personal.d.bbn.com/~dmontana/papers/stgp.pdf (visited on 08/25/2013).

Poli, Riccardo, William B. Langdon, and Nicholas Freitag McPhee (2008). *A field guide to genetic programming*. (With contributions by J. R. Koza). Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk. URL: http://www.gp-field-guide.org.uk (visited on 08/25/2013).

Ralph, W (2006). “Painting the Bell Curve: The Occurrence of the Normal Distribution in Fine Art”. preparation.

Rechenberg, Ingo (1965). “Cybernetic solution path of an experimental problem”. In:

Rigau, J., M. Feixas, and M. Sbert (2008). “Informational Aesthetics Measures”. In: *Computer Graphics and Applications, IEEE* 28.2, pp. 24–34. ISSN: 0272-1716. DOI: 10.1109/MCG.2008.34.

Ross, Brian J., William Ralph, and Hai Zong (2006). "Evolutionary Image Synthesis Using a Model of Aesthetics". In: *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*. Ed. by Gary G. Yen et al. Vol. 1. Vancouver: IEEE Press, pp. 3832–3839. ISBN: 0-7803-9487-9. DOI: `doi:10.1109/CEC.2006.1688430`. URL: `http://www.cosc.brocku.ca/~bross/research/CEC2006.pdf` (visited on 08/25/2013).

Ross, Brian J. and Han Zhu (July 2004). "Procedural texture evolution using multi-objective optimization". In: *New Gen. Comput.* 22.3, pp. 271–293. ISSN: 0288-3635. DOI: `10.1007/BF03040964`. URL: `http://dx.doi.org/10.1007/BF03040964` (visited on 08/25/2013).

Rueckert, Johannes (2013). "Reflection-basierte Genetische Programmierung am Beispiel Evolutionärer Kunst". Project thesis. Fachhochschule Dortmund.

Schwefel, Hans-Paul (1965). "Kybernetische Evolution als Strategie der experimentellen Forschung in der Strömungstechnik". In: *Master's thesis, Technical University of Berlin*.

— (1975). "Evolutionsstrategie und numerische Optimierung". PhD thesis. Technische Universität Berlin.

— (1977). *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie – Mit einer vergleichenden Einführung in die Hill-Climbing- und Zufallsstrategie*. Basel: Birkhäuser. ISBN: 978-3-764-30876-6.

— (1981). *Numerical Optimization of Computer Models*. New York, NY, USA: John Wiley & Sons, Inc. ISBN: 0471099880.

— (1987). *Collective phenomena in evolutionary systems*. Dortmund Dekanat Informatik, Univ.

Shirriff, Ken W (1993). "An investigation of fractals generated by $z \to 1/z^n + c$". In: *Computers & graphics* 17.5, pp. 603–607.

Sims, Karl (July 1991). "Artificial evolution for computer graphics". In: *SIGGRAPH Comput. Graph.* 25.4, pp. 319–328. ISSN: 0097-8930. DOI: `10.1145/127719.122752`. URL: `http://www.karlsims.com/papers/siggraph91.html` (visited on 08/25/2013).

Spehar, Branka et al. (2003). “Universal aesthetic of fractals”. In: *Computers & Graphics* 27.5, pp. 813–820.

Takagi, Hideyuki (2001). “Interactive evolutionary computation: Fusion of the capabilities of EC optimization and human evaluation”. In: *Proceedings of the IEEE* 89.9, pp. 1275–1296.

Union, International Telecommunication (2011). *Recommendation ITU-R BT.601-7, Studio encoding parameters of digital television for standard 4:3 and wide-screen 16:9 aspect ratios*. URL: http://www.itu.int/rec/R-REC-BT.601-7-201103-I/en (visited on 08/25/2013).

Veldhuizen, David A. van and Gary B. Lamont (2000). “Multiobjective Evolutionary Algorithms: Analyzing the State-of-the-Art”. In: *Evolutionary Computation* 8.2, pp. 125–147.

Wolpert, D. H. and W. G. Macready (Apr. 1997). “No free lunch theorems for optimization”. In: *Trans. Evol. Comp* 1.1, pp. 67–82. ISSN: 1089-778X. DOI: 10.1109/4235.585893. URL: http://dx.doi.org/10.1109/4235.585893 (visited on 08/25/2013).

Wolpert, David H and William G Macready (1995). *No free lunch theorems for search*. Tech. rep. SFI-TR-95-02-010, Santa Fe Institute.