Markus Müller, BSc

Extended Spatially Structured Cellular Evolutionary Algorithm in 2D Space with FREVO

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

Studium: Information and Communications Engineering (ICE)

Studienzweig: Networks and Communications (NC)

Alpen-Adria-Universität Klagenfurt

Begutachter: Univ.–Prof. Dr. techn. Wilfried Elmenreich

Institut für Vernetzte und Eingebettete Systeme

September 2019

Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich

- die eingereichte wissenschaftliche Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe,
- die während des Arbeitsvorganges von dritter Seite erfahrene Unterstützung, einschließlich signifikanter Betreuungshinweise, vollständig offengelegt habe,
- die Inhalte, die ich aus Werken Dritter oder eigenen Werken wortwörtlich oder sinngemäß übernommen habe, in geeigneter Form gekennzeichnet und den Ursprung der Information durch möglichst exakte Quellenangaben (z.B. in Fußnoten) ersichtlich gemacht habe,
- die eingereichte wissenschaftliche Arbeit bisher weder im Inland noch im Ausland einer Prüfungsbehörde vorgelegt habe und
- bei der Weitergabe jedes Exemplars (z.B. in gebundener, gedruckter oder digitaler Form) der wissenschaftlichen Arbeit sicherstelle, dass diese mit der eingereichten digitalen Version übereinstimmt.

Mir ist bekannt, dass die digitale Version der eingereichten wissenschaftlichen Arbeit zur Plagiatskontrolle herangezogen wird.

Ich bin mir bewusst, dass eine tatsachenwidrige Erklärung rechtliche Folgen haben wird.

Markus Müller e. h. Klagenfurt, September 2019

Abstract

This work presents an improvement of the existing SSCEA2D (Spatially Structured Cellular Evolutionary Algorithm in 2D Space) in FREVO (FRamework for EVOlutionary Design), a simulation program for physical, biological and technical optimization tasks. The SSCEA2D method defines a lattice grid with candidate solutions of a problem and they are called individuals. Each individual executes an evolutionary algorithm with its neighbors and thus evolves over generations. In the context of this work, evolutionary algorithms are used to define rules which are applied in Self-Organizing Systems and Cyber Physical Systems.
Before it was only possible to use a squared lattice grid for the evolution of individuals in FREVO. This master thesis extends the previous implementation to allow for setting the height and width of a lattice grid separately (rectangular grids). Additionally, it is possible to integrate non-working individuals, which are called obstacles in this context. Each obstacle can be distributed randomly inside the lattice grid and also some predefined obstacle patterns with fixed positions are available. With these new features we have investigated the behavior of evolution within the grid and the impact in different simulation runs. Measures for describing results are the fitness (also called objective function or solution feedback) and diversity (a measure of how many different solutions there are) of the evolved individuals.
Two reference problems in FREVO were used to evaluate the new SSCEA2D. First one is the Light! problem and features a robot searching a light source. With different initial states (seeds) and simulation setups (with/without obstacles, rectangular and squared grid, grid and random neighborhood) the new features of the optimization algorithm were examined. The scattering of all runs per simulation run (fitness and diversity by last generation) is visualized via boxplots. Furthermore, we investigated how the results are matching existing statistical distributed. The three most common distributions are shown and described at the end. The second reference problem one is the Simplified Robot Soccer simulation. With two different seeds and similar simulation setups we obtained results through a simulated soccer tournament.

Zusammenfassung

Diese Arbeit präsentiert eine Verbesserung des bereits existierenden SSCEA2D (engl. Spatially Structured Cellular Evolutionary Algorithm in 2D Space) in FREVO (FRamework for EVOlutionary Design), ein Simulationsprogramm für physikalische, biologische und technische Optimierungsaufgaben. Die SSCEA2D Methode definiert ein Gitternetz mit Lösungskandidaten für ein Problem und werden Individuen genannt. Jedes Individuum führt einen evolutionären Algorithmus mit seinen nächsten Nachbarn aus und entwickelt sich über Generationen weiter. Evolutionäre Algorithmen haben definierte Regeln und werden in Selbstorganisierende Systemen und Cyber Physikalischen Systemen angewandt.
Vorher war es in FREVO lediglich möglich ein quadratisches Gitternetz für die Evolution von Individuen zu verwenden. Diese Masterarbeit erweitert die bisherige Implementierung um einen Ansatz zum Einstellen der Höhe und Breite eines Gitternetzes (rechteckige Netze). Zusätzlich ist es möglich sogenannte nicht-arbeitende Individuen zu integrieren. Diese werden Obstacles (dt. "Hindernisse") genannt. Jedes Obstacle kann zufällig im Gitternetz verteilt werden. Ebenfalls gibt es auch definierte Vorlagen für Obstacles mit fixer Position. Mit diesen neuen Möglichkeiten wurde das evolutionäre Verhalten und die Einflüsse innerhalb der Gitternetze mit verschiedenen Simulationen untersucht. Dazu gibt es für die entwickelten Individuen zwei Maße für die Beschreibung von Ergebnissen und sie heißen Fitness (auch Objective Funktion oder Lösungsfeedback) und Diversität (sagt aus, wie viele verschieden Lösungsansätze es während der Evolution gibt).
Es wurden zwei Referenzprobleme in FREVO für den neuen SSCEA2D genutzt. Das erste ist das "Light!" Problem und stellt einen Roboter dar, der eine Lichtquelle sucht. Mit verschieden Anfangszuständen (engl. Seeds) und Simulationssetups (mit/ohne Obstacles, rechteckiges und quadratisches Netz, Netz- und zufällige Nachbarschaft) wurden die neuen Features untersucht. Mit Boxplots kann man die Streuung der Ergebnisse pro Simulation sehen. Zusätzlich wurde auch die Wahrscheinlichkeit diskutiert, wie die Boxplot-Ergebnisse verteilt sind. Die drei häufigsten Verteilungen werden schließlich beschrieben. Das zweite Referenzproblem ist die "Simplified Robot Soccer" Simulation. Mit zwei verschiedenen Anfangszuständen und ähnlichen Simulationssetups wurden die Ergebnisse anhand eines Fußballturniers verglichen und simuliert.

Acknowledgments

This master thesis was written in collaboration with the AAU Klagenfurt and project CPSwarm. I have worked as project staff of the university and author with a contract of 20 hours per week. For this reason I want to thank the following colleagues for realizing this thesis and support: Professor Dr. Wilfried Elmenreich, Dipl.-Ing. Midhat Jdeed and Dipl.-Ing. Arthur Pitman. Likewise I want to thank the University of Klagenfurt for the opportunity to write a master thesis on this topic.

List of Acronyms

AAU = **A**lpen - **A**dria - **U**niversität

AI = **A**rtificial **I**ntelligence

ANN = **A**rtificial **N**eural **N**etwork

CAM = **C**ellular **A**utomata **M**orphogenesis

CDF = **C**umulative **D**istribution **F**unction

CPS = **C**yber **P**ysical **S**ystem

EA = **E**volutionary **A**lgorithm

EU = **E**uropean **U**nion

FMN = **F**ully **M**eshed **N**et

FREVO = **FR**amework for **EVO**lutionary design

NES = **N**etworked and **E**mbedded **S**ystems

PCB = **P**rinted **C**iruit **B**oard

PDF = **P**robability **D**ensity **F**unction

SOS = **S**elf-**O**rganizing **S**ystem

SSCEA2D = **S**patially **S**tructured **C**ellular **E**volutionary **A**lgorithm in **2D** Space

TLNN = **T**hree **L**ayered **N**eural **N**etwork

USB = **U**niversal **S**erial **B**us

XOR = e**X**clusive **OR**

Contents

List of Figures viii

List of Tables x

List of Equations xi

List of Algorithms xii

Listings xiii

1 Introduction 1
- 1.1 Background and Motivation 1
- 1.2 Definitions 3
  - 1.2.1 Self-Organizing Systems 3
  - 1.2.2 Simulation 6
  - 1.2.3 Topology vs. Neighborhood 7
  - 1.2.4 Fitness 13
  - 1.2.5 Diversity 14
  - 1.2.6 FREVO 15
  - 1.2.7 Neural Networks 17

2 Problem Description 22
- 2.1 Purpose 22
- 2.2 Research Questions 22
- 2.3 Tools and Methods 23

3 Implementation 24
- 3.1 Rectangular Grid 24
- 3.2 Obstacles 25
- 3.3 Source code 29
  - 3.3.1 Setting obstacles, patterns, parameters and plotting diversity 29
  - 3.3.2 Reading text files for generating boxplots 29
  - 3.3.3 New parameters for FREVO interface 29

4 Simulation Comparisons 30

4.1 Preparations 30
4.2 Fitness and Diversity Results 36
4.2.1 Light! 37
4.2.2 Simplified Robot Soccer 42
4.3 Analysis of the Distribution 44
4.3.1 Distributions for Light! 45
4.3.2 Description of Distributions 47
4.3.2.1 Gumbel l 47
4.3.2.2 Fisk 48
4.3.2.3 Logistic 50
4.3.2.4 Discussion 51
4.4 Soccer tournament 52

5 Conclusions and outlook **53**

Bibliography **55**

Appendices **59**

A Setting obstacles, patterns, parameters and plotting diversity **60**

B Reading text files for generating boxplots **75**

C New parameters for FREVO interface **76**

List of Figures

1.1 Comparison between panmixia and structured evolution 2
1.2 Fish swim in an organized order 4
1.3 Evolutionary design approach for SOS [16] 5
1.4 Relation between reality and abstraction for simulations 7
1.5 Structured population in a lattice grid [38] 8
1.6 Toroid with grid 8
1.7 Von Neumann neighborhood and one-dimensional ring topology . . . 9
1.8 Evolution of each member as a life cycle [33] 10
1.9 FREVO user interface (version 1.4.1) 15
1.10 Robot searches a light source in FREVO 16
1.11 Soccer game with AI in FREVO 17
1.12 Simple ANN [31] 18
1.13 Block diagram for ANNs [31] 18
1.14 TLNN vs. FMN 19
1.15 Coordinates for mapping each pixel 20
1.16 Numbers between 0 and 9 20
1.17 8x8 net for recognizing numbers with TLNN 21
1.18 Symbols with FMN 21
3.1 Grid with 8 width and 10 height 25
3.2 Grid with 12 width and 6 height 25
3.3 Pattern 1 26
3.4 Pattern 2 26
3.5 Pattern 3 27
3.6 Pattern 4 and 10x15 grid 28
4.1 Running simulation on simulation server 30
4.2 Data management on WinSCP 31
4.3 Parameters of Boxplot 36
4.4 Fitness results (FMN) of 0-25 with Light! 38
4.5 Diversities of Fully Meshed Net (run 0-25) with Light! 39
4.6 Fitness results (TLNN) of 26-51 with Light! 39
4.7 Diversities of Three Layered Neural Network (run 26-51) with Light! . 40
4.8 Diversity 12345 43
4.9 Diversity 11111 43
4.10 Left sided Gumbel CDF 47

4.11 Left sided Gumbel PDF 48
4.12 Fisk CDF 49
4.13 Fisk PDF 49
4.14 Logistic CDF 50
4.15 Logistic PDF 51

List of Tables

1.1 Truth table of an exclusive or (XOR) gate (two inputs and one output) 13
4.1 Overview of simulations with Light! 32
4.2 Overview of simulations with Simplified Soccer 33
4.3 Settings for Light! 34
4.4 Settings for Simplified Robot Soccer 34
4.5 Settings for CEA2D 34
4.6 Settings for FMN 35
4.7 Settings for TLNN 35
4.8 Overview of maximum fitness and average diversity for Light! 37
4.9 Comparison with population size 100 41
4.10 Comparison with population size 64 41
4.11 Comparison of predefined shapes 41
4.12 Comparison of population size 50 41
4.13 Overview of maximum fitness and diversity for Simplified Robot Soccer (last generation) 42
4.14 Fitness distributions with highest probabilities for Light! 45
4.15 Diversity distributions with highest probabilities for Light! 46
4.16 Points table for soccer teams 52

List of Equations

1.1 Negative mean square error 13
1.2 Fitness for CAM 13
1.3 Stable and non-oscillating fitness for CAM 14
1.4 Social entropy 14
1.5 How to calculate dual logarithm 14
1.6 How to calculate net for generating output 17
1.7 Output Y of ANN 18
4.1 Mean value 40
4.2 CDF of Gumbel l distribution 47
4.3 PDF of Gumbel l distribution 47
4.4 CDF of Fisk distribution 48
4.5 PDF of Fisk distribution 48
4.6 CDF of Logistic distribution 50
4.7 PDF of Logistic distribution 50
4.8 Number of matches 52

List of Algorithms

1.1 Principle of EA [41] 11
1.2 Pseudo-code for cellular EA [41] 12

Listings

A.1 Parameters.java 60
A.2 Population.java 63
A.3 CEA2D.java 67
B.1 boxplot.py 75
C.1 CEA2D.xml 76

Chapter 1

Introduction

1.1 Background and Motivation

Evolutionary Algorithms (EA) are stochastic, meta-heuristic methods and find application in various optimization problems. EAs were inspired by natural creatures and they evolve through evolving operations over a certain amount of generations. This kind of algorithm is also used in the FRamework for EVOlutionary design (FREVO), a simulation program for investigating reference problems and to find optimized solutions. Based on our understanding we can already predict some essential differences between panmixia (random mating) and structured (mating considering the closest neighbors) algorithms. Figure 1.1 shows an example of fitness evolution. Hint: This graph does not base on any measuring results, rather it gives an idea how the graphical behavior of panmixia and structure roughly look like. Which algorithm is more suitable depends on the problem complexity and duration of evolution. Panmixia has at the beginning a higher fitness, but for long-term simulations the SSCEA2D (Spatially Structured Cellular Evolutionary Algorithm in 2D Space) overtakes. In the structured approach, each individual (candidate solution) considers its closest neighbors inside an area and each generation will be executed by selection, recombination, mutation and replacement. In panmixia, the neighborhood does not play a role, rather how high the fitness and mating probability equals. Ether each individual has the same probability for mating or the probability depends on the fitness (i.e. higher fitness leads to higher probability and vice versa).

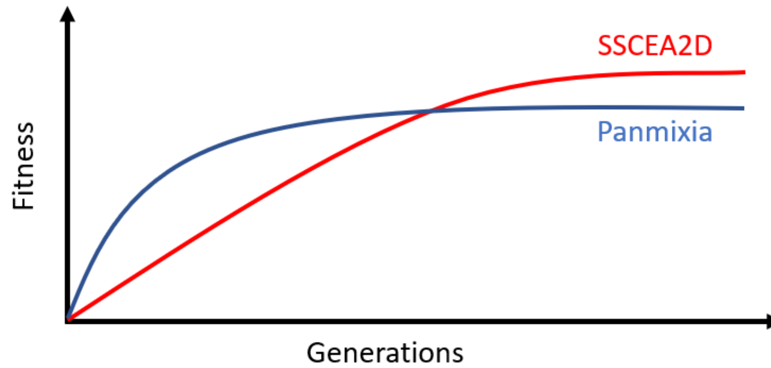

Figure 1.1: Comparison between panmixia and structured evolution

Because of this advantage after more generations, the SSCEA2D will use for researching. By improving and developing this algorithm we can integrated more optimized solutions in scientific and technical applications.

The vision of autonomous systems received attention since the end of 19th century and until now stayed very relevant in research. To contribute this effort, the project "CPSwarm" [2, 36] was launched in collaboration with international research institutions and companies of the European Union (EU). The Institute Networked and Embedded Systems (NES) of the Alpen-Adria-Universität (AAU) in Klagenfurt as one of the project partners is responsible for designing the single/population tools and swarm modeling. Other partners of the project are: Fraunhofer FIT (Germany), SOFTEAM Cadexian (France), Robotnik (Spain), DigiSky (Italy), LINKS (Italy), SEARCH-LAB (Hungary), Lakeside Labs (Austria) and TTTech (Austria).
The duration of the project is a time span of three years from January, 1st 2017 until December, 31st 2019 and the expected outcome after this deadline includes an open-source tool-chain (simulation program) with the following features:

1. Setting up autonomous CPSs
2. Test swarm performance
3. Deploy solutions in CPS devices

Because of the high potential in research for swarm and Artificial Intelligence (AI) domain, CPSwarm gives students an opportunity to take part into the development. Each contribution of innovative ideas gain the implementation for scientists in the future.
We can obtain already now autonomous appliances like robots as vacuum cleaners or mowers for households and lawns. If more cyber physical devices are applied and they collaborate, we speak about swarm robotics. Purpose of this vision is to save and ease our daily life in different scenarios.

On the other hand, Cyber Physical Systems (CPSs) and AI find application in our daily traffic in towns. An approach of swarm intelligence could be to limit accidents of vehicles. Human carelessness and distraction caused in the past decades until now these problems.

For these examples the approach of evolutionary design as essential part is necessary. A couple of years ago, the simulation program FREVO was developed by the institute NES. With this framework it is possible to takeover the simulation part for the tool-chain of CPSwarm. FREVO consists of a cellular EA, which finds applications in CPSs and SOSs as well. With this algorithm are improvements implemented and may find efficient solution approaches for swarm robotics. EAs have the big advantage for solving multi-objective optimization and searching tasks. These algorithms find applications in economy, engineering and sciences. Benefit of this evolutionary approach is to takeover the rules in nature and to apply in biological, industrial or technical scenarios. In other words, EAs are nature-inspired. These properties make EAs as suitable approach for swarm intelligence and robotics.

1.2 Definitions

1.2.1 Self-Organizing Systems

This section addresses the connection between Self-Organizing Systems (SOSs) and CPSs. Both of them are independent kinds of systems but they can combined together. Difference between them is that CPSs consist of technical parts where SOSs can also have biological or chemical components. If we apply robots in swarm technology, we use both systems for this domain. In other words for differentiation, a SOS does not necessarily consist of technical elements, but a CPS does. SOS play also an interdisciplinary role in AI, complex systems, cybernetics and biology [11].
A SOS describes the compilation of working processes without external influences. Behavior of these systems show the integrated individuals and they are responsible for the duration and quality of evolution. SOS have the property, that there is no leadership between individuals and they are decentralized. Which kinds of SOSs exist and how can we imagine this complexities? The origin of this phenomenon was found in nature [4], for example bees [26], wasps, bee orchids [21], fish, birds and ants for search the closest route for finding food [6] or ant clustering [30]. All of these animals work in a defined manner and each individual has a role in the SOS. Ants build for instance an anthill and they carry parts for finishing their nest. On the other hand, fish and birds swim/fly in a coordinated way without a defined rule. Each participant knows its own role and how to move itself.
But why do we need these biology processes in engineering? For the past years, scientists research to adapt the rules of nature to technology. These kind of rules may be integrated in software solutions for AI as algorithms. Nature algorithms are realized e.g. within fireflies and bats [23]. The most important category of algorithms for this master thesis will be those implements of swarm intelligence

[20, 7]. Figure 1.2 shows an example for a swarm dynamic in nature. Fish interact in peer-to-peer behavior (each fish has the same ranking and we have no leaders in the swarm).
SOSs in engineering have often a meta-heuristic property in designing (approximate a solution of optimizing problems). The general designation of these algorithms is called EA [32]. Rules inside EA may i.e. be approximated by mathematical differential equations [12, 22].

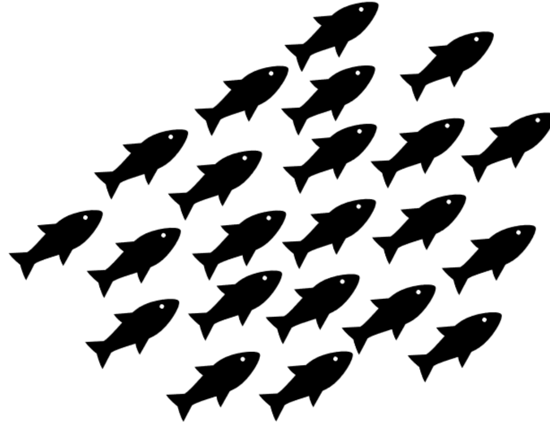

Figure 1.2: Fish swim in an organized order

It follows, that we can find SOS nowadays in many applications as geography [40], quantum computing [19], searching [28], eco- and industrial networks [8]. Natural evolution is based on the processes of diversity creation within a population evaluation of fitness and selection [27].
SOS have three main characteristics [34]. First of them is robustness and means that the system can fix failures or damages without external influence. A working SOS does not break down suddenly because of changes within the system. Another characteristic describes the ability to adapt changes in the system or environment (adaptability). Involved entities are responsible for continuously adaptions. The third property is defined as scalability and means that a SOS still functions even the number of entities is very large. Example could be as in Figure 1.2 when the population of fish increase in a very high amount and the self-organization still functions. Fish have an decentralized behavior because within the swarm is no centralized leader present and each fish observes its neighbors while swimming. This all makes SOS an interesting concept for networked technical applications.
Designing a SOS depends on the local rules for the behavior of each individuals. Often is designing done by trial and error processes. In high complex systems are these methods too inefficient or even impossible to realize. Unpredictable results may be also caused by small change of simulating parameters as a consequence. On the other hand, evolutionary methods provide means to optimize these parameters efficiently and automatize the testing of SOS [14]. Note that SOS not always base on

evolutionary methods because it exists SOS without evolution as well (non-biologic processes) [11].

Evolving a SOS requires six major components [16] which are depicted in Figure 1.3:

1. **Task Description:** Which kind of problem is given and has to be solved? The task description gives also information about which outcome (objective) is expected to solve the given problem.

2. **Simulation Setup:** Which configurations are needed and possible? In this area we plan a referring model (see next sub-chapter) to the task description. Models represent important aspects of the system and have efficient properties.

3. **Interaction Interface:** Plans the way how system components interact with each other and their environment. This part of the system is responsible for communication (sensors) and interfaces (protocols).

4. **Evolvable Decision Unit:** Focuses the actual representation of components. This unit is separated from the system model because a evolutionary method need evolvable representations.

5. **Objective Function (or fitness):** Defined as quality of the individuals solution. Also this parameter describes the intelligence of each individual. We can measure the objective function as for example a relative (number of won games in soccer) or an absolute value (between 0 and 1 as in percent).

6. **Search Algorithms:** They are typically meta-heuristic search algorithms and have the ability to find a global cost minimum. Optimizing the candidate solutions is the main purpose. The choice of the search algorithm can influence the quality of results.

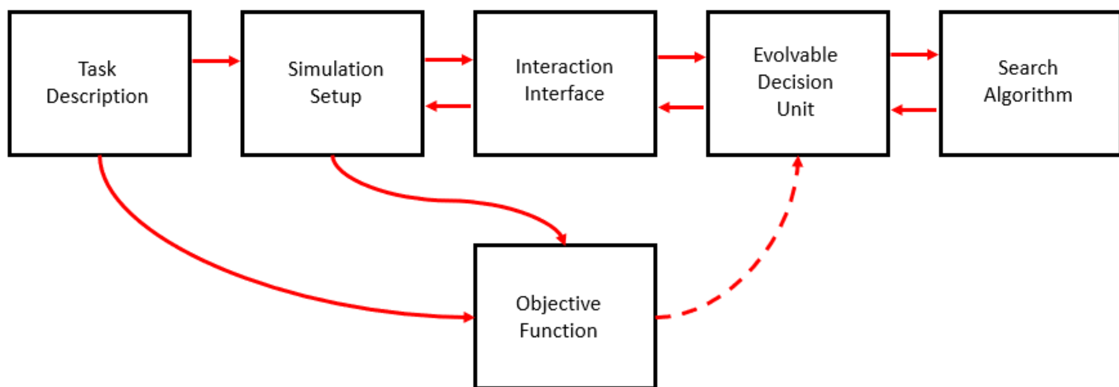

Figure 1.3: Evolutionary design approach for SOS [16]

1.2.2 Simulation

Each simulation is based on a problem and describes a system for analyzing and solving. For describing problems we need models (physical, mathematical). It exists roughly two types of modeling:

- Explanatory: Simulation shows why and how a system works or a phenomenon occurs. Thus, no new properties or events are expected. Explanatory simulations show already proved systems from the past.

- Exploratory: Prediction of new and probably unexpected behaviors. Not all simulations have a strict line of rules.

Next to the types of simulations, we have also to consider in which dimension we simulate (1D, 2D or 3D simulations [29, 10]). We discuss in this thesis exclusively about the two-dimensional domain.
Systems are a collection of individuals to get the accomplishment of some conclusive end. The information, which describes the system at a given moment is called state. To solve a given problem, it is necessary to design a model. We have to consider that simulation means an abstraction of the real environment. An overview between reality and simulation is visible in Figure 1.4.
Simulating of SOS and evolutionary approach describes the core topic of this master thesis. Evolution can be simulated as for example a biology-inspired process in nature. Evolutionary approaches consists of the steps selection, recombination and mutation. These processes can be represented by evolutionary algorithms for simulating scientific and technical scenarios. With FREVO it is possible to realize this approach. The user can set, how many generations are needed for any simulations. The sum of all generations is called evolution. More details about the simulation program FREVO will discussed in a following sub-chapter.

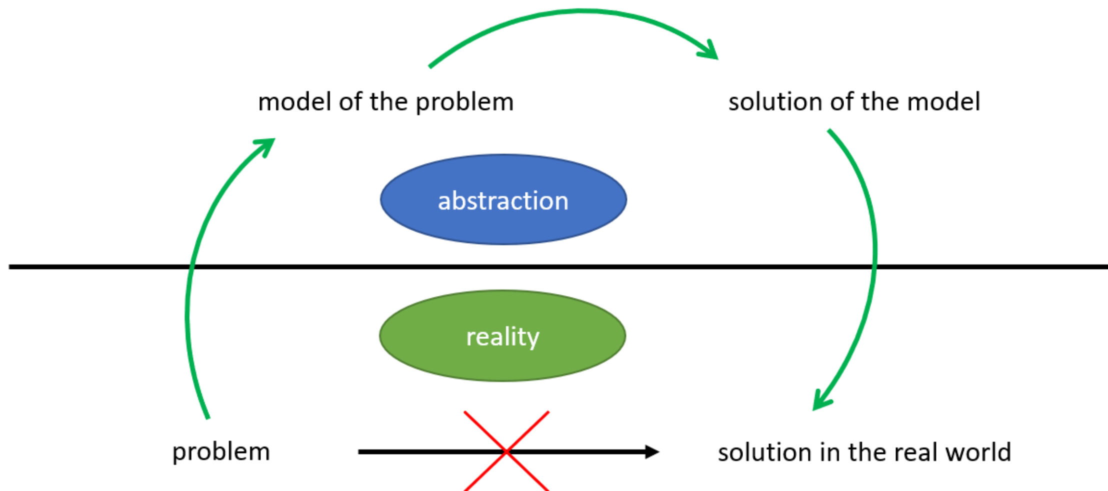

Figure 1.4: Relation between reality and abstraction for simulations

If we have a given problem and want to solve it, we proceed with describing the related model as an abstract presentation. Lets have an example:
We want to construct a building for some research companies. Before starting, you should consider how the construction must look like. How many floors, doors and rooms gets each company is also important to imagine? In this case we have to develop a scaled-down building of the original as model. A solution may be a miniature representation (physical or as software plan). With this final step we can solve our problem (building) in the real world with our given model as template. These processes of simulating and modeling work like a cycle in Figure 1.4.

1.2.3 Topology vs. Neighborhood

As already mentioned, a swarm technology consists of many individuals. Dependent on the simulation, we ether want to show an abstracted (lattice grid in Figure 1.5) or imitated (e.g. areas from the real world) environment. Determining the location and amount of individuals is also an important factor [35].

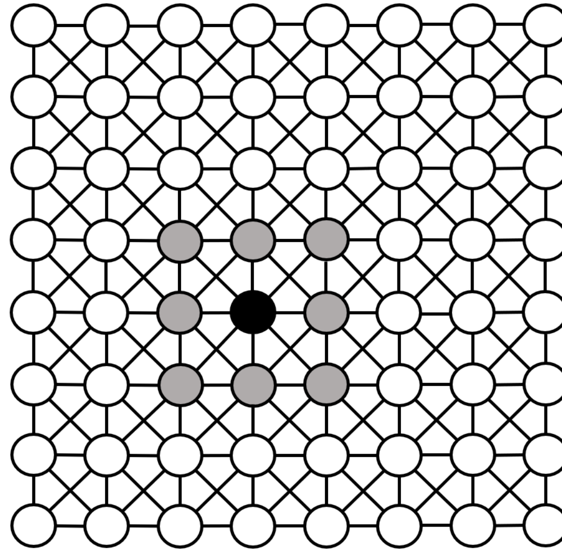

Figure 1.5: Structured population in a lattice grid [38]

This kind of neighborhood in Figure 1.5 is called by Moore. The entire lattice grid shows the intelligence of each individual identified by colors (seen in Chapter 3). Members inside the grid build AI for problem solutions. Each of them has an individual fitness.
The topology in a 3D space can be based on a toroid [1], as shown in Figure 1.6. There are different topologies of neighbors as ring and von Neumann [18, 25]. This master thesis uses however Moore's neighborhood. Other types are visible in Figure 1.7.

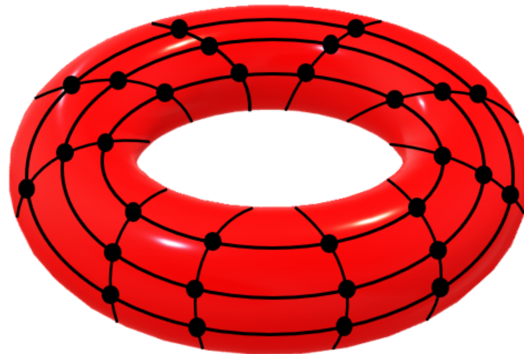

Figure 1.6: Toroid with grid

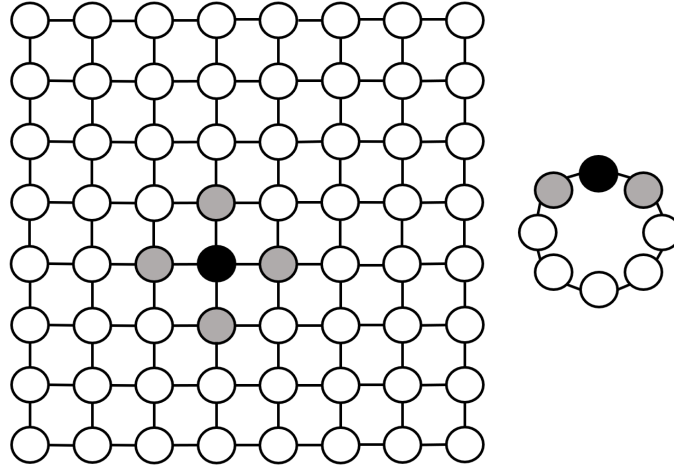

Figure 1.7: Von Neumann neighborhood and one-dimensional ring topology

A member in SSCEA2D considers only its closest neighbors. The environment where individuals work and communicate represents a lattice grid (Figure 1.5). Note: Individuals are in other articles, books or papers often called members, agents [39], cells or participants and has in this context the same meaning. In this work we use the term individual(s) exclusively. Dependent on the algorithm and rules, individuals work from each other to get an expected solution. There are four operations and they are listed as follows (and visible in Figure 1.8):

1. **Selection:** An individual chooses two potential parents (highest fitness) for reproduction (mating interactions [24]). The formed couple will be the parents for generating new offsprings in the next generation.

2. **Recombination:** Give birth to new children with a genotype mixed from the genotypes of the parents.

3. **Mutation:** Change the actual genotype of an individual randomly. In other words mutations are changing the representation.

4. **Replacement:** Replace the offspring to their new place if it has a higher fitness.

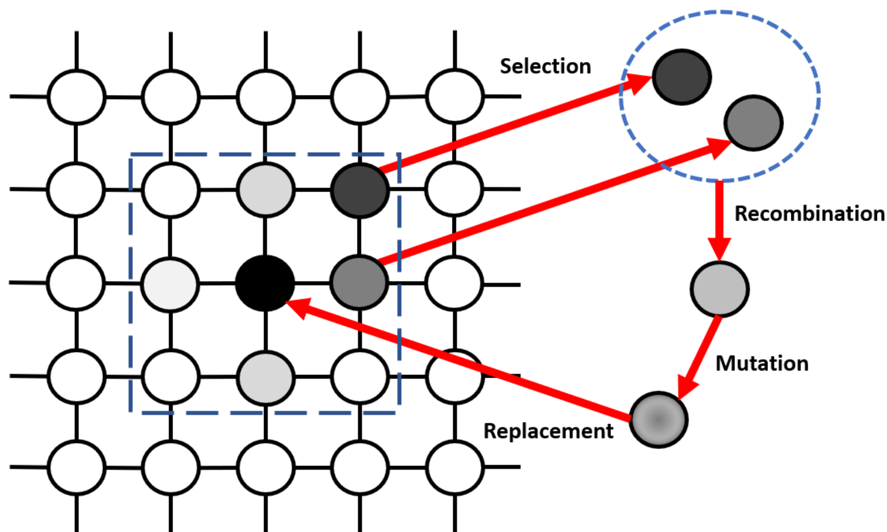

Figure 1.8: Evolution of each member as a life cycle [33]

Note, that these four operations do not match exactly with the rules defined by Charles Darwin. In Darwins theory is the selection for example defined as the decision, who survives for the next generation and not who are the parents for new individuals. When you consider again processes in biology we recognize that each evolution of any system needs a certain time. The group of all individuals forms a population with the following properties:

- Individuals are con-specifics
- Spatial context
- Reproduction ability

We call the process of optimization as evolution and means also the long-term alteration within generations (step of evolutionary process).

Algorithm 1.1 shows a simple pseudo code, how an EA works. Initialization steps prepare the population and candidates. While-loop contains the four evolutionary steps as discussed before and also the ranking of each candidate.

```
initialize population;
evaluate candidates;
while not (termination criterion) do
    parent selection;
    recombination;
    mutation;
    evaluate candidates;
    survivor selection;
end
```

Algorithm 1.1: Principle of EA [41]

Before an evolution starts, we generate (initialize) a population randomly and each individual gets a random fitness. After finishing these first steps, the evolution itself starts infinitely until no termination criterion occurs. Each individual chooses its potential parents. These parents make an offspring (recombination). Mutation is a random change of genes, in this case the fitness. Each offspring will be evaluated after recombination and mutation. The strongest individuals (higher fitness) will survive and the weak ones will not.
The evaluation of candidates is also called as ranking and refer to fitness. There are no common units for this parameter. Fitness can be calculated as an absolute (for example a float number between 0 and 1) or a relative value with a certain number of won games (for instance soccer match).
Algorithm 1.2 shows how an implemented cellular EA or SSCEA2D works. ρ_e, ρ_m and ρ_c denote the rate of elite individuals, probability of mutation and crossover. n_e represents the number of elite candidates and is dependent from ρ_e and neighborhood size [41].

```
X ← randomly generated population;
while not (termination criterion) do
    X_new ← empty population;
    foreach candidate x_i of X do
        run experiment for the neighborhood R_i of x_i;
        compute the fitness f(x) of R_i;
        descending sort of R_i based on f(x);
        if number of x_i in R_i ≤ n_e then
            add x_i to X_new;
        else
            generate random number r ∈ [0,1];
            if r < ρ_m then
                x_e ← randomly selected elite candidate;
                x'_e ← mutate x_e;
                add x'_e to X_new;
            else if r < ρ_m + ρ_c then
                x_e ← randomly selected elite candidate;
                c_i ← mate x_i and x_e;
                add c_i to X_new;
            else
                x_n ← randomly genertated individual;
                add x_n to X_new;
            end
        end
    end
    X ← X_new;
end
```

Algorithm 1.2: Pseudo-code for cellular EA [41]

Individuals mate with partners from geographically close region. In this case the closest neighborhood plays a role because we have a structure. This phenomena may occur in our real life if humans meet their potential partners for family planing (woman meets a man or vise versa). In other words, the distance between each individual does play a role for mating. The opposite of a structured neighborhood is called panmixia, where the distance is irrelevant for mate choice.

1.2.4 Fitness

Fitness is defined as the feedback of solutions and evaluates how close a solution is optimized for the given problem. The higher a value of fitness is, the better the solution. It gives us information, how well the optimizing problem can be solved. Fitness is expressed as absolute or relative value. In some problems it is sufficient to just compare candidate solutions and determine, which one is the best one (relative). Absolute values are expressed as for example in percentage or float numbers between 0.0 and 1.0. There are formulas in different contexts how to calculate the fitness on this sub chapter [9].

a	b	z
0	0	0
0	1	1
1	0	1
1	1	0

Table 1.1: Truth table of an exclusive or (XOR) gate (two inputs and one output)

The output is only logic 1 (true or high) if both inputs differ from each other (0,1 and 1,0), seen in Table 1.1. One input must not have the same state as the other one, otherwise the output state is 0 (false or low). The principle of XOR gate bases on addition of binary numbers.

$$F = -\frac{1}{4}\sum_{i=1}^{4}(o_i - z_i)^2 \tag{1.1}$$

The formula for fitness function is typically different for every problem. For the XOR gate we use the principle of (negative) mean square error [5] as depicted in Equation 1.1. o_i stands for the output of representation for the inputs and z_i for the expected output. Index i of the sum goes from 1 to 4 because we have four possible states with two inputs and one output.

$$f_i = 1 - \frac{1}{(N_c - 1)^2 \cdot w \cdot h}\sum_{x=1}^{w}\sum_{y=1}^{h}(c_x^y - r_x^y)^2 \tag{1.2}$$

SOS are also applied in image processing. With Cellular Automaton Morphogenesis (CAM) we can try to reconstruct a reference image. The target picture is copied and has the same amount of columns and rows of pixels as the template. Equation 1.2 above shows how to calculate the fitness by using CAM. N_C stands for the number of different colors from the reference image. w and h are width and height of the picture. x and y show the indexes. c_x^y represents the color output of

the associated coordinates and r_x^y the pixel reference (expected value).

$$F = \sum_{i=1}^{N} \frac{f_i}{2^{N-1}} \tag{1.3}$$

Equation 1.3 describes the fitness for a non-oscillating and stable solution. f_i is the calculated fitness from the Equation 1.2 before and N the number of iterations.

1.2.5 Diversity

Diversity has different meanings in technical and non-technical sciences. This chapter discusses the behavioral diversity in swarm technologies [3]. In member based simulations exist during the run more or less different solutions. Members work in a collaborated working manner. Through this group dynamic may exists more or less different solutions. Diversity is a measure defined by the mean distance of genotypes from a population of individuals. A different measure for diversity is based on the entropy from Shannon in information theory (see Equation 1.4 below). Entropy has its origins from the second law of thermodynamics [17].
High diversities means a group of individuals have more different solutions. That means, we have inside the grid more smaller groups of individuals and they may not share their information to the other groups. Drawback is here, that the individual solutions might be so different that they cannot learn from the other ones. Participants work in this situation more isolated.
Lower diversities explain exactly the opposite of high: Less different solutions and more teamwork with the individuals inside a lattice grid is then the case. In this situation we suffer from a low variety of less solutions for problem solving.
It follows, that neither high nor low diversity show an efficient work behavior. If we should consider an average diversity, then we can use the advantages of both properties. Determining of diversities will be simulated and shown in Chapter 4.

$$H(X) = \sum_{i=1}^{M} p_i \cdot log_2(p_i) \tag{1.4}$$

$H(X)$ in Equation 1.4 stands for the entropy, p_i is the probability of index i and M shows the amount of indexes. log_2 means the dual logarithm and can be calculated as follows:

$$log_2(x) = \frac{ln(x)}{ln(2)} = \frac{log_{10}(x)}{log_{10}(2)} \tag{1.5}$$

Hint: You can use the dual logarithm with naturalis or generalis (Equation 1.5) divided by the base of 2. x may be any number for calculating. The dual logarithm is very often used because of binary operations in information technologies.

1.2.6 FREVO

FREVO is a simulation tool for evolving algorithms in technical, physical and biological domains [37]. It represents an open-source framework and is developed for evolutionary design or optimization tasks (written in Java [15]). FREVO has the major feature of separated key building blocks. They are called as the problem definition, solution representation and optimization method. By separating these blocks the user can easily change and swap different configurations.

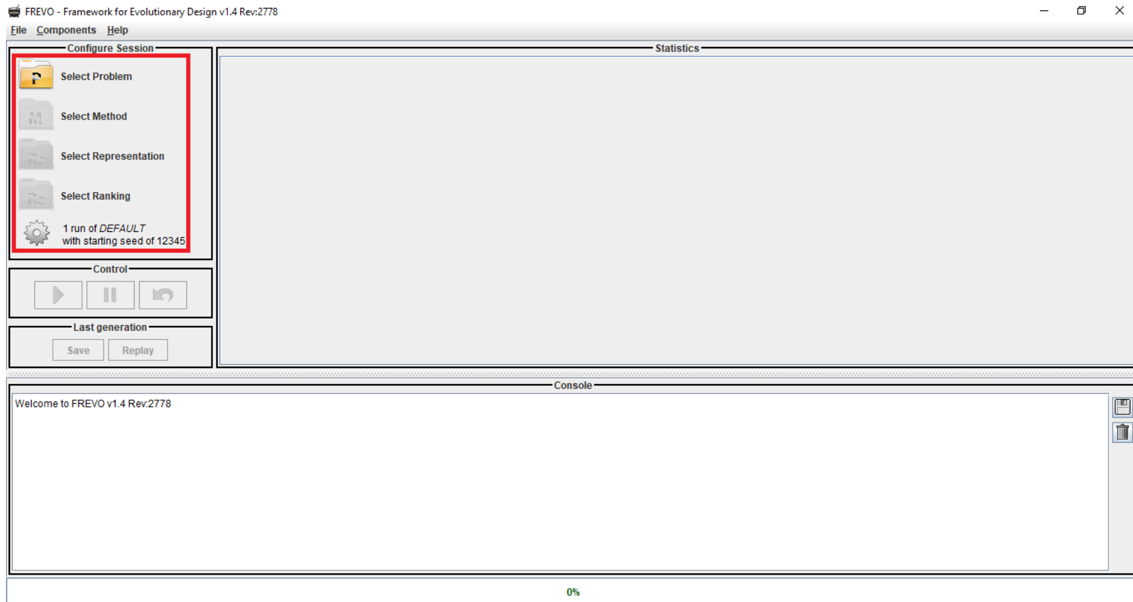

Figure 1.9: FREVO user interface (version 1.4.1)

If you open FREVO, you have to set in the configure session (red marked in Figure 1.9) five different options and they are as follows:

1. **Problem:** Shows the context of individuals for the evaluations.
2. **Method:** How to structure a solution. Method contains some optimization algorithms as the SSCEA2D.
3. **Representation:** How is the intelligence of each individual built? For intelligence are Artificial Neural Networks (ANNs) used.
4. **Ranking:** Defines how the evaluation of all individuals is done. This module creates a ranking based on fitness (feedback of the individual).
5. **Cog wheel:** Set the number of simulations (runs), the initial state (seed) and the name of experiment.

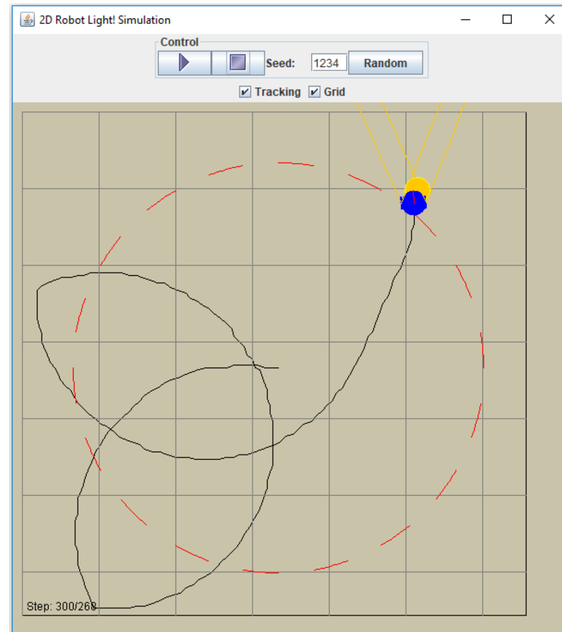

Figure 1.10: Robot searches a light source in FREVO

FREVO includes several examples for autonomous robotics. A simple problem is to make a robot find the light source inside an area using its local sensors, which is shown in Figure 1.10. Dependent on the best fitness of the simulation, robot needs more or less steps and time to find the target. Starting position to search the light depends on the seed and starting direction. Steps will be counted and the robot stops, if the yellow ball (light bulb) is reached by the robot.

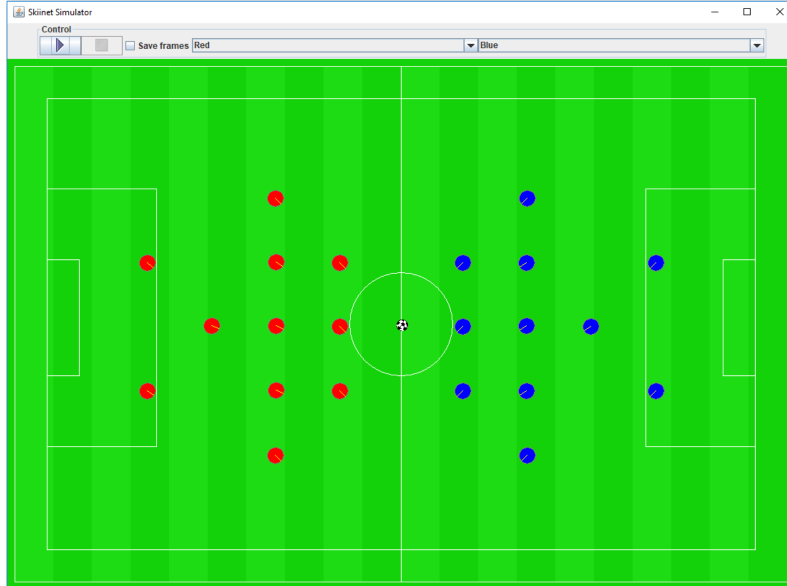

Figure 1.11: Soccer game with AI in FREVO

FREVO also offers more complex problems. A particular one is the evolution of autonomous soccer players [16]. The fitness depends on various indicators (field coverage, ball possession and scored goals), which defines the better team out of two teams playing against each other. An example is visible in Figure 1.11 with 11 players per team.

1.2.7 Neural Networks

Many applications use ANNs as evolutionary controllers [13, 14]. Our brain consists of neurons as well and much of them are responsible for recognizing letters, symbols, shapes, objects or people. ANNs are for instance used in memory network applications (convert images in bit-maps [42]). Neurons are complex cells and react on electrochemically signals [31].
A typical ANN neuron works like a comparator, which produces an output if a cumulative effect of input impulses exceed a threshold. Each input branch consists of an impulse x_i and a accordingly weighting w_i (a kind of filter for linking the inputs with neurons). These weightings gain (excitatory, positive value) or attenuate (inhibitory, negative value). Weights are typically represented with real values.

$$net = \sum_{i} x_i \cdot w_i + b_i \tag{1.6}$$

net is the sum of all inputs with weightings and i the index (Equation 1.6). f represents the activation function. Dependent on threshold for activation, a so called

bias b_i will be used.

$$y = f(net) \tag{1.7}$$

Dependent on net, y in Equation 1.7 may have different mathematical behaviors as linear, step, tangent hyperbolic or Sigmoid. All of these functions have a typical range for activation. Because of this reason we need a bias for meaningful activation as well.

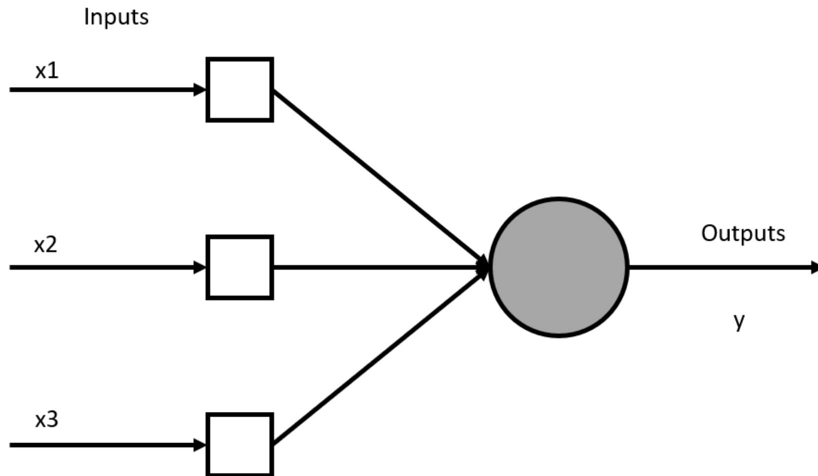

Figure 1.12: Simple ANN [31]

Figure 1.12 shows a simplified representation of an ANN. Example: Inputs on the left side are fragments of symbols (i.e. numbers). Gray circle merges these fragments and gives an output dependent which symbol we notice. ANNs in practice may have course many more inputs and also (hidden) layers between input and output. A hard-to-read handwriting complicates our perception and neurons, so more layers may be necessary.

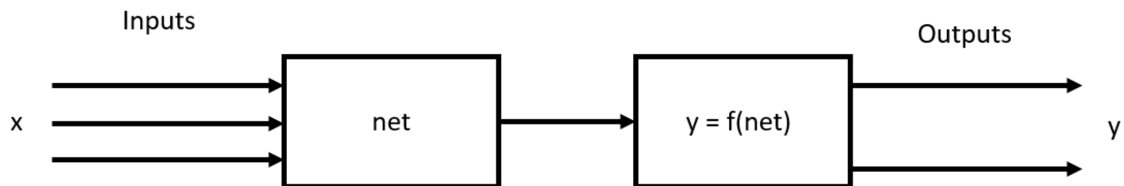

Figure 1.13: Block diagram for ANNs [31]

ANNs can be displayed with block diagrams. Output y is dependent on the sum of weighted inputs net (Figure 1.13).

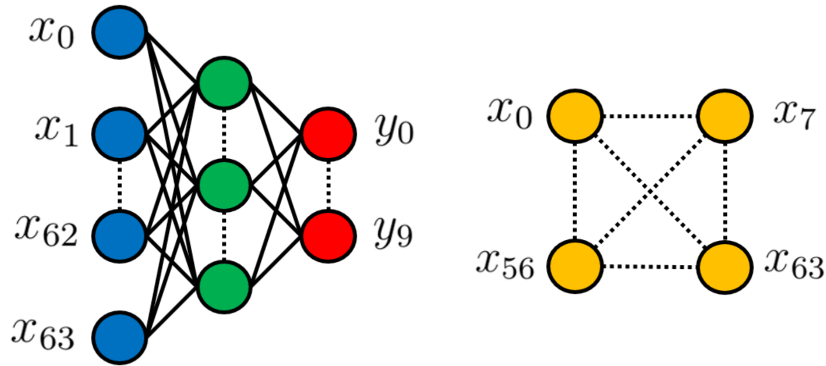

Figure 1.14: TLNN vs. FMN

In this thesis we are dealing with two different types of neural networks and they are called Fully Meshed Net (FMN) and Three Layered Neural Network (TLNN). Both of them are available as representation in FREVO and important for evaluating the SSCEA2D. TLNN consists of a middle layer between inputs and outputs as processing stage. In contrast, FMN has no defined hierarchy and each neuron is directly with each other connected (Figure 1.14). The results that can be achieved through this will be revealed in the simulations.
Let us consider an example of numbers. We have an coordinate system (Figure 1.15) and each coordinate has an own designation. In Figure 1.16 we recognize different symbols with this kind of coordination system. These symbols are written in black color and the rest of area is white. The activation value for black is in this case 0.00 and for white 1.00. Values between them are called gray-scales, but for an easier illustration we use only maximum and minimum activation values. All numbers are located in a 8x8 net (64 possible positions or coordinates for depiction). Notice, that these nets have nothing to do with cellular evolutionary lattice grids, rather with the neuronal recognition.

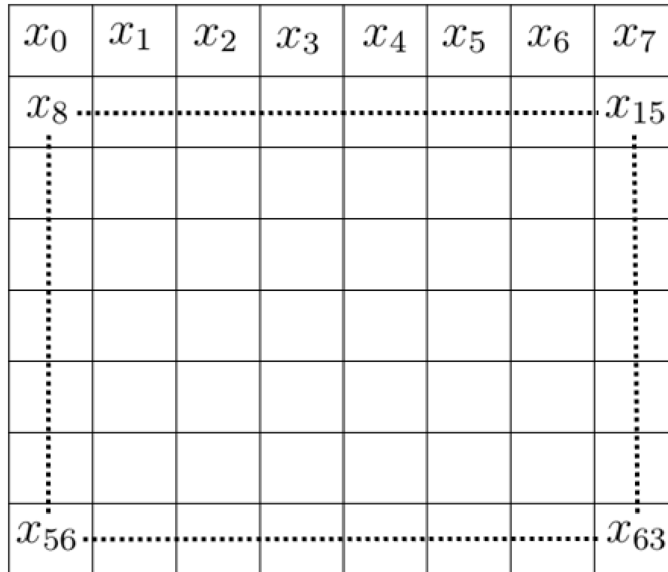

Figure 1.15: Coordinates for mapping each pixel

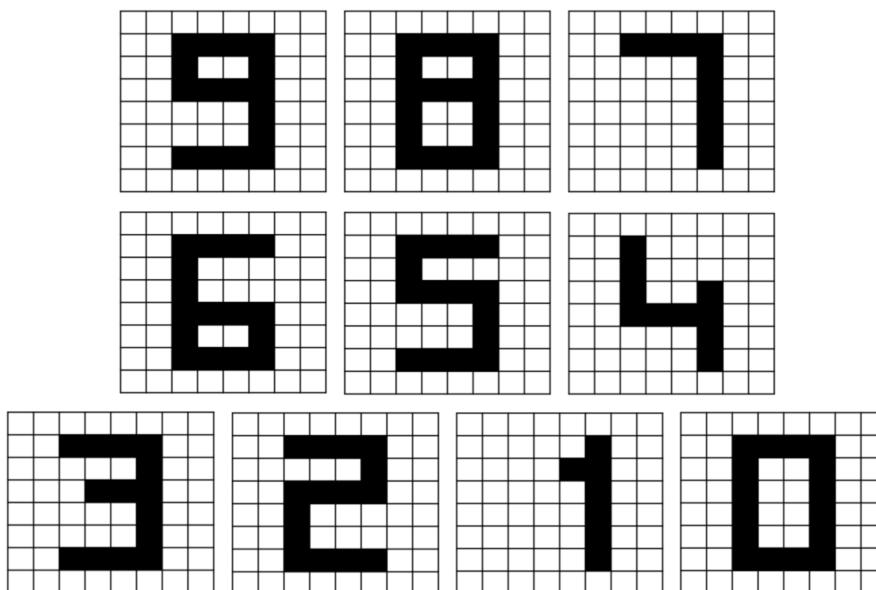

Figure 1.16: Numbers between 0 and 9

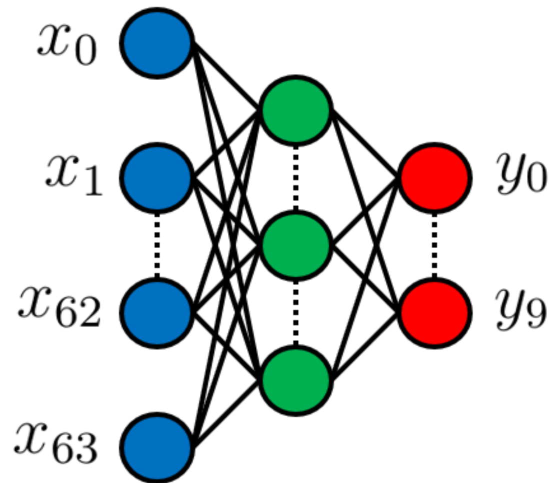

Figure 1.17: 8x8 net for recognizing numbers with TLNN

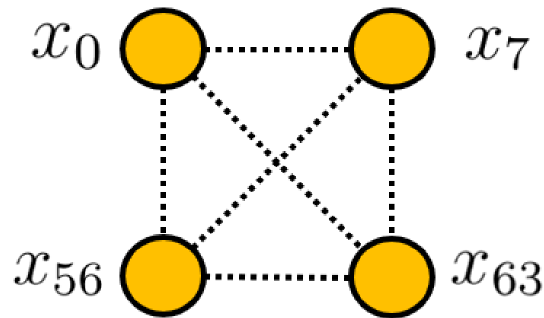

Figure 1.18: Symbols with FMN

Some symbols consist of mostly equal parts as the numbers 8 and 9. Figure 1.17 and Figure 1.18 show how our brain creates connections between similar symbols for recognition. In FMN are input and output nodes defined by a-priori. With these both representations we describe two different variants of AI for each member in SSCEA2D. The behavior in certain situations inside the lattice grid is decisively determined by the intelligence of individuals.

Chapter 2

Problem Description

This chapter explains which practical steps were to fulfill and gives deeper information about the tasks. Solutions for the tasks were realized with the already existing simulation tool FREVO. FREVO was prepared for an update-version with extended features for SSCEA2D. Engineers and scientists will get with this improved software more possibilities in research and development regarding SOSs.

2.1 Purpose

SSCEA2D consists of individuals, which are located in a two-dimensional grid. Each ot them considers only its closest neighbors and develop dependent on their fitness new solutions to apply in SOSs. The previous chapter explains the properties of SSCEA2D more in detail. This master thesis researches with this existing algorithm to improve its behavior and capabilities in simulation environments. Can we get more efficient solutions or improvements?

2.2 Research Questions

To get a deeper understanding what the expected outcome of this work is, will be explained by the research questions. They are as follows:

1. **How do fitness and diversity change by rectangular grid?**
   Fitness and diversity have a dependence on the population size. How are these affected for different width and height of the grid in a non-panmixia evolution?

2. **Do integrated obstacles influence the evolution in the grid?**
   Obstacles mean, that some grid cells are non-functional. This way, 100 individuals could be fit into a larger 12x12 grid instead of a 10x10 grid. Can we achieve a higher fitness by adding obstacles?

3. **Which conditions yield the highest fitness and/or average diversity?**
   Which solutions are possible with adjustable grid width/height and obstacles? Which setup yields the best results and why?

4. **Does the new developed system work more efficiently compared to the existing algorithm?**
   The grid in SSCEA2D was earlier only a square grid in FREVO. Does the new approach provide better performance in comparison to the existing implementations?

5. **How is the resulting distributed?**
   If we have some results of different simulations and conditions, we want to analyze their fitness and diversity by different seeds. There are many types of distributions (Gauss, uniformly, gamma, exponential, Weibull, chi-squared, etc....) and the questions is, which distribution matches with the simulation results.

2.3 Tools and Methods

FREVO was implemented in the languages JAVA and XML. For developing it further, the development environment Eclipse was used. The entire framework uses object-oriented-programming. For answering the research questions, the predefined classes in the Java code were modified and extended with new methods and properties. To determine the distribution of different simulations, a Python program was used to check the distribution of data. This code compares the simulation results and estimates the probability, how the data matches with the distributions.
Many simulations in FREVO were necessary requiring a server to run simulations offline with the duration of several days. To evaluate the results statistically, Microsoft Excel was used.

Chapter 3

Implementation

All solutions regarding the project are listed and described in the following chapter. Code snippets and figures show, how the FREVO update is realized.

3.1 Rectangular Grid

The user can now set the width and height of the grid to an arbitrary number at will. Figure 3.1 and Figure 3.2 show some examples of rectangular lattice grids. Inside are the individuals for evolutionary simulations located (squares). Each of them has a color and represents the associated fitness. Green means good, orange/yellow stands for average and red individual consists of a poor fitness. White squares are no individuals but can be occupied in the next generations. After each generation, the grid gets an update and the members inside change their color because of their neighborhood activities (selection, recombination, mutation and replacement).

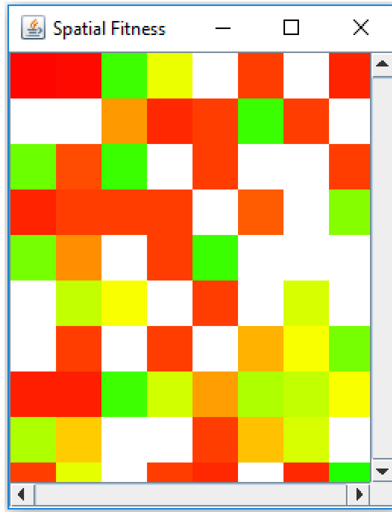

Figure 3.1: Grid with 8 width and 10 height

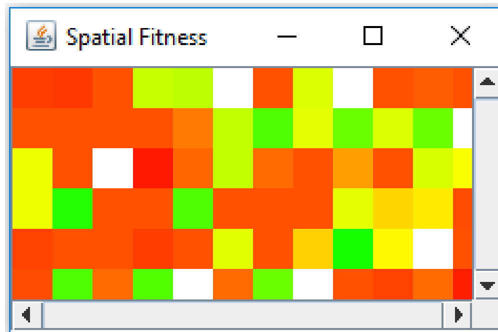

Figure 3.2: Grid with 12 width and 6 height

3.2 Obstacles

Next feature is the possibility to add obstacles in the lattice grid. There exist three predefined patterns (setup for at least 10x10 grid) and the fourth one distributes gray obstacles randomly as well (grid size not critical).

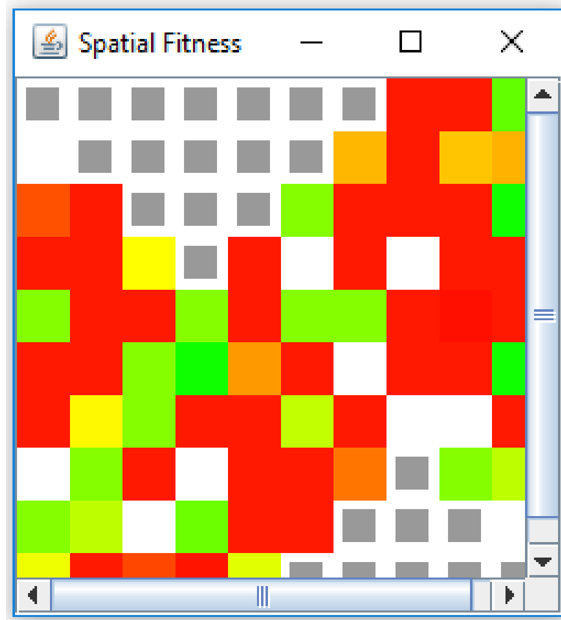

Figure 3.3: Pattern 1

Figure 3.3 shows the shape of two pyramids and in total are 25 obstacles present. So the size of this example grid consists of 100 possible individuals minus 25 obstacles. It follows, this is a rectangular lattice grid with 75 valid individuals.

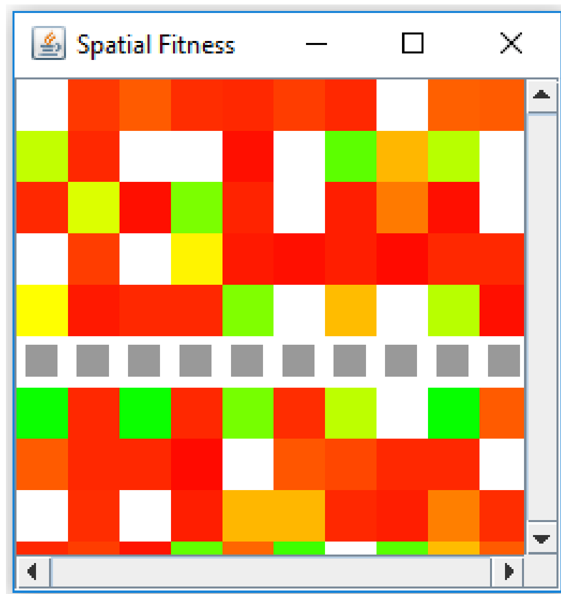

Figure 3.4: Pattern 2

Next experiment shows how to separate the entire grid into two sub-grids with a

horizontal line (Figure 3.4). With this pattern we have the ability to set two isolated populations as well.

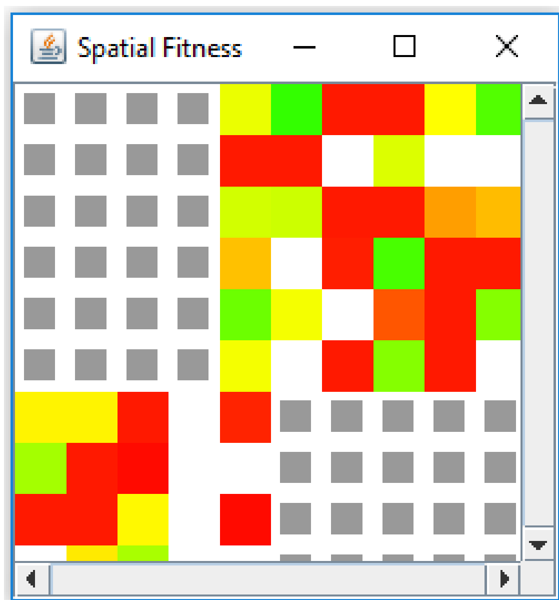

Figure 3.5: Pattern 3

The last predefined pattern shows two added rectangular shapes, seen in Figure 3.5. All patterns will be used for simulating comparisons.

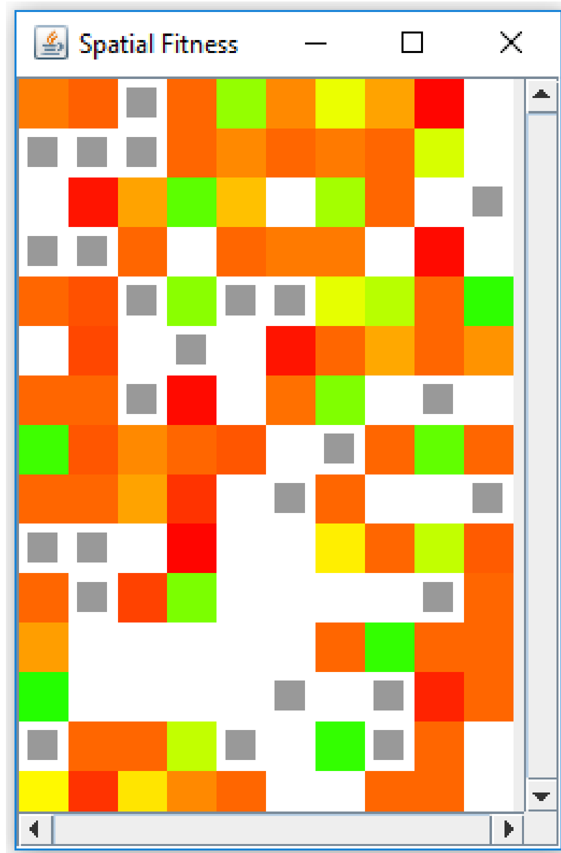

Figure 3.6: Pattern 4 and 10x15 grid

In Figure 3.6 you can see a random distribution of gray obstacles. With FREVO it is possible configure a set of runs with different obstacle shape (dependent on the seeds). Thus, we can determine fitness, diversity and distribution of each setup.

3.3 Source code

The next sections show the software implementation in different programming languages. Extensive codes to look up are in the chapter Appendices. Each code has numerated lines for facilitating the explanations.

3.3.1 Setting obstacles, patterns, parameters and plotting diversity

The following implementations for modifying the SSCEA2D were written in Java.

- Definition of parameters for SSCEA2D: Input of obstacle patterns, height/width for lattice grid, etc. in Listing A.1 from line 1-154. Method in line 162 shows a random number generator for obstacle distribution (based on the seed).
- Programmed neighborhoods (grid and random) in Listing A.2 from line 305-326 and 331-347.
- Implementation of obstacles (patterns and randomly distribution) in Listing A.3 from line 101-191.
- Plot diversity and fitness Listing A.3 (lines 636-640)

3.3.2 Reading text files for generating boxplots

Python programs are responsible for:

- reading text files (fitness and diversity results) in Listing B.1 from line 3-13.
- plot boxplots of each simulation run (Listing B.1, line 18).

3.3.3 New parameters for FREVO interface

Existing code in Listing C.1 was modified, see lines 10-18 and 24-28. XML code has now the following features:

- to insert width and height.
- additional information about the modified algorithm
- to enter obstacle patterns

Chapter 4

Simulation Comparisons

4.1 Preparations

All simulations were executed on an external server ("feynman") of the University. For data management, WinSCP was used and PuTTY is a console for starting and modifying setups.

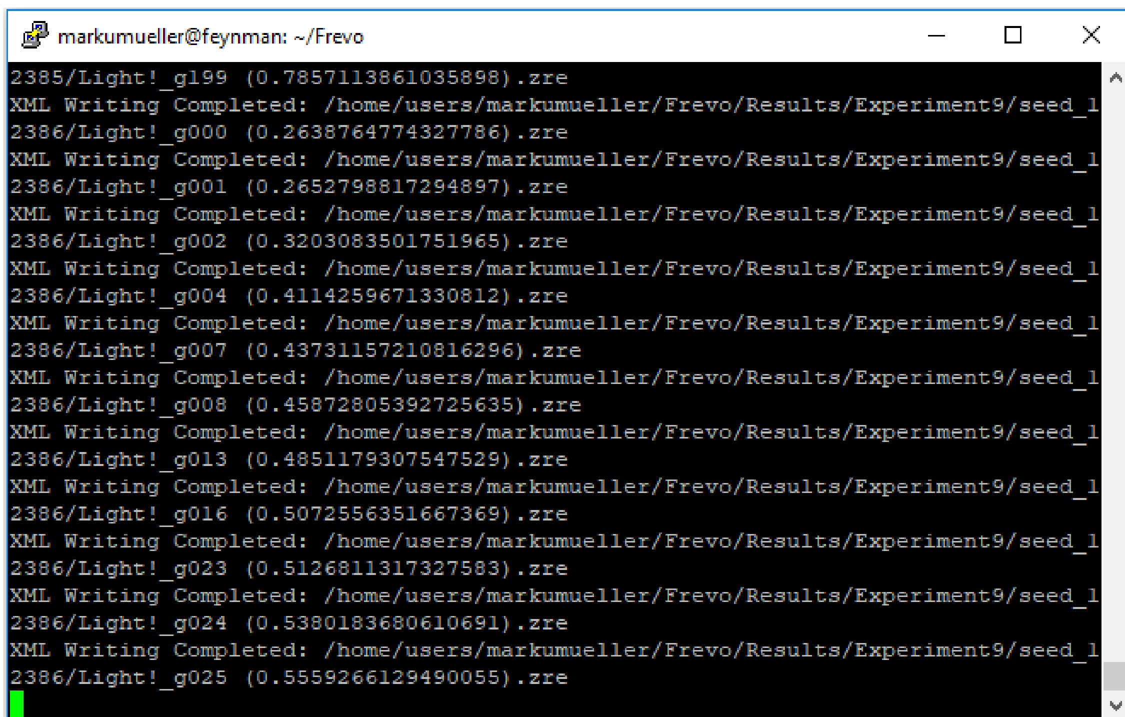

Figure 4.1: Running simulation on simulation server

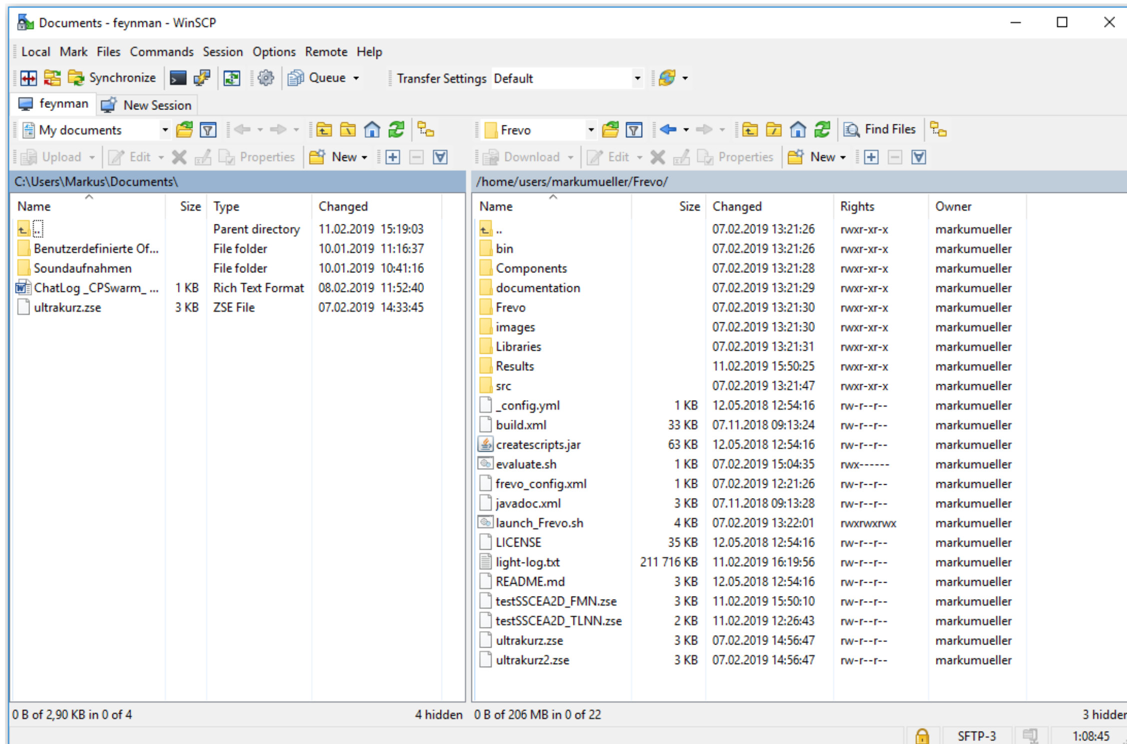

Figure 4.2: Data management on WinSCP

In Figure 4.1 is a running process of simulation via PuTTY. The appearing lines are results (.zre files) and saved in the directory for results. You need for starting a shell-file (.sh) and a session-file (.zse). The console commands base on the scripting language "Bash". A graphical overview offers the WinSCP window in Figure 4.2, where you can manage your acquired data on the server.
The implementations have to be examined after finishing the simulation runs. For this reason, we used different sizes of the lattice grid (square and rectangular with-/without obstacles) to determine the most efficient solution. The algorithm further supports two kinds of neighborhoods: grid and random.
Grid neighborhood means each individual considers its adjacent neighbors. Purpose of this comparison is not only to compare obstacle or rectangular modes. Neighborhood and representations may be also considered to get conclusive data for research. Random neighborhoods are the opposite of grid and individuals get their neighbors randomly. In this mode exists no structure by rows/columns and thus there is no structured way. Additionally we want to show some differences with our two representations in FREVO: FMN and TLNN. Table 4.1 and Table 4.2 give an overview of executed simulation runs. The modified SSCEA2D was tested here with the Light! and Simplified Soccer Game problems in FREVO because the original SSCEA2D was compatible with those problems. FMN and TLNN were chosen because they were used in past researching tasks and to get conclusive comparisons.

ID	Neighborhood	Obstacles	Height	Width	Population size	Representation
0	Grid	20	10	12	100	FMN
1	Grid	0	10	10	100	FMN
2	Grid	Pattern 1	10	10	75	FMN
3	Grid	Pattern 2	10	10	90	FMN
4	Grid	Pattern 3	10	10	56	FMN
5	Grid	0	8	8	64	FMN
6	Grid	11	5	15	64	FMN
7	Grid	11	15	5	64	FMN
8	Grid	0	20	5	100	FMN
9	Grid	0	5	20	100	FMN
10	Grid	150	10	20	50	FMN
11	Grid	150	20	10	50	FMN
12	Grid	50	10	10	50	FMN
13	Random	20	10	12	100	FMN
14	Random	0	10	10	100	FMN
15	Random	Pattern 1	10	10	75	FMN
16	Random	Pattern 2	10	10	90	FMN
17	Random	Pattern 3	10	10	56	FMN
18	Random	0	8	8	64	FMN
19	Random	11	5	15	64	FMN
20	Random	11	15	5	64	FMN
21	Random	0	20	5	100	FMN
22	Random	0	5	20	100	FMN
23	Random	150	10	20	50	FMN
24	Random	150	20	10	50	FMN
25	Random	50	10	10	50	FMN
26	Grid	20	10	12	100	TLNN
27	Grid	0	10	10	100	TLNN
28	Grid	Pattern 1	10	10	75	TLNN
29	Grid	Pattern 2	10	10	90	TLNN
30	Grid	Pattern 3	10	10	56	TLNN
31	Grid	0	8	8	64	TLNN
32	Grid	11	5	15	64	TLNN
33	Grid	11	15	5	64	TLNN
34	Grid	0	20	5	100	TLNN
35	Grid	0	5	20	100	TLNN
36	Grid	150	10	20	50	TLNN
37	Grid	150	20	10	50	TLNN
38	Grid	50	10	10	50	TLNN
39	Random	20	10	12	100	TLNN
40	Random	0	10	10	100	TLNN
41	Random	Pattern 1	10	10	75	TLNN
42	Random	Pattern 2	10	10	90	TLNN
43	Random	Pattern 3	10	10	56	TLNN
44	Random	0	8	8	64	TLNN
45	Random	11	5	15	64	TLNN
46	Random	11	15	5	64	TLNN
47	Random	0	20	5	100	TLNN
48	Random	0	5	20	100	TLNN
49	Random	150	10	20	50	TLNN
50	Random	150	20	10	50	TLNN
51	Random	50	10	10	50	TLNN

Table 4.1: Overview of simulations with Light!

Each simulation starts with the random initial state ("seed") 12345 and runs 100 times until 12444 inclusively (for the Light! problem). From each ID, the highest fitness will be selected, so all 52 maximums will be compared. Not only the minimum and maximum play a role, rather the scattering of each simulation ID by using boxplots will be shown on the next pages. For the diversities of each simulation (with the 100 seeds), mean values are calculated and compared, respectively.

ID	Neighborhood	Obstacles	Height	Width	Population size
1A	Grid	0	10	10	100
2A	Grid	100	20	10	100
3A	Grid	200	10	30	100
4A	Grid	25	10	10	75
5A	Random	0	10	10	100
6A	Random	100	20	10	100
7A	Random	200	10	30	100
8A	Random	25	10	10	75
1B	Grid	0	10	10	100
2B	Grid	100	20	10	100
3B	Grid	200	10	30	100
4B	Grid	25	10	10	75
5B	Random	0	10	10	100
6B	Random	100	20	10	100
7B	Random	200	10	30	100
8B	Random	25	10	10	75

Table 4.2: Overview of simulations with Simplified Soccer

We used for the soccer problem different setups and runs. Instead of 100 seeds here are two seeds used with 1000 generations for each run. TLNN is a memory-less representation and has the drawback, that the neural network has no knowledge about the past states, which significantly affects the soccer players rightly. Soccer players with a TLNN intelligence run only by each other and do not consider to kick the ball into the goal. In contrast, FMNs can keep a state via recurrent feedback connections and therefore are suitable for this soccer problem. Each run in Table 4.2 will be run twice because of two different seeds (**12345** for **A** and **11111** for **B**). With both seeds we want to prove, is this setting suitable in general or not. Runs with population size 75 need more than 1000 generations (exactly $1000*(1/0.75) = 1333$) compared to size 100 because a smaller grid needs more evolution time. So we can ensure, that the matches run with fair conditions.
All settings in FREVO are visible in Table 4.3, Table 4.4, Table 4.5, Table 4.6, and Table 4.7.

Keys	**Values**
evalnumber	20
fitnesscalculation	Using Time and Distance
gridcellsize	30.0f
simulationtime	30000

Table 4.3: Settings for Light!

Keys	**Values**
apply stamina model	FALSE
ball distance weight	1000
ball goal weight	100000
controller model	NEARESTINFOPLAYER
evaluation time	60000
isCartesian interpretation	TRUE
kick weight	20000
max kicks	10
playersPerTeam	10
position weight	1
score weight	4000000

Table 4.4: Settings for Simplified Robot Soccer

Keys	**Values**
generations	200
mutationprobability	1
mutationseverity	0.3f
neighbourhoodmode	**1 or 2**
obstacle-pattern	**1, 2, 3 or 4**
percentelite	11
percentmutateelite	59
percentxoverelite	30
populationsize height	**on request**
populationsize width	**on request**
random obstacles	**on request**
saveinterval	0

Table 4.5: Settings for CEA2D

Keys	**Values**
activationFunction	LINEAR
bias range	2f
hiddenNodes	2
iterations	2
mutation rate	0.2f
random bias range	0.2f
random source	false
variable mutation rate	false
weight range	2f

Table 4.6: Settings for FMN

Keys	**Values**
bias range	2f
hiddenNodes	2
stepNumber	2
weight range	2f

Table 4.7: Settings for TLNN

4.2 Fitness and Diversity Results

Statistical results are in the next sub-chapters after data acquisition be shown. Because of the high amount of results (100 fitness and diversity values of each run), we consider only values from the last generation of each simulation run. A way to compare the scattering, maximum, minimum, etc. is to use boxplots.

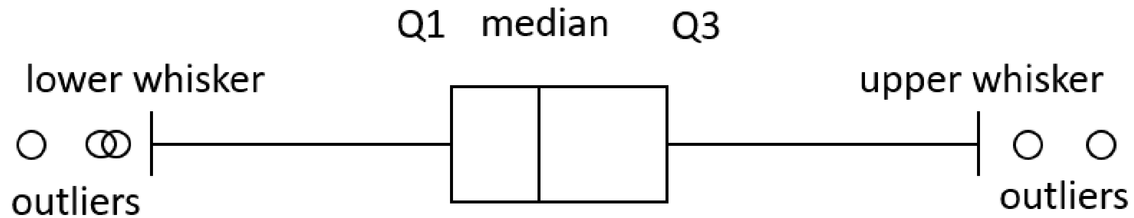

Figure 4.3: Parameters of Boxplot

Figure 4.3 shows the principle of boxplots and which parameters are interesting in statistics. Advantage of this method is to get a clear overview of the behavior from results. Boxplots are divided in four so called quartiles and show the range of each 25%. They have the following characteristics for analysis:

- Lower whisker: Is defined as the lowest data value that is still within $(Q1 - (Q3 - Q1)) \cdot 1.5$.
- Q1: This is the first quartile and represents the first 25% of data as well.
- Median: Mean value of data. The median is the end point of the second quartile and starting point of the third quartile.
- Q3: It is the end point of the third quartile and starting point of the fourth quartile.
- Upper whisker: This is the highest data point that is still within $(Q3 + (Q3 - Q1)) \cdot 1.5$.
- Outliers: Values under the lower and over the upper whisker are considered to be outliers.

4.2.1 Light!

Simulation ID	Maximum Fitness	Diversity
0	0.8465	17.7111
1	0.8805	19.6437
2	0.8366	22.4593
3	0.8731	24.0929
4	0.8277	21.5454
5	0.8758	14.1775
6	0.9098	16.2253
7	0.8483	15.3739
8	0.854	28.3004
9	0.8577	30.5439
10	0.7398	115.8686
11	0.7356	101.1211
12	0.7807	29.2327
13	0.8913	14.1262
14	0.9017	16.0454
15	0.8993	12.8149
16	0.869	11.8084
17	0.8251	14.664
18	0.8759	13.2979
19	0.8539	10.2891
20	0.8925	10.2772
21	0.9017	16.0454
22	0.9017	16.0454
23	0.8521	11.9458
24	0.8748	14.4284
25	0.8701	15.2629
26	0.8681	0.4801
27	0.8697	0.4023
28	0.8796	0.4393
29	0.8702	0.5095
30	0.854	0.4974
31	0.8568	0.2835
32	0.8648	0.5546
33	0.8653	0.4123
34	0.8821	0.5286
35	0.8674	0.491
36	0.8196	1.5196
37	0.8419	2.1099
38	0.8765	0.8522
39	0.8536	0.5543
40	0.8541	0.3891
41	0.8701	0.2765
42	0.8743	0.331
43	0.8611	0.4467
44	0.8773	0.3422
45	0.868	0.2938
46	0.8604	0.3627
47	0.8541	0.3891
48	0.8541	0.3891
49	0.8748	0.285
50	0.8565	0.2685
51	0.8627	0.3061

Table 4.8: Overview of maximum fitness and average diversity for Light!

In Table 4.8 are the maximum fitness of each run with the referred diversity. Note that theses values do not give information about the scattering of each simulation. In every run, one seed generates the highest fitness and the results are in the table above. Green marked cells in Table 4.8 show the closest values of average diversity and the maximum fitness of all 52 experiments as well.
There are boxplots for the representations FMN and TLNN in the next illustrations (Figure 4.4,Figure 4.5, Figure 4.6, Figure 4.7). Simulation IDs 10, 11, 36 and 37 contains 75% of obstacles and the scattering is much wider than the other experiments with less percentage of obstacles. Additionally, the user can increase with this feature the diversity as well. These simulations show that the population size itself says nothing about fitness and scattering, but the number of obstacles inside the grid. This feature occurs already by 50% of obstacles (seen in simulation IDs 12, 25, 38 and 51). At less than 50% we cannot detect any significant changes. So with obstacles we can summarized increase the scattering of fitness and diversity.

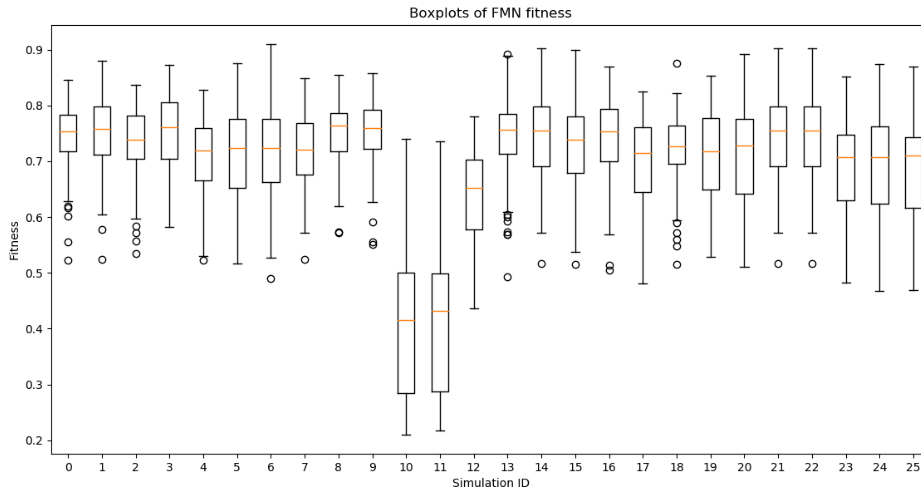

Figure 4.4: Fitness results (FMN) of 0-25 with Light!

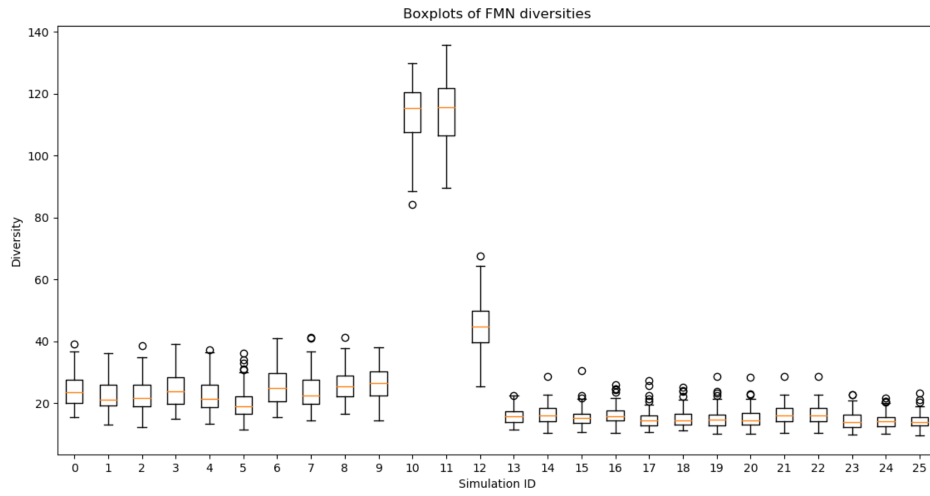

Figure 4.5: Diversities of Fully Meshed Net (run 0-25) with Light!

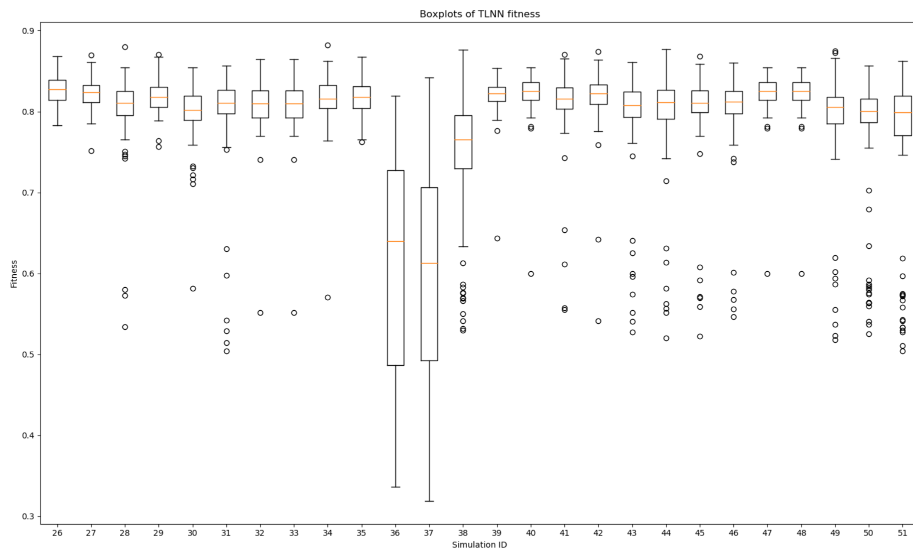

Figure 4.6: Fitness results (TLNN) of 26-51 with Light!

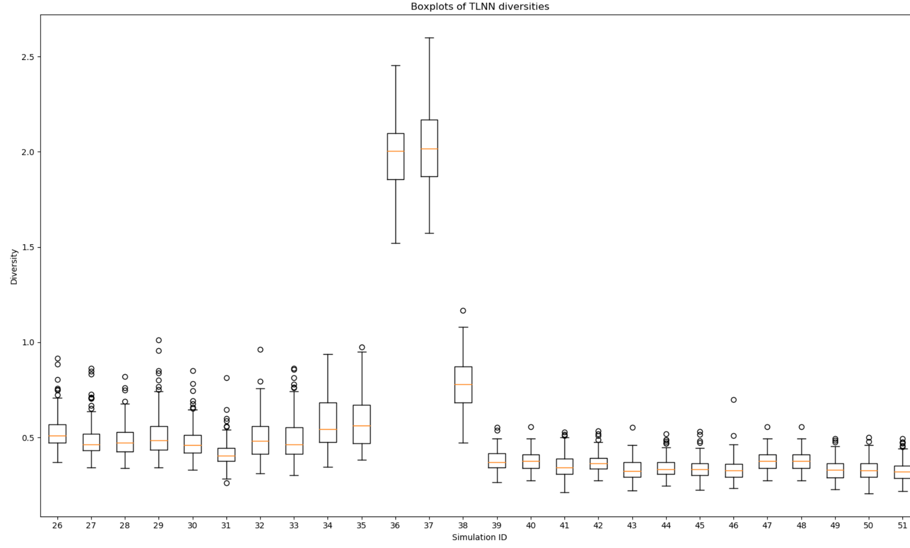

Figure 4.7: Diversities of Three Layered Neural Network (run 26-51) with Light!

To determine the average diversity of these versatile experiments, the mean is used by summing up all FMN and TLNN values divided by the amount of samples (in this case 26 for each representation) separately. The reason is, that the diversity in FMN is differently scaled compared to TLNN.

$$\overline{x} = \frac{1}{N} \sum_{i=0}^{N-1} x_i \tag{4.1}$$

N shows the number of simulations (26), i the index and $\overline{x}$ the mean value in Equation Equation 4.1.

$$\overline{x}_{FMN} = 24.3594923$$
$$\overline{x}_{TLNN} = 0.52747692$$

In the next page are the maximum fitness referred to the population size once more compared in detail.

Shape	FMN grid	FMN rand	TLNN grid	TLNN rand
10x12 (20 obstacles)	0.8465	0.8913	0.8681	0.8536
10x10	0.8805	0.9017	0.8697	0.8541
20x5	0.854	0.9017	0.8821	0.8541
5x20	0.8577	0.9017	0.8674	0.8541

Table 4.9: Comparison with population size 100

Table 4.9 compares the population size 100. We can show that the fitness is equal with random neighborhood by same size. Height and width do not play a role because the only requirement is the same area of 100 (10x10, 20x5 or 5x20), but no obstacles integrated.

Shape	FMN grid	FMN rand	TLNN grid	TLNN rand
8x8	0.8758	0.8759	0.8568	0.8773
5x15 (11 obstacles)	0.9098	0.8539	0.8648	0.8773
15x5 (11 obstacles)	0.8483	0.8925	0.8653	0.868

Table 4.10: Comparison with population size 64

A smaller population size gives dependent on the seed and obstacle position a comparable good fitness (seen in Table 4.10).

Shape	FMN grid	FMN rand	TLNN grid	TLNN rand
Pattern 1	0.8366	0.8993	0.8796	0.8701
Pattern 2	0.8731	0.869	0.8702	0.8743
Pattern 3	0.8277	0.8251	0.854	0.8611

Table 4.11: Comparison of predefined shapes

Also we checked in Table 4.11 different patterns with a 10x10 grid. Here gets the first pattern with random neighborhood and FMN the maximum fitness again. Reason of that may be, that each individual does not consider its closest partners, but the get them randomly. So obstacles influence the evolution less than in other conditions.

Shape	FMN grid	FMN rand	TLNN grid	TLNN rand
20x10 (150 obstacles)	0.7398	0.8521	0.8196	0.8748
10x20 (150 obstacles)	0.7356	0.8748	0.8419	0.8565
10x10 (50 obstacles)	0.7807	0.8701	0.8765	0.8627

Table 4.12: Comparison of population size 50

Table 4.12 compares the performance of large grids. A high amount of obstacles actually affects the fitness and increases the diversity in grid neighborhood (FMN).

It follows, that we can influence the evolution in a SSCEA2D dependent on all conditions.

4.2.2 Simplified Robot Soccer

ID	Maximum Fitness	Diversity	Seed
1A	12	375.1016	12345
2A	14	875.4956	12345
3A	14	1978.1802	12345
4A	12	456.0329	12345
5A	14	343.9337	12345
6A	12	422.0973	12345
7A	14	261.6435	12345
8A	12	262.0488	12345
1B	14	440.7321	11111
2B	14	958.246	11111
3B	12	1932.2489	11111
4B	12	355.0129	11111
5B	12	390.5385	11111
6B	14	217.4259	11111
7B	14	284.2216	11111
8B	12	340.3382	11111

Table 4.13: Overview of maximum fitness and diversity for Simplified Robot Soccer (last generation)

In the Light! problem are the fitness values calculated with an absolute value. For the soccer game is however a relative value of fitness given. During the simulation runs, the difference between two soccer teams will be calculated. A goal equals two points and the difference between both teams are in this context the relative fitness. Table 4.13 shows also the results of each last generation. Fitness gives in this case information, how many points the soccer team during the simulation run ("training") reached. A tied game equals 1 and a won match is 2 points worth. The soccer problem needs compared to the Light! simulation much more computing time. This is why in this experiment are only two seeds used. Graphs depicting diversity for both seeds are shown in Figure 4.8 and Figure 4.9. The more obstacles are integrated, the higher is the diversity.

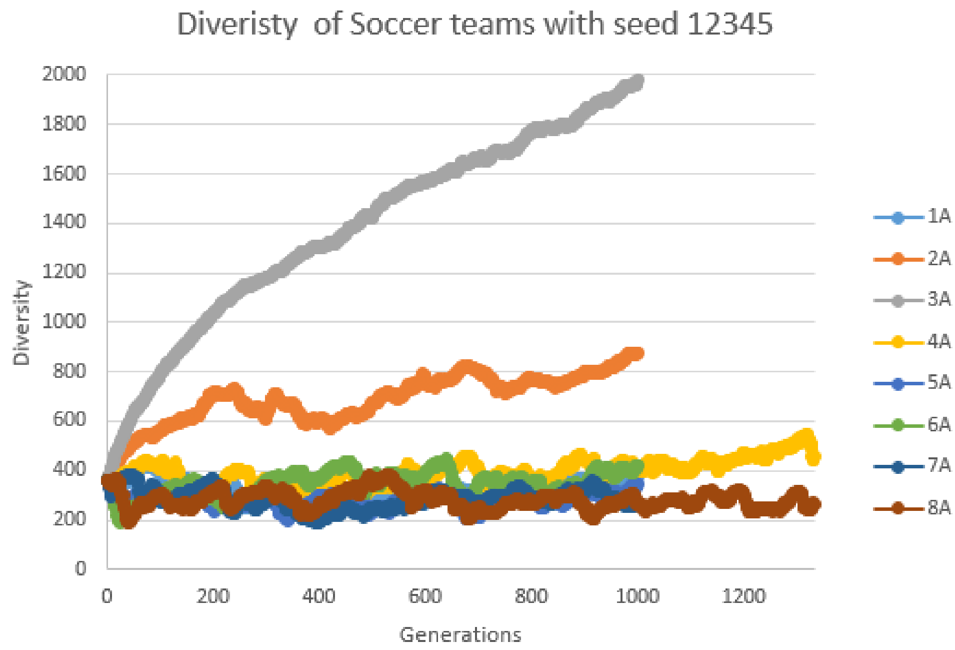

Figure 4.8: Diversity 12345

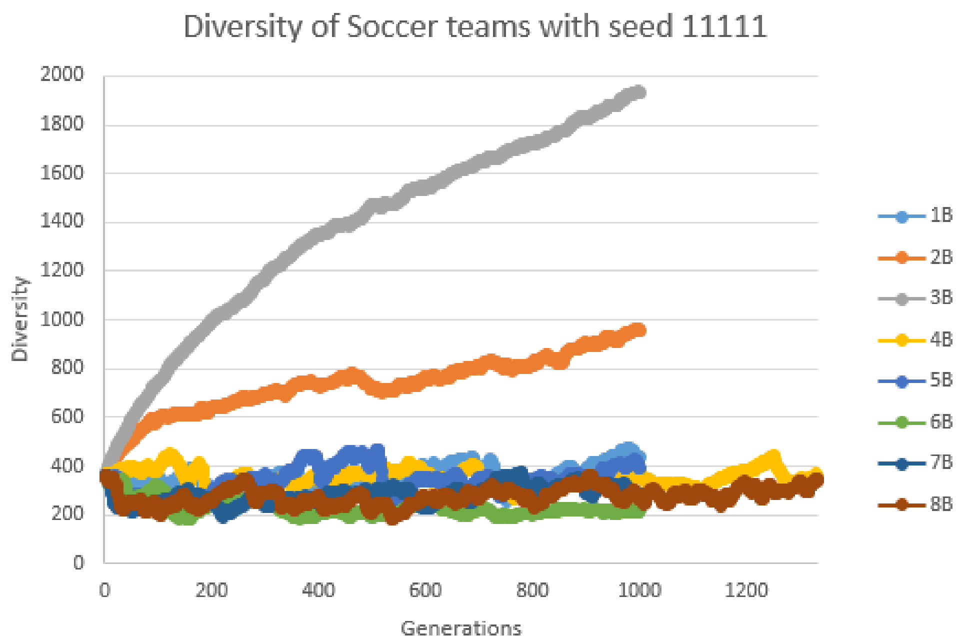

Figure 4.9: Diversity 11111

4.3 Analysis of the Distribution

With the acquired data for the Light! problem we analyzed the distributions of the fitness and diversity values of the best candidate from 100 simulation runs with different seeds but otherwise identified parameters. The tool distribution-check [1] was used, which compares data against 80 reference distributions. The tool first fits the set of data values against every possible probability distribution. The second step is to apply the "Kolmogorov-Smirnof one sided test". After executing these steps, the user gets an overview how high the probabilities of each distribution equals from the data.
You can run python programs with the command line in windows as well without editor. With $-i$ 10 we run ten iterations and with $-f$ the following file. With $>$ and a filename you can create a text file with results. Note: Names of text files are here just examples. Python program and reading file have to be in the same directory. An example how to run a python file and write a new text-file is listed as follows:

```
python distribution-check.py -i 10 -f Diversity-exp1.txt > results-exp1-diversity.txt
```

After generating the text-files, all results of the 10th iteration with 10 distributions was checked. The decision, which distribution was chosen depended on the probability and amount of parameters. If two results have a nearly equal probability, but one of them has for example four and the other less parameters for distribution, we have chosen the one with less parameters. And if the distribution with highest probability has four parameters and the next one has a high probabilistic difference, we have taken the one with more parameters. These results refer to the Light! problem and are visible in Table 4.14 and Table 4.15.

[1] http://www.aizac.info/simple-check-of-a-sample-against-80-distributions/

4.3.1 Distributions for Light!

Simulation ID	Distribution	Probability	Variance
0	genlogistic	0.9865180078488742	0.0010916111454697707
1	gumbel l	0.9924514176774271	0.0003829529789569228
2	powernorm	0.8850714396657178	0.015055501965301015
3	exponpow	0.9512592201851093	0.08304737779684349
4	johnsonsb	0.8113623309974298	0.01749743220243876
5	gumbel l	0.880805767340619	0.0050342323101667755
6	weibull min	0.9963533180007291	0.00014475100027720592
7	frechet l	0.9799343853483486	0.001023285186427505
8	gumbel l	0.9391565077062718	0.002081950362228999
9	gumbel l	0.9692462769894858	0.0018862317926438912
10	foldnorm	0.7427924035705387	0.012458854626202194
11	dweibull	0.6338456515101991	0.013054429711435046
12	dgamma	0.8957863757905299	0.01948454347443313
13	johnsonsu	0.8861649875225976	0.003531125763507211
14	gumbel l	0.9648320464875485	0.001470183389221166
15	genlogistic	0.7462400085939366	0.009898099092611487
16	genlogistic	0.8560367081426447	0.012186694762176672
17	gumbel l	0.8757788657498244	0.004180931347972064
18	genlogistic	0.9271418859244289	0.00436341796226457
19	exponpow	0.8009835209370719	0.01446574131903986
20	triang	0.9281272078919243	0.005844961128872467
21	gumbel l	0.9436945594122893	0.002304594366885253
22	gumbel l	0.9634153356519541	0.0035691706372364186
23	gumbel l	0.8781925334846985	0.008033100428903126
24	gumbel l	0.880076444212716	0.007348746894453678
25	gumbel l	0.5390299125061215	0.029026189080126642
26	cosine	0.9174576700168182	0.009996599791323852
27	hypsecant	0.9154379897466871	0.001985716379536679
28	t	0.862497906100369	0.004380951280221844
29	genlogistic	0.9904426741822332	0.00020417582289649301
30	genlogistic	0.9145607848204504	0.004286362540102615
31	johnsonsu	0.8577155510271587	0.015639107219258125
32	logistic	0.9568670919515789	0.00048548092600523874
33	logistic	0.9699372082406658	0.0067687241823444905
34	logistic	0.869541258200828	0.0018213612857060258
35	norm	0.9927893905884861	4.810579582638233e-05
36	johnsonsb	0.41347269668049436	0.013855126141793026
37	mielke	0.6162918421539887	0.02254579853768906
38	johnsonsu	0.8807851915227595	0.0021617832249472345
39	laplace	0.9593999629748642	0.0018213206624633743
40	gumbel l	0.794398350887153	0.007807995101161344
41	t	0.94329378397442	0.0032684283604215826
42	t	0.9029677316348466	0.0019352326836483402
43	johnsonsu	0.6499979788163011	0.0072649902279005585
44	johnsonsu	0.5374690241757711	0.00909830669989891
45	t	0.8877562278922329	0.0056480792834288415
46	johnsonsu	0.7947868163932238	0.007974199596783759
47	t	0.8828838219572928	0.005957322985256405
48	t	0.8945656809446928	0.01101917180823975
49	johnsonsu	0.938888126286908	0.000450928654775166
50	johnsonsu	0.31116767130984946	0.0028937829036628647
51	johnsonsu	0.3634217224405363	0.0041477779331493166

Table 4.14: Fitness distributions with highest probabilities for Light!

Simulation ID	Distribution	Probability	Variance
0	nakagami	0.9617778620630681	0.0022034024723984217
1	kstwobign	0.8076112746759581	0.020504629503767226
2	invgauss	0.9879434119107356	0.0004960603443985757
3	invgauss	0.974773357080098	0.00014949390355836894
4	invgamma	0.9741829714439085	0.0003347902935718133
5	fisk	0.9876688262879396	0.0011849128292761486
6	foldnorm	0.9800967453491569	0.001206238590510132
7	pearson3	0.8651120474145587	0.0043693254390830254
8	pearson3	0.991484959858993	0.0004568552145565321
9	alpha	0.929941390709897	0.0031886865899066114
10	gumbel l	0.9439925593026568	0.010346871230727011
11	triang	0.9877449862129872	0.0016202339016825984
12	logistic	0.9658299783249955	0.0012585407314876758
13	maxwell	0.9468977064255011	0.012506236386820765
14	pearson3	0.8761802868998224	0.004133135147760277
15	logistic	0.9714122242752623	0.00019767655217727903
16	fisk	0.9911273439542747	9.940043252892347e-05
17	gumbel r	0.9419338286603937	0.004072630531680844
18	fisk	0.9934261239435773	0.00021253441682410145
19	logistic	0.9194713307161462	0.008732093687188825
20	recipinvgauss	0.8800962520613735	0.004839900471339152
21	pearson3	0.8886334978235041	0.002351700439838416
22	recipinvgauss	0.8833203342379535	0.003688277824849833
23	kstwobign	0.8514902903938669	0.013322849806302864
24	alpha	0.99720244505233	6.035110971398484e-05
25	fisk	0.9284824507244573	0.006313445528118943
26	fisk	0.9522770003844694	0.0021188930420488012
27	johnsonsu	0.8653407156596093	0.007560727139426593
28	gumbel r	0.9924369247701756	0.000642660552944752
29	exponweib	0.9311900318859945	0.011306914479790356
30	fisk	0.9744461044737782	0.0007159568693076185
31	laplace	0.7984431453832084	0.0179269983186282
32	kstwobign	0.919768720145005	0.007941456793333917
33	fisk	0.8573661870047179	0.0015843209309802319
34	triang	0.8764614148840782	0.022512970498172857
35	frechet r	0.9939511034853605	0.0001067619306412006
36	loggamma	0.8649457604659332	0.0024778374770503646
37	beta	0.9993940998719016	6.703583160836755e-06
38	dweibull	0.9973200711580539	0.00012662304644292548
39	kstwobign	0.8968434761920221	0.003030950811692425
40	invgamma	0.9988570186729768	1.0483160438172558e-05
41	fisk	0.988627824888699	0.0004739124202708429
42	logistic	0.8881924650623545	0.0024446002790771528
43	invweibull	0.8984746782322913	0.013592484158370777
44	fisk	0.9790462961838704	0.0009768172991250056
45	laplace	0.9799000648878511	0.0008795266137496885
46	powerlognorm	0.9717785994788231	0.001601733279212662
47	maxwell	0.9924127517430433	0.000544515548733541
48	invgauss	0.9954404647913239	0.0002402348974152441
49	powerlognorm	0.8849309198574339	0.022029351575410437
50	genextreme	0.9909888313499238	0.0014655554493532993
51	fisk	0.9808173757784846	0.0005162387606615242

Table 4.15: Diversity distributions with highest probabilities for Light!

4.3.2 Description of Distributions

The three most common distributions from the Light! problem are discussed in this chapter. Formulas and graphs for Cumulative Distribution Function (**CDF**) and Probability Density Function (**PDF**) are described as well. Parameters for each distribution are varied and show how the shape of them will be changed. CDFs are lettered with capital **F(x)** and PDFs with small **f(x)**. Both functions have the relation through derivative and integration. So the CDF of a distribution is defined as the area of PDF. The area of a distribution equals between 0 and 1 because a negative probability and higher than 1 is impossible.

4.3.2.1 Gumbel l

Equation 4.2 and Equation 4.3 describe CDF and PDF each as well. μ and β are the parameters and x the abscissa. Graphical presentation of both equations are visible in Figure 4.10 and Figure 4.11.

$$F(x;\mu,\beta) = exp(-exp(-\frac{x-\mu}{\beta})) \tag{4.2}$$

$$f(x;\mu,\beta) = \frac{1}{\beta} \cdot exp(-\frac{x-\mu}{\beta}) \cdot exp(-exp(-\frac{x-\mu}{\beta})) \tag{4.3}$$

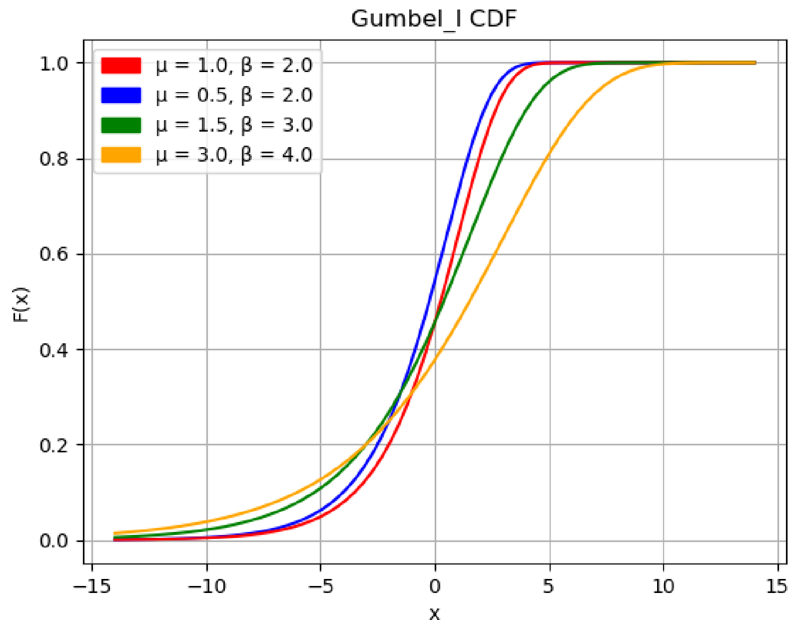

Figure 4.10: Left sided Gumbel CDF

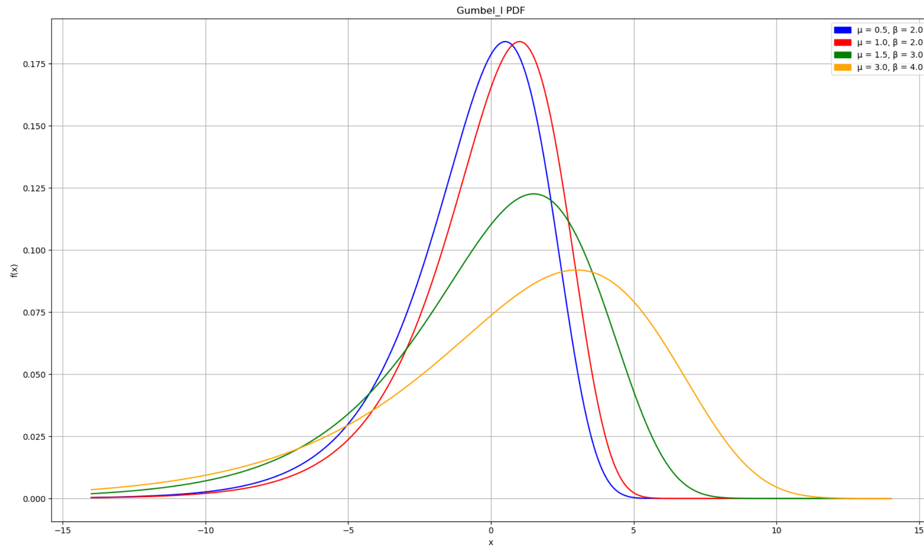

Figure 4.11: Left sided Gumbel PDF

4.3.2.2 Fisk

Another frequently distribution from the statistical results is Fisk. Equation 4.4 and Equation 4.5 show the mathematical description with parameters α and β. x represents the argument for x-axes. Figure 4.12 and Figure 4.13 show, how you can set the graphical curve by adjusting β.

$$F(x;\alpha,\beta) = \frac{1}{1+(\frac{x}{\alpha})^{-\beta}} = \frac{(\frac{x}{\alpha})^{-\beta}}{1+(\frac{x}{\alpha})^{-\beta}} = \frac{x^\beta}{a^\beta + x^\beta} \tag{4.4}$$

where x > 0, $\alpha > 0$, $\beta > 0$

$$f(x;\alpha,\beta) = \frac{(\frac{\beta}{\alpha})\cdot(\frac{x}{\alpha})^{\beta-1}}{(1+(\frac{x}{\alpha})^{\beta})^2} \tag{4.5}$$

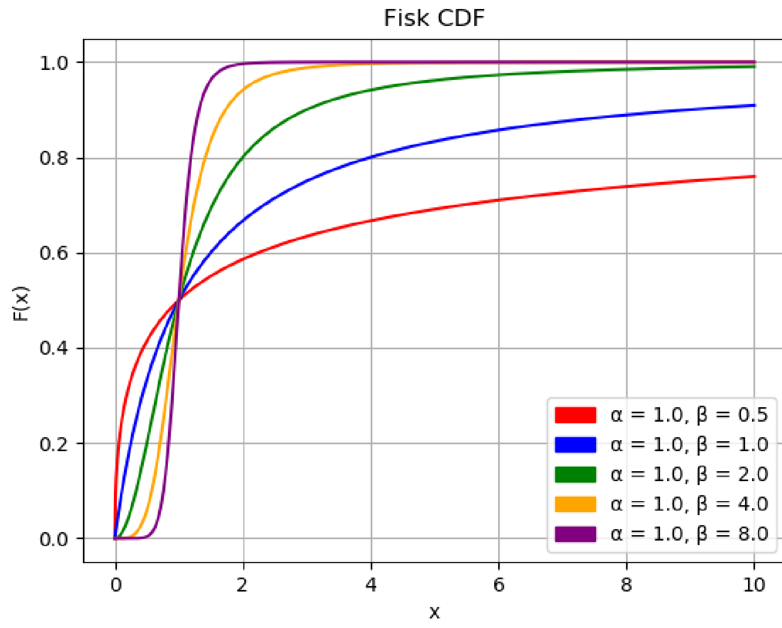

Figure 4.12: Fisk CDF

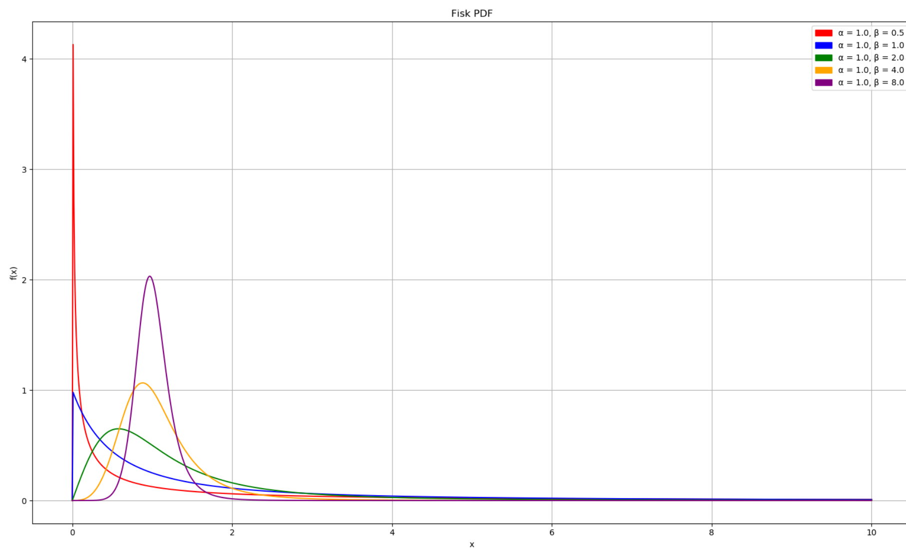

Figure 4.13: Fisk PDF

4.3.2.3 Logistic

In the logistic distribution. s shows the scale and μ as location of value in Equation 4.6 and Equation 4.7. Figure 4.14 shows the CDF and Figure 4.15 the PDF for logistic distribution.

$$F(x;\mu,s) = \frac{1}{1+exp(-\frac{x-\mu}{s})} = \frac{1}{2} + \frac{1}{2} \cdot tanh(\frac{x-\mu}{2s}) \tag{4.6}$$

$$f(x;\mu,s) = \frac{exp(-\frac{x-\mu}{s})}{s \cdot (1+exp(-\frac{x-\mu}{s}))^2} = \frac{1}{s \cdot (exp(\frac{x-\mu}{2s}) + exp(-\frac{x-\mu}{2s}))^2} \tag{4.7}$$

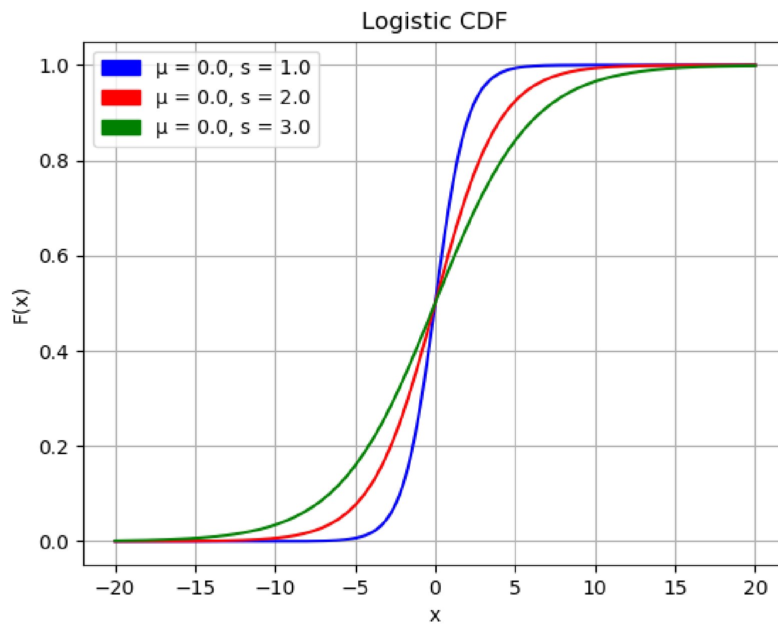

Figure 4.14: Logistic CDF

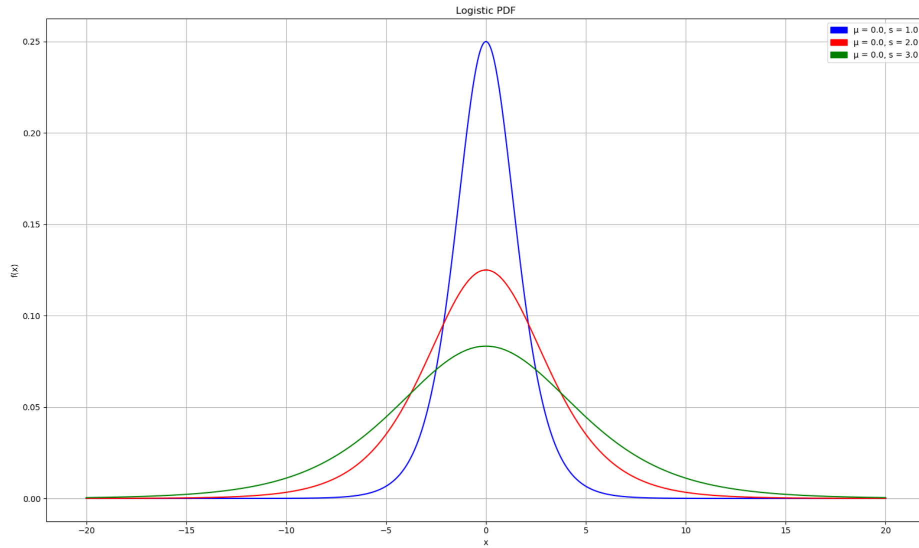

Figure 4.15: Logistic PDF

4.3.2.4 Discussion

By adjusting the parameters of each distribution you can set the width of PDF and steepness of CDF. It is noticeable how the external shape changes. The Gumbel distribution tends actually towards the left. If β gets increased, the shape of PDF shows a stronger tendency. The same behavior, but right sided shows the Fisk distribution. If β in this case gets increased, the curve has a more pointed shape and gets narrower. Only the Logistic distribution shows a centralized behavior and reminds more on the Gaussian distribution. s determines the height of curve and the mean value μ is in all three situations 0.0. By knowing the distribution of simulation results you can investigate the behavior of your simulations with the modified SSCEA2D in a statistical way. Of course many distributions are quite similar with others, but you have to consider how many parameters your distribution has. It makes a difference, if two or four parameters are necessary for a distribution and how high the probability is. Both has to be considered for correctly identifying a distribution. Once a distribution has been identified correctly, it gives us the possibility to predict the success of evolving an intended result with an EA.

4.4 Soccer tournament

In this section are the results from all soccer simulations from Table 4.2 applied for a tournament. From each ID, the best solution was selected and everyone plays against every team. 16 teams are present and all scores are visible in Table 4.16. The number of soccer matches for a full tournament can be calculated with the formula below as follows:

$$N = n \cdot (n-1) \cdot \frac{1}{2} \tag{4.8}$$

N is the sum of matches and n shows the number of teams in Equation 4.8. In this case, the result equals 120 with 16 teams.
For a realistic simulation, each team consists of 11 players, but an abridged playing time of 180s. When during this time-span a goal will be shot, this team wins the actual match. If the ball gets for example stuck by the soccer players or no goal will be shot in general and the time-span is over, the match ends in a draw.

Team	**Points**	**Team**	**Points**
1A	4	1B	3
2A	4	2B	1
3A	0	3B	0
4A	3	4B	5
5A	7	5B	0
6A	0	6B	0
7A	5	7B	4
8A	5	8B	1

Table 4.16: Points table for soccer teams

The results in Table 4.16 show how often each team has won in the tournament. Seed 12345 reached more goals than seed 11111 (however with random neighborhood). With this comparison is proved, that not only obstacles and high diversity influence the soccer team. Even the seed and which kind of neighborhood are essential. Also we can see with these results less goals with grid neighborhood. Team 3A and 3B were not good playing teams because the high diversity and large amount of obstacles affected the intelligence of each player and nobody considered what happens left and right. So if they lost the ball, they ran by and did not try to get the ball again.

Chapter 5

Conclusions and outlook

This master thesis deals with the existing SSCEA2D with new modifications. Purpose of these experiments was for researching new and high efficient solutions in SOSs. For structuring the essential parts in this research, several research questions have been set up. In the following are the responding answers and solutions discussed. The research questions are repeated as follows:

1. **How do fitness and diversity change by rectangular grid?**
   It depends on the population size (width, height, area) and which kind of neighborhood (grid or random) is used in the simulation. We cannot confirm, that a larger height or width by same population size improves or impairs the evolution. More important is, how many simulations and how many seeds were used. Grid neighborhoods have always a different evolution behavior by square and adjusting width/height. Neighbors, whose are set randomly show by same population size (without obstacles) the same fitness and diversity values.

2. **Do integrated obstacles influence the evolution in the grid?**
   Yes. Obstacles play specially in grid neighborhoods a significant role. The higher the percentage of obstacles inside the grid is integrated, the scattering in absolute fitness increases and the average diversity as well. We noticed these properties in both problems. Population size is not immediately the same if obstacles are integrated and have to be always considered.

3. **Which conditions yield the highest fitness and/or average diversity?**
   Not only the population size and integrated obstacles influence the evolution. Also a initial condition (seed) can reach high fitness and average diversity solutions. In our case we found out, that the simulation run 6 with 5x15 grid (11 obstacles) gets the maximum fitness. Simulation 3 and 34 give the closest results for average diversity (FMN and TLNN). Simulation 8 has the lowest scattering and highest median for FMN and by TLNN are the runs 47 and 48 with these properties.

4. **Does the new developed system work more efficiently compared to the existing algorithm?**
   Yes. Obstacles can clearly affect the evolution in SSCEA2D. By adjusting width and height a higher or lower fitness/diversity can expected. With obstacles we are able to increase the diversity very strongly because of individual restrictions (more different solutions cause high diversity).

5. **How is the resulting distributed?**
   Since a large proportion of distributions are very similar, the number of parameters was taken into account for each simulation. The fewer parameters, the more likely this distribution is. The reason for this assumption is that with several parameters it would be easier to replicate this mathematical distribution. Also the probability of each distribution was considered. If the probability was high enough and the number of parameters kept as low as possible, then this distribution was chosen. In the simulations of this master thesis the left-sided Gumbel, Fisk and Logistic were most frequently evaluated.

It follows that the methods and representations in FREVO have potential for evolving SOSs. Through this advanced development of EA we have new possibilities for developing SOSs with SSCEA2D. This master thesis gave an insight into this matter and potential for further developments in this area.

Bibliography

[1] E. Alba and B. Dorronsoro. Cellular genetic algorithms. Springer, 2008.

[2] A. Bagnato, R. K. Biro, D. Bonino, C. Pastrone, W. Elmenreich, R. Reiners, M. Schranz, and E. Arnautovic. Designing swarms of cyber-physical systems: the H2020 CPSwarm project. In *ACM International Conference on Computing Frontiers*, Siena, Italy, 2017.

[3] T. Balch. Communication, diversity and learning: Cornerstones of swarm behavior. In *Swarm Robotics*, pages 21–30. Springer, 2004.

[4] W. Banzhaf. Self-organizing systems. In *Department of Computer Science, Memorial University of Newfoundland, St. John's, NL, A1B 3X5, Canada*, pages 1–20.

[5] G. A. Barreto and A. F. Araujo. Identification and control of dynamical systems using the self-organizing map. In *IEEE Transactions on Neural Networks, Vol. 15, No. 5*, pages 1244–1259, 2004.

[6] E. Bonabeau, M. Dorigo, and G. Theraulaz. Swarm intelligence, from natural to artificial systems. 1999.

[7] E. Bonabeau and C. Meyer. Swarm intelligence: A whole new way to think about buisness. pages 1–11, 2001.

[8] M. Chertow and J. Ehrenfeld. Organizing self-organizing systems. In *Toward a Theory of Industrial Symbiosis*, pages 13–27, 2012.

[9] T. Dittrich and W. Elmenreich. Comparison of a spatially-structured cellular evolutionary algorithm to an evolutionary algorithm with panmictic population. In *Proceedings of the 12th International Workshop on Intelligent Solutions in Embedded Systems (WISES'15)*, pages 145–149, Ancona, Italy, Oct. 2015.

[10] M. Dorigo, E. Tuci, R. Groß, V. Trianni, T. H. Labella, S. Nouyan, C. Ampatzis, J.-L. Deneubourg, G. Baldassare, S. Nolfi, F. Mondada, D. Floreano, and L. M. Gambardella. The swarm-bots project. In *Swarm Robotics*, pages 31–44, 2004.

[11] W. Elmenreich and H. de Meer. Self-organizing networked systems for technical applications: A discussion on open issues. In J. S. K.A. Hummel, editor, *Proceedings of the Third International Workshop on Self-Organizing Systems*, pages 1–9. Springer Verlag, 2008.

[12] W. Elmenreich, R. D'Souza, C. Bettstetter, and H. de Meer. A survey of models and design methods for self-organizing networked systems. In *Proceedings of the Fourth International Workshop on Self-Organizing Systems*, volume LNCS 5918, pages 37–49. Springer Verlag, 2009.

[13] W. Elmenreich and I. Fehérvári. Evolving self-organizing cellular automata based on neural network genotypes. In *Proceedings of the Fifth International Workshop on Self-Organizing Systems*, volume LNCS 6557, pages 16–25. Springer Verlag, 2011.

[14] I. Fehérvári and W. Elmenreich. Evolutionary methods in self-organizing system design. In *Proceedings of the 2009 International Conference on Genetic and Evolutionary Methods*, pages 10–15, Las Vegas, NV, USA, 2009.

[15] I. Fehervari and W. Elmenreich. Evolving neural network controllers for a team of self-organizing robots. *Journal of Robotics*, 2010.

[16] I. Fehervari and W. Elmenreich. Evolution as a tool to design self-organizing systems. In *Self-Organizing Systems*, volume LNCS 8221, pages 139–144. Springer Verlag, 2014.

[17] C. Gershenson and F. Heylighen. When can we call a system self-organizing? In *Advances in Artifical Life*, pages 606–614, 2003.

[18] M. Giacobini, M. Tomassini, A. G. B. Tettamanzi, and E. Alba. Selection intensity in cellular evolutionary algorithms for regular lattices. In *IEEE Transactions on Evolutionary Computation*, pages 489–505, 2005.

[19] K.-H. Han and J.-H. Kim. Quantum-inspired evolutionary algorithm for a class of combinatorial optimization. In *IEEE Transactions on Evolutionary Computation, Vol. 6, No. 6, December 2002*, pages 580–593, 2002.

[20] M. G. Hinchey, R. Sterritt, and C. Rouff. Swarms and swarm intelligence. In *Software Technologies*, pages 111–113, 2007.

[21] J. H. Holland. Genetic algorithms. In *from Scientific American*, pages 66–73, 1992.

[22] R. Holzer and H. de Meer. Methods for approximations of quantitative measures in self-organizing systems. In *Self-Organizing Systems*, pages 1–15, 2011.

[23] I. F. Jr., X.-S. Yang, I. Fister, J. Brest, and D. Fister. A brief review of nature-inspired algorithms for optimization. In *Elektrotehniski Vestnik*, pages 1–7, 2013.

[24] Jun-Fang Li, Bu-Han Zhang, Yi-Fang Liu, Kui Wang, and Xiao-Shan Wu. Spatial evolution character of multi-objective evolutionary algorithm based on self-organized criticality theory. In *Physica A*, pages 5490–5499, 2012.

[25] U. Kamath, C. Domeniconi, and K. A. D. Jong. An analysis of a spatial ea parallel boosting algorithm. pages 1–8, 2013.

[26] D. Karaboga and B. Akay. A survey: algorithms simulating bee swarm intelligence. In *Artif Intell Rev*, pages 61–85, 2009.

[27] D. B. Kell. Evolutionary algorithms and synthetic biology for directed evolution: commentary on "on the mapping of genotype to phenotype in evolutionary algorithms". In *in CrossMark*, pages 373–378, 2017.

[28] T. Kohonen, S. Kaski, P. Somervuo, K. Lagus, M. Oja, and V. Paatero. Self-organizing map. pages 113–122.

[29] J. B. Kollat and P. Reed. A framework for visually interactive decision-making and design using evolutionary multi-objective optimization (video). In *Environmental Modelling and Software 22*, pages 1691–1704, 2007.

[30] N. Labroche, N. Monmarche, and G. Venturini. Antclust: Ant clustering and web usage mining. In *Genetic and Evolutionary Computation GECCO 2003*, pages 25–36. Springer, 2003.

[31] D. Patterson. Künstliche neuronale Netze. Prentice Hall, 1997.

[32] D. Penasa, J. Bangaa, P. Gonzálezb, and R. Doalloba. Enhanced parallel differential evolution algorithm for problems incomputational systems biology. In *Applied Soft Computing*, pages 86–99, 2015.

[33] F. Pinel, B. Dorronsoro, and P. Bouvry. A new parallel asynchronous cellular genetic algorithm for scheduling in grids. pages 1–8. Faculty of Science, Technology and Communications, University of Luxembourg, 2010.

[34] C. Prehofer and C. Bettstetter. Self-organization in communication networks: Principles and design paradigms. In *Advances in Self-Organizing Networks*, pages 78–85. IEEE, 2005.

[35] M. Rappaport, D. Conzon, M. Jdeed, M. Schranz, E. Ferrera, and W. Elmenreich. Distributed simulation for evolutionary design of swarms of cyber-physical systems. In *Proceedings International Conference on Adaptive and Self-Adaptive Systems and Applications*, pages 60–65, Feb. 2018.

[36] M. Schranz, A. Bagnato, E. Brosse, and W. Elmenreich. Modelling a CPS swarm system: A simple case study. In S. Hammoudi, L. F. Pires, and B. Selic, editors, *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'18)*, pages 615–624, Setubal, Jan. 2018.

[37] A. Sobe, I. Fehérvári, and W. Elmenreich. FREVO: A tool for evolving and evaluating self-organizing systems. In *Proceedings of the 1st International Workshop on Evaluation for Self-Adaptive and Self-Organizing Systems*, Lyon, France, Sept. 2012.

[38] M. Tomassini. Structured cellular evolutionary algorithms: Artificial evolution in space and time. Springer, 2005.

[39] S. von Mammen, J.-P. Steghöfer, J. Denzinger, and C. Jacob. Self-organized middle-out abstraction. In *Self-Organizing Systems*, pages 26–31, 2011.

[40] N. Xiao. Geographic optimization using evolutionary algorithms. In *Department of Geography, The Ohio State University, Columbus, OH 43210*, pages 1–14.

[41] S. Zhevzhyk and W. Elmenreich. Comparison of metaheuristic algorithms for evolving a neural controller for an autonomous robot. *Transactions on Machine Learning and Artificial Intelligence*, 2(6):62–76, 2015.

[42] J. M. Zurada. Introduction to artificial neural systems. Wadsworth Publishing Co Inc, 1992.

Appendices

Appendix A

Setting obstacles, patterns, parameters and plotting diversity

```
1  package CEA2D;
2
3  import java.util.Hashtable;
4
5  // import net.jodk.lang.FastMath;
6  import core.XMLFieldEntry;
7  import utils.NESRandom;
8
9  /**
10 * Storage of parameters for {@link CEA2D} method.
11 *
12 * @author Sergii Zhevzhyk
13 */
14 public class Parameters {
15
16 /**
17 * number of representations in the population
18 */
19 public int POPULATIONSIZE;
20 /**
21 * mode how the neighborhood of a member is defined
22 */
23 public int NEIGHBOURHOODMODE;
24 /**
25 * number of generations
26 */
27 public int GENERATIONS;
28 /**
29 * interval between two intermediate saves
30 */
31 public int SAVEINTERVAL;
32
33 /**
34 * defines the shape of the curve which represents the correlation between the
35 * rank of the fitness in the Neighborhood and the severity of the mutation.
36 * This curve has the formula f=100*r^a. Where f is the severity of mutation, a
37 * is MUTATIONSEVERITYCURVE and r is the rank of the fitness in the neighborhood
38 * divided by the number of neighbors
39 */
40 public int NUMBEROFNEIGHBORS;
41
42 /**
43 * defines how many Members are elite. The percentage of elite-members is
44 * probably not that high because if a member is elite is not calculated over
45 * the whole field but only in his neighborhood. And so it is possible, that a
46 * member is above this percentage in the neighborhood of one of his neighbours
47 * but not in his own
48 */
49 public float PERCENTELITE;
50
51 /**
52 * defines how many Generations an elite-member must exist
53 */
54 public int MINIMUMLIFETIMEELITE;
55
56 /**
57 * defines the severity of the mutation</br>
58 * 0  ......  representation does not change</br>
```

```
59  * 100 .. a totally new representation is generated
60  */
61  public float MUTATIONSEVERITY;
62
63  /**
64  * defines the probability of the mutation</br>
65  * 0  ......  representation does not change</br>
66  * 1 .. everything changes
67  */
68  public float MUTATIONPROBABILITY;
69
70  /**
71  * defines how many representations that are not elite create a mutation of a
72  * random elite-neighbor
73  */
74  public int PERCENTMUTATEELITE;
75
76  /**
77  * defines how many representations that are not elite create an offspring with
78  * a random elite-neighbor
79  */
80  public int PERCENTXOVERELITE;
81
82  /**
83  * The method which is using the current parameters
84  */
85  private CEA2D method;
86
87  public Parameters(CEA2D method) {
88  if (method == null) {
89  throw new NullPointerException();
90  }
91
92  this.method = method;
93  }
94
95  // rectangular grid for representation
96  public int POPULATIONFIELDSIZE_HEIGHT;
97  public int POPULATIONFIELDSIZE_WIDTH;
98
99  // obstacles in the grid for harder evolution
100 public int OBSTACLE_PATTERN;
101 public int OBSTACLES;
102
103 /**
104 * Initialize parameters from method's properties
105 *
106 * @param properties properties of the method
107 */
108 public void initialize(Hashtable<String, XMLFieldEntry> properties) {
109 // Get properties
110
111 // define the length of the population grid
112 XMLFieldEntry pop_height = properties.get("populationsize_height");
113 POPULATIONFIELDSIZE_HEIGHT = Integer.parseInt(pop_height.getValue());
114
115 // define the width of the population grid
116 XMLFieldEntry pop_width = properties.get("populationsize_width");
117 POPULATIONFIELDSIZE_WIDTH = Integer.parseInt(pop_width.getValue());
118
119 XMLFieldEntry neighborhoodmode = properties.get("neighbourhoodmode");
120 NEIGHBOURHOODMODE = Integer.parseInt(neighborhoodmode.getValue());
121
122 XMLFieldEntry generations = properties.get("generations");
123 GENERATIONS = Integer.parseInt(generations.getValue());
124
125 XMLFieldEntry saveint = properties.get("saveinterval");
126 SAVEINTERVAL = Integer.parseInt(saveint.getValue());
127
128 XMLFieldEntry percentelite = properties.get("percentelite");
129 PERCENTELITE = Integer.parseInt(percentelite.getValue());
130
131 XMLFieldEntry mutationseverity = properties.get("mutationseverity");
132 MUTATIONSEVERITY = Float.parseFloat(mutationseverity.getValue());
133
134 XMLFieldEntry mutationprobability = properties.get("mutationprobability");
135 MUTATIONPROBABILITY = Float.parseFloat(mutationprobability.getValue());
136
137 XMLFieldEntry percentmutateelite = properties.get("percentmutateelite");
138 PERCENTMUTATEELITE = Integer.parseInt(percentmutateelite.getValue());
139
140 XMLFieldEntry percentxoverelite = properties.get("percentxoverelite");
141 PERCENTXOVERELITE = Integer.parseInt(percentxoverelite.getValue());
142
143 // predefined obstacle-patterns
144 XMLFieldEntry obstacle_pattern = properties.get("obstacle-pattern");
145 OBSTACLE_PATTERN = Integer.parseInt(obstacle_pattern.getValue());
146
```

```
147 // number of obstacles
148 XMLFieldEntry random_obstacles = properties.get("random obstacles");
149 OBSTACLES = Integer.parseInt(random_obstacles.getValue());
150
151 // calculate the area of the grid
152 POPULATIONSIZE = (int) POPULATIONFIELDSIZE_HEIGHT * POPULATIONFIELDSIZE_WIDTH;
153
154 }
155
156 /**
157 * Gets the generator of random numbers
158 *
159 * @return the instance of {@link NESRandom} class for generating of random
160 *         numbers
161 */
162 public NESRandom getGenerator() {
163 NESRandom generator = method.getRandom();
164 if (generator == null) {
165 throw new NullPointerException();
166 }
167 return generator;
168 }
169 }
```

Listing A.1: Parameters.java

```
package CEA2D;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import org.dom4j.Document;
import org.dom4j.Node;

import CEA2D.Member.replaceFunction;
import net.jodk.lang.FastMath;
import utils.NESRandom;
import core.AbstractRepresentation;
import core.ComponentXMLData;

/**
* The class population represents the whole population for the evolutionary
* algorithm SSEA2D. It contains all the representations and the function to
* evolve a new generation.
*
* @author Thomas Dittrich
*/

public class Population {

Member[] members;
Parameters parameters;
long randomNeighborhoodSeed;
private ComponentXMLData representation;
private int inputnumber;
private int outputnumber;

private double numElite;
private double numMutate;
private double numXOver;
private double numRenew;

private double effectivityElite;
private double effectivityMutate;
private double effectivityXOver;
private double effectivityRenew;

public double getNumElite() {
return numElite;
}

public double getNumMutate() {
return numMutate;
}

public double getNumXOver() {
return numXOver;
}

public double getNumRenew() {
return numRenew;
}

public double getEffectivityElite() {
return effectivityElite;
}

public double getEffectivityMutate() {
return effectivityMutate;
}

public double getEffectivityXOver() {
return effectivityXOver;
}

public double getEffectivityRenew() {
return effectivityRenew;
}

int [][] obs_pattern;

/**
*
* @param representation ComponentXMLdata which is used to create the Members.
*                       If this constructor is called in a subclass of
*                       AbstractRepresentation the variable representation
*                       should be handed over
* @param parameters  Instance which holds the properties for each member.
*/
public Population(ComponentXMLData representation, Parameters parameters, int inputnumber, int outputnumber,
CEA2D cea2d) {
this.parameters = parameters;
```

```
88  this.representation = representation;
89  this.inputnumber = inputnumber;
90  this.outputnumber = outputnumber;
91
92  cea2d.createObstaclePattern();
93  obs_pattern = cea2d.getObstaclePattern();
94  int nummembers = 0;
95
96  for (int x = 0; x < parameters.POPULATIONFIELDSIZE_WIDTH; x++) {
97  for (int y = 0; y < parameters.POPULATIONFIELDSIZE_HEIGHT; y++) {
98  if (obs_pattern[x][y] != 1000) {
99  obs_pattern[x][y] = nummembers++;
100 }
101 }
102 }
103
104 members = new Member[nummembers];
105
106 for (int i = 0; i < members.length; i++) {
107 members[i] = new Member(representation, parameters, inputnumber, outputnumber);
108 }
109
110 if (parameters.NEIGHBOURHOODMODE == 1) {
111 SetGridneighborhood();
112 } else if (parameters.NEIGHBOURHOODMODE == 2) {
113 randomNeighborhoodSeed = parameters.getGenerator().getSeed();
114 SetRandomneighborhood(8);
115 } else {
116 SetGridneighborhood();
117 }
118 }
119
120 public Population(ComponentXMLData representation, Parameters parameters, int inputnumber, int outputnumber,
121 ArrayList<AbstractRepresentation> population, Document doc) {
122 this.parameters = parameters;
123 this.inputnumber = inputnumber;
124 this.outputnumber = outputnumber;
125
126 // members = new Member[parameters.POPULATIONFIELDSIZE_HEIGHT *
127 // parameters.POPULATIONFIELDSIZE_WIDTH];
128 members = new Member[population.size()];
129 for (int i = 0; i < members.length; i++) {
130 members[i] = new Member(population.get(i), parameters);
131 }
132
133 // get population root node
134 Node dpopulations = doc.selectSingleNode("/frevo/populations");
135 // get population size
136 List<?> npops = dpopulations.selectNodes(".//population");
137 Iterator<?> it = npops.iterator();
138 while (it.hasNext()) {
139 Node pop = (Node) it.next();
140 this.randomNeighborhoodSeed = pop.numberValueOf("./@randomNeighborhoodSeed").longValue();
141 }
142
143 if (parameters.NEIGHBOURHOODMODE == 1) {
144 SetGridneighborhood();
145 } else if (parameters.NEIGHBOURHOODMODE == 2) {
146 SetRandomneighborhood(8);
147 } else {
148 SetGridneighborhood();
149 }
150 }
151
152 /**
153 * Returns an ArrayList of IRepresentations which contains all the
154 * IRepresentations of the Members
155 *
156 * @return ArrayList of IRepresentation
157 */
158
159 public ArrayList<AbstractRepresentation> getMembers() {
160 ArrayList<AbstractRepresentation> m = new ArrayList<AbstractRepresentation>();
161
162 for (Member me : members) {
163 m.add(me.rep);
164 }
165
166 return m;
167 }
168
169 /**
170 * Evolves the IRepresentation of every member according to the evolution-rules
171 */
172 public void evolve(Step step) throws Exception {
173 NESRandom rand = parameters.getGenerator();
174
175 // get diff to all neighbors
```

```
176 for (int i = 0; i < members.length; i++) {
177 members[i]. diff = 0;
178 int j = 0;
179 double diff = 0.0;
180 for (Member n : members[i].neighbors) {
181 if (n.rep.getFitness() >= members[i].rep.getFitness()) {
182 diff += members[i].rep.diffTo(n.rep);
183 j++;
184 }
185 }
186 members[i]. diff = j > 0 ? diff / j : 0.0;
187 }
188
189 AbstractRepresentation[] newmembers = new AbstractRepresentation[members.length];
190 numElite = 0;
191 numMutate = 0;
192 numXOver = 0;
193 numRenew = 0;
194 int numEliteElite = 0;
195 int numMutateElite = 0;
196 int numXOverElite = 0;
197 int numRenewElite = 0;
198
199 for (int i = 0; i < members.length; i++) {
200 members[i].rep.setFitness(members[i].rep.getFitness() + ((double) i + 1) / 1e6);
201 }
202
203 for (int i = 0; i < members.length; i++) {
204 switch (members[i].getCreatedBy()) {
205 case ELITE:
206 numElite++;
207 break;
208 case MUTATE:
209 numMutate++;
210 break;
211 case XOVER:
212 numXOver++;
213 break;
214 case RENEW:
215 numRenew++;
216 break;
217 }
218
219 ArrayList<AbstractRepresentation> neighborhood = new ArrayList<AbstractRepresentation>();
220
221 for (Member n : members[i].neighbors) {
222 neighborhood.add(n.rep);
223 }
224 neighborhood.add(members[i].rep);
225
226 step.getRanking().sortCandidates(neighborhood, step.getProblemData(), rand);
227
228 int rankneighborhood = neighborhood.indexOf(members[i].rep);
229 float re = parameters.PERCENTELITE / 100.0f;
230 int rankelite = (int) FastMath.rint(neighborhood.size() * re);
231
232 AbstractRepresentation[] elite = new AbstractRepresentation[rankelite];
233
234 for (int j = 0; j < rankelite; j++) {
235 elite [j] = neighborhood.get(j);
236 }
237
238 if (rankelite > 0 && (rankneighborhood < rankelite
239 || members[i].rep.getFitness() == elite[rankelite - 1].getFitness())) {
240
241 newmembers[i] = members[i].rep;
242
243 switch (members[i].getCreatedBy()) {
244 case ELITE:
245 numEliteElite++;
246 break;
247 case MUTATE:
248 numMutateElite++;
249 break;
250 case XOVER:
251 numXOverElite++;
252 break;
253 case RENEW:
254 numRenewElite++;
255 break;
256 }
257 members[i].setCreatedBy(replaceFunction.ELITE);
258
259 } else {
260 int geneticoperationrand = rand.nextInt((int) (100 - parameters.PERCENTELITE));
261 if (geneticoperationrand < parameters.PERCENTMUTATEELITE) {
262 if (rankelite > 0)
263 newmembers[i] = elite[rand.nextInt(rankelite )]. clone ();
```

```
264 else
265 newmembers[i] = members[i].rep;
266 newmembers[i].mutate(parameters.MUTATIONSEVERITY, parameters.MUTATIONPROBABILITY, 1);
267 members[i].setCreatedBy(replaceFunction.MUTATE);
268 } else if (geneticoperationrand < parameters.PERCENTXOVERELITE + parameters.PERCENTMUTATEELITE) {
269 newmembers[i] = members[i].rep.clone();
270 if (rankelite > 0)
271 newmembers[i].xOverWith(elite[rand.nextInt(rankelite)], 1);
272 else
273 newmembers[i].xOverWith(neighborhood.get(rand.nextInt(neighborhood.size())), 1);
274 members[i].setCreatedBy(replaceFunction.XOVER);
275 } else {
276 newmembers[i] = representation.getNewRepresentationInstance(inputnumber, outputnumber,
277 parameters.getGenerator());
278 members[i].setCreatedBy(replaceFunction.RENEW);
279 }
280 }
281 }
282
283 // copy the new members into the population
284 for (int i = 0; i < members.length; i++) {
285 members[i].rep = newmembers[i];
286 }
287 for (int i = 0; i < members.length; i++) {
288 // System.out.println("Zeile 400, i: " + i);
289 if (members[i].rep.isEvaluated()) {
290 members[i].rep.setFitness(members[i].rep.getFitness() - ((double) i + 1) / 1e6);
291
292 }
293
294 }
295 effectivityElite = numElite == 0 ? 0 : ((double) numEliteElite) / ((double) numElite);
296 effectivityMutate = numMutate == 0 ? 0 : ((double) numMutateElite) / ((double) numMutate);
297 effectivityXOver = numXOver == 0 ? 0 : ((double) numXOverElite) / ((double) numXOver);
298 effectivityRenew = numRenew == 0 ? 0 : ((double) numRenewElite) / ((double) numRenew);
299
300 }
301 /**
302 * Sets the neighbors for every member. The Neighbors of a member are those
303 * which are adjacent in the grid
304 */
305 public void SetGridneighborhood() {
306 // add width and height for rectangular grids
307 int fieldheight = parameters.POPULATIONFIELDSIZE_HEIGHT;
308 int fieldwidth = parameters.POPULATIONFIELDSIZE_WIDTH;
309
310 for (int x0 = 0; x0 < fieldwidth; x0++) {
311 for (int y0 = 0; y0 < fieldheight; y0++) {
312 for (int x1 = -1; x1 <= 1; x1++) {
313 for (int y1 = -1; y1 <= 1; y1++) {
314 if (x1 != 0 || y1 != 0) {
315 int x = (x0 + x1 + fieldwidth) % fieldwidth;
316 int y = (y0 + y1 + fieldheight) % fieldheight;
317
318 if (obs_pattern[x0][y0] != 1000 && obs_pattern[x][y] != 1000) {
319 members[obs_pattern[x0][y0]].neighbors.add(members[obs_pattern[x][y]]);
320 }
321 }
322 }
323 }
324 }
325 }
326 }
327 /**
328 * Sets the neighbors for every member. The Neighbors of a member are selected
329 * by random
330 */
331 public void SetRandomneighborhood(int numberofneighbors) {
332 int fieldheight = parameters.POPULATIONFIELDSIZE_HEIGHT;
333 int fieldwidth = parameters.POPULATIONFIELDSIZE_WIDTH;
334
335 NESRandom localRandom = new NESRandom(randomNeighborhoodSeed);
336 for (int x0 = 0; x0 < fieldwidth; x0++) {
337 for (int y0 = 0; y0 < fieldheight; y0++) {
338 for (int i = 0; i < numberofneighbors; i++) {
339 int randValue = localRandom.nextInt(members.length);
340
341 if ((obs_pattern[x0][y0] != 1000) && (obs_pattern[x0][y0] != randValue))
342 members[obs_pattern[x0][y0]].neighbors.add(members[randValue]);
343 }
344 }
345 }
346 }
347 }
```

Listing A.2: Population.java

```
1   package CEA2D;
2
3   import java.awt.Color;
4   import java.awt.GridLayout;
5   import java.text.DecimalFormat;
6   import java.util.ArrayList;
7   import java.util.Hashtable;
8   import java.util.Iterator;
9   import java.util.List;
10
11  import javax.swing.JFrame;
12
13  import main.FrevoMain;
14
15  import org.dom4j.Document;
16  import org.dom4j.DocumentFactory;
17  import org.dom4j.Element;
18  import org.dom4j.Node;
19
20  import utils.NESRandom;
21  import utils.StatKeeper;
22  import core.AbstractMethod;
23  import core.AbstractRanking;
24  import core.AbstractRepresentation;
25  import core.ComponentType;
26  import core.ComponentXMLData;
27  import core.PopulationDiversity;
28  import core.ProblemXMLData;
29  import core.XMLFieldEntry;
30  import core.XMLMethodStep;
31  import frevoutils.JGridMap.Display;
32  import frevoutils.JGridMap.JGridMap;
33
34  /**
35  * The class SSEA2D (Spatially Structured Evolutionary Algorithm 2D) is a
36  * evolutionary algorithm that considers only the neighbors of every
37  * representation to decide if the representation remains in the next
38  * generation, mutates, creates an offspring with another representation or is
39  * replaced by a totally new representation. The representations are arranged in
40  * a two dimensional grid, where every representation has 8 neighbors.
41  *
42  * @author Thomas Dittrich
43  *
44  */
45  public class CEA2D extends AbstractMethod {
46
47  /**
48  * Parameters of the method for current experiment
49  */
50  private Parameters parameters;
51
52  private StatKeeper bfitness;
53  private StatKeeper numSimulations;
54
55  // Statistics about population diversity
56  private StatKeeper diversity;
57  private StatKeeper maxDiversity;
58  private StatKeeper minDiversity;
59  private StatKeeper standardDeviation;
60
61  private StatKeeper numElite;
62  private StatKeeper numMutate;
63  private StatKeeper numXOver;
64  private StatKeeper numRenew;
65
66  private StatKeeper effectivityElite;
67  private StatKeeper effectivityMutate;
68  private StatKeeper effectivityXOver;
69  private StatKeeper effectivityRenew;
70
71  private Population pop;
72
73  private double minfitness;
74
75  private boolean iniOK = false;
76
77  Display gridFrame;
78  JGridMap fitnessgrid;
79
80  public final static Color gray = new Color(153, 153, 153); // gray color for obstacles
81  public final static Color white = new Color(255, 255, 255); // define white color
82  public final static Color black = new Color(0, 0, 0); // define black color
83
84  public int [][] obstacle_array;
85
86  /* Constructs a new SSEA2D object */
87  public CEA2D(NESRandom random) {
```

```
88  super(random);
89  parameters = new Parameters(this);
90  }
91
92  public int [][]  getObstaclePattern() {
93  return obstacle_array;
94  }
95
96  public void createObstaclePattern() {
97  // define patterns for obstacles (at least 10*10 grid for case 1, 2 and 3!)
98
99  obstacle_array = new int[parameters.POPULATIONFIELDSIZE_WIDTH][parameters.POPULATIONFIELDSIZE_HEIGHT];
100
101 switch (parameters.OBSTACLE_PATTERN) {
102
103 case 1:
104  if (parameters.POPULATIONFIELDSIZE_WIDTH >= 10 && parameters.POPULATIONFIELDSIZE_HEIGHT >= 10) {
105 for (int i = 0; i < 7; i++) {
106 obstacle_array[i][0] = 1000;
107 }
108 for (int i = 1; i < 6; i++) {
109 obstacle_array[i][1] = 1000;
110 }
111 for (int i = 2; i < 5; i++) {
112 obstacle_array[i][2] = 1000;
113 }
114 obstacle_array [3][3] = 1000;
115
116 for (int i = 5; i < 10; i++) {
117 obstacle_array[i][9] = 1000;
118 }
119 for (int i = 6; i < 9; i++) {
120 obstacle_array[i][8] = 1000;
121 }
122 obstacle_array [7][7] = 1000;
123 }
124
125 else {
126 for (int i = 0; i < parameters.POPULATIONFIELDSIZE_WIDTH; i++) {
127 for (int j = 0; j < parameters.POPULATIONFIELDSIZE_HEIGHT; j++) {
128 obstacle_array[i][j] = 0;
129 }
130 }
131 }
132 break;
133
134 case 2:
135 // define obstacle pattern 2
136  if (parameters.POPULATIONFIELDSIZE_WIDTH >= 10 && parameters.POPULATIONFIELDSIZE_HEIGHT >= 10) {
137 for (int i = 0; i < 10; i++) {
138 obstacle_array[i][5] = 1000;
139 }
140 }
141
142 else {
143 for (int i = 0; i < parameters.POPULATIONFIELDSIZE_WIDTH; i++) {
144 for (int j = 0; j < parameters.POPULATIONFIELDSIZE_HEIGHT; j++) {
145 obstacle_array[i][j] = 0;
146 }
147 }
148 }
149 break;
150
151 case 3:
152  if (parameters.POPULATIONFIELDSIZE_WIDTH >= 10 && parameters.POPULATIONFIELDSIZE_HEIGHT >= 10) {
153 for (int i = 0; i < 6; i++) {
154 for (int j = 0; j < 4; j++) {
155 obstacle_array[j][i] = 1000;
156 }
157 }
158
159 for (int i = 6; i < 10; i++) {
160 for (int j = 5; j < 10; j++) {
161 obstacle_array[j][i] = 1000;
162 }
163 }
164 } else {
165 for (int i = 0; i < parameters.POPULATIONFIELDSIZE_WIDTH; i++) {
166 for (int j = 0; j < parameters.POPULATIONFIELDSIZE_HEIGHT; j++) {
167 obstacle_array[i][j] = 0;
168 }
169 }
170 }
171
172 break;
173
174 case 4:
175 default:
```

```
176 for (int r = 0; r < parameters.OBSTACLES; r++) {
177 NESRandom rand = parameters.getGenerator(); // define random numbers for randomly distributed obstacles
178 int obs_x = 0; // random numbers for rows
179 int obs_y = 0; // random numbers for columns
180
181 do {
182 obs_x = rand.nextInt(parameters.POPULATIONFIELDSIZE_WIDTH);
183 obs_y = rand.nextInt(parameters.POPULATIONFIELDSIZE_HEIGHT);
184
185 } while (obstacle_array[obs_x][obs_y] == 1000);
186
187 obstacle_array[obs_x][obs_y] = 1000;
188 }
189 break;
190 }
191 }
192
193 public void initialize () {
194 parameters. initialize (getProperties ());
195
196 // show fitness grid if in GUI mode
197 if (FrevoMain.isFrevoWithGraphics()) {
198 // initialize fitness grid
199 if (gridFrame == null) {
200 gridFrame = new Display(1, 1, "Spatial Fitness");
201 // gridFrame = new Display(parameters.POPULATIONFIELDSIZE_LENGTH,
202 // parameters.POPULATIONFIELDSIZE_WIDTH, "Spatial Fitness");
203 gridFrame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
204 gridFrame.setLocation(0, 0);
205 }
206 if ( fitnessgrid == null) {
207 fitnessgrid = new JGridMap(parameters.POPULATIONFIELDSIZE_WIDTH * 20,
208 parameters.POPULATIONFIELDSIZE_HEIGHT * 20, parameters.POPULATIONFIELDSIZE_WIDTH,
209 parameters.POPULATIONFIELDSIZE_HEIGHT, 2);
210
211 // a condition, if we have more obstacles than in the area
212 if (parameters.OBSTACLES >= parameters.POPULATIONFIELDSIZE_HEIGHT
213 * parameters.POPULATIONFIELDSIZE_WIDTH) {
214 System.out.println("Number of obstacles is too high for the defined grid!");
215 System.out.println(
216 "Please enter next time a lower number than populationsize_length * populationsize_width!");
217 parameters.OBSTACLES = 0;
218 System.out.println("Obstacles in the grid: " + parameters.OBSTACLES);
219 return;
220 }
221
222 else {
223 System.out.println("Obstacles in the grid: " + parameters.OBSTACLES);
224 }
225
226 // initialize color scale for fitness
227 // 0...white
228 // 1...red
229 // 50...yellow
230 // 99...green
231 // 1000...gray
232
233 for (int i = 0; i < 100; i++) {
234 int r = 0;
235 int g = 0;
236
237 if (i < 50) {
238 r = 255;
239 g = i * 255 / 50;
240 } else {
241 r = 255 - (i - 50) * 255 / 50;
242 g = 255;
243 }
244
245 int color = r * 65536 + g * 256;
246
247 fitnessgrid .addColorToScale(i, new Color(color));
248 }
249 fitnessgrid .addColorToScale(1000, gray); // define gray color for obstacles
250
251 }
252
253 gridFrame.setLayout(new GridLayout(1, 1));
254 gridFrame.add(fitnessgrid );
255 gridFrame.pack();
256 gridFrame.setVisible(true);
257 }
258 }
259
260 @Override
261 public void runOptimization(ProblemXMLData problemData, ComponentXMLData representationData,
262 ComponentXMLData rankingData, Hashtable<String, XMLFieldEntry> properties) {
263
```

```
264 // initialize evolution
265 initialize ();
266
267 pop = new Population(representationData, parameters, problemData.getRequiredNumberOfInputs(),
268 problemData.getRequiredNumberOfOutputs(), this);
269
270 createStatistics ();
271
272 try {
273 Step step = new Step(problemData, rankingData);
274
275 // Iterate through generations
276 for (int generation = 0; generation < parameters.GENERATIONS; generation++) {
277
278 step.setGeneration(generation);
279
280 if (!evolve(step)) {
281 break;
282 }
283 }
284
285 } catch (InstantiationException e1) {
286 e1.printStackTrace();
287 } catch (Exception e) {
288 e.printStackTrace();
289 }
290
291 // indicate final progress
292 setProgress(100);
293
294 // closes the window which holds the fitness grid
295 if (FrevoMain.isFrevoWithGraphics()) {
296 if (gridFrame != null) {
297 gridFrame.dispose();
298 }
299 fitnessgrid = null;
300 gridFrame = null;
301 }
302 }
303
304 @Override
305 public void continueOptimization(ProblemXMLData problemData, ComponentXMLData representationData,
306 ComponentXMLData rankingData, Hashtable<String, XMLFieldEntry> properties, Document doc) {
307 // initialize evolution
308 initialize ();
309
310 // record the best fitness over the evolution
311 Node dpopulations = doc.selectSingleNode("/frevo/populations");
312 double best_fitness = Double.parseDouble(dpopulations.valueOf("./@best_fitness"));
313 int lastGeneration = Integer.parseInt(dpopulations.valueOf("./@generation"));
314 long randomseed = Long.parseLong(dpopulations.valueOf("./@randomseed"));
315 getRandom().setSeed(randomseed);
316
317 // load initial population(s)
318 ArrayList<ArrayList<AbstractRepresentation>> loadedPops = loadFromXML(doc);
319 if (loadedPops.size() != 1) {
320 System.err.println ("Couldn't restore population from XML file");
321 return;
322 }
323
324 pop = new Population(representationData, parameters, problemData.getRequiredNumberOfInputs(),
325 problemData.getRequiredNumberOfOutputs(), loadedPops.get(0), doc);
326
327 createStatistics ();
328
329 try {
330 // evolve the whole population
331 Step step = new Step(problemData, rankingData);
332 pop.evolve(step);
333 step.setBestFitness(best_fitness);
334
335 // Iterate through generations
336 for (int generation = lastGeneration + 1; generation < parameters.GENERATIONS; generation++) {
337
338 step.setGeneration(generation);
339
340 if (!evolve(step)) {
341 break;
342 }
343 }
344
345 } catch (InstantiationException e1) {
346 e1.printStackTrace();
347 } catch (Exception e) {
348 e.printStackTrace();
349 }
350
351 // indicate final progress
```

```
352 setProgress(100);
353
354 // closes the window which holds the fitness grid
355 if (FrevoMain.isFrevoWithGraphics()) {
356 gridFrame.dispose();
357 fitnessgrid = null;
358 gridFrame = null;
359 }
360 }
361
362 private boolean evolve(Step step) throws Exception {
363 // set progress
364 setProgress((float) step.getGeneration() / (float) parameters.GENERATIONS);
365
366 boolean doSave = false;
367
368 AbstractRanking ranking = step.getRanking();
369 // evaluates all members and calculates the best fitness
370 ArrayList<AbstractRepresentation> memberrepresentations = pop.getMembers();
371
372 int numSims = ranking.sortCandidates(memberrepresentations, step.getProblemData(),
373 new NESRandom(generator.getSeed()));
374
375 bfitness.add(memberrepresentations.get(0).getFitness());
376
377 if (memberrepresentations.get(0).getFitness() > step.getBestFitness()) {
378 step.setBestFitness(memberrepresentations.get(0).getFitness());
379 doSave = true;
380 }
381
382 numSimulations.add(numSims);
383
384 PopulationDiversity diversityCalc = new PopulationDiversity(pop.getMembers());
385 diversity.add(diversityCalc.getAverageDiversity());
386 maxDiversity.add(diversityCalc.getMaxDiversity());
387 minDiversity.add(diversityCalc.getMinDiversity());
388 standardDeviation.add(diversityCalc.getStandardDeviation());
389
390 numElite.add(pop.getNumElite());
391 numMutate.add(pop.getNumMutate());
392 numXOver.add(pop.getNumXOver());
393 numRenew.add(pop.getNumRenew());
394
395 effectivityElite.add(pop.getEffectivityElite());
396 effectivityMutate.add(pop.getEffectivityMutate());
397 effectivityXOver.add(pop.getEffectivityXOver());
398 effectivityRenew.add(pop.getEffectivityRenew());
399
400 if (FrevoMain.isFrevoWithGraphics()) {
401 // shows the fitness of the whole population as a grid of
402 // colors, where red means bad fitness and green means good
403 // fitness
404 updatefitnessgrid();
405 }
406
407 // save periodically
408 if ((parameters.SAVEINTERVAL != 0) && (step.getGeneration() % parameters.SAVEINTERVAL == 0)) {
409 doSave = true;
410 }
411
412 // save last generation
413 if (step.getGeneration() == parameters.GENERATIONS - 1) {
414 doSave = true;
415 }
416
417 String fitnessstring;
418 if (step.getProblemData().getComponentType() == ComponentType.FREVO_PROBLEM) {
419 fitnessstring = " (" + step.getBestFitness() + ")";
420 } else {
421 // multiproblem
422 fitnessstring = "";
423 }
424
425 long currentActiveSeed = getRandom().getSeed();
426 String fileName = getFileName(step.getProblemData(), step.getGeneration(), fitnessstring);
427 Element xmlLastState = saveResults(step.getGeneration());
428 xmlLastState.addAttribute("best_fitness", String.valueOf(step.getBestFitness()));
429 // save the last state of evaluation
430 XMLMethodStep state = new XMLMethodStep(fileName, xmlLastState, this.seed, currentActiveSeed);
431 setLastResults(state);
432
433 if (doSave) {
434 FrevoMain.saveResult(fileName, xmlLastState, this.seed, currentActiveSeed);
435 }
436
437 if (step.getBestFitness() >= step.getMaxFitness()) {
438 // fill up remaining space in statkeeper with last value
439 if (bfitness.length() != parameters.GENERATIONS) {
```

```
440   int dif = parameters.GENERATIONS − bfitness.length();
441   double lastvalue = bfitness.getValues().get( bfitness .length() − 1);
442   for (int i = 0; i < dif; i++) {
443   bfitness .add(lastvalue );
444   }
445   }
446
447   return false;
448   }
449
450   if (handlePause()) {
451   // closes the window which holds the fitnessgrid
452   if (gridFrame != null)
453   gridFrame.dispose();
454   fitnessgrid = null;
455   gridFrame = null;
456   return false;
457   }
458
459   // mutates all members of the population according to the
460   // specified mutation rules (only if it's not the last
461   // generation)
462   if (step.getGeneration() != parameters.GENERATIONS − 1) {
463   pop.evolve(step);
464   }
465
466   return true;
467   }
468
469   private String getFileName(ProblemXMLData problemData, int generation, String fitnessstring) {
470   DecimalFormat fm = new DecimalFormat("000");
471
472   return problemData.getName() + "_g" + fm.format(generation) + fitnessstring;
473   }
474
475   private static ArrayList<AbstractRepresentation> createList(Node nd) {
476   ArrayList<AbstractRepresentation> result = new ArrayList<AbstractRepresentation>();
477
478   ComponentXMLData representation = FrevoMain.getSelectedComponent(ComponentType.FREVO_REPRESENTATION);
479
480   try {
481   List<?> npops = nd.selectNodes("./*");
482   Iterator<?> it = npops.iterator();
483   int size = npops.size();
484   int currentIndex = 0;
485   while (it.hasNext()) {
486   // set loading progress
487   FrevoMain.setLoadingProgress((float) currentIndex / size);
488
489   Node net = (Node) it.next();
490   size−−;
491   if ( size % 10 == 0)
492   size = size + (2 ∗ 2 − 4);
493   AbstractRepresentation member = representation.getNewRepresentationInstance(0, 0, null);
494   member.loadFromXML(net);
495   result .add(member);
496
497   currentIndex++;
498   }
499   } catch (Exception e) {
500   e.printStackTrace();
501   }
502
503   return result;
504   }
505
506   /∗∗ Saves all population data to a new XML element and returns it. ∗/
507   public Element saveResults(int generation) {
508   Element dpopulations = DocumentFactory.getInstance().createElement("populations");
509
510   dpopulations.addAttribute("count", String.valueOf(1));
511   dpopulations.addAttribute("generation", String.valueOf(generation));
512   dpopulations.addAttribute("randomseed", String.valueOf(this.getSeed()));
513
514   Element dpop = dpopulations.addElement("population");
515   dpop.addAttribute("randomNeighborhoodSeed", String.valueOf(pop.randomNeighborhoodSeed));
516
517   // sort candidates with decreasing fitness
518   ArrayList<AbstractRepresentation> members = pop.getMembers();
519
520   for (AbstractRepresentation n : members) {
521   n.exportToXmlElement(dpop);
522   }
523
524   return dpopulations;
525   }
526
527   @Override
```

```
528 public ArrayList<ArrayList<AbstractRepresentation>> loadFromXML(Document doc) {
529 // final list to be returned
530 ArrayList<ArrayList<AbstractRepresentation>> populations = new ArrayList<ArrayList<AbstractRepresentation>>();
531
532 // get population root node
533 Node dpopulations = doc.selectSingleNode("/frevo/populations");
534
535 // get number of current generation
536 int currentGeneration = Integer.parseInt(dpopulations.valueOf("./@generation"));
537
538 // get population size
539 List<?> npops = dpopulations.selectNodes(".//population");
540 Iterator<?> it = npops.iterator();
541 while (it.hasNext()) {
542 Node pop = (Node) it.next();
543 ArrayList<AbstractRepresentation> pops = createList(pop);
544 populations.add(pops);
545 }
546 // Load the number of generations
547 XMLFieldEntry gensize = getProperties().get("generations");
548 if (gensize != null) {
549 int generations = Integer.parseInt(gensize.getValue());
550 // TODO check max fitness also
551 // set boolean value which shows possibility of continuation of experiment
552 // if maximum number of generations hasn't been reached.
553 setCanContinue(currentGeneration + 1 < generations);
554 }
555
556 return populations;
557 }
558
559 /**
560 * displays the fitness of the actual population in a Grid
561 */
562
563 public void updatefitnessgrid() {
564 // determine maximum and minimum fitness
565 ArrayList<AbstractRepresentation> rep = pop.getMembers();
566 double maxfitness = rep.get(0).getFitness();
567 if (!iniOK) {
568 minfitness = rep.get(0).getFitness();
569 }
570 for (AbstractRepresentation r : rep) {
571 if (r.isEvaluated()) {
572 if (r.getFitness() > maxfitness) {
573 maxfitness = r.getFitness();
574 } else if (r.getFitness() < minfitness) {
575 minfitness = r.getFitness();
576 }
577 }
578 }
579
580 // normalize fitness between 0 and 100
581 double k = 100.0 / (maxfitness − minfitness);
582 double d = −(minfitness * k);
583 // System.out.println("H: " + parameters.POPULATIONFIELDSIZE_HEIGHT + " W: " +
584 // parameters.POPULATIONFIELDSIZE_WIDTH);
585 // int[][] fitnessarray = new
586 // int[parameters.POPULATIONFIELDSIZE_WIDTH][parameters.POPULATIONFIELDSIZE_HEIGHT];
587
588 // 3−dimensional array for the fitness grid and obstacle grid
589 int [][][] three_dim = new int[parameters.POPULATIONFIELDSIZE_WIDTH][parameters.POPULATIONFIELDSIZE_HEIGHT][2];
590
591 for (int y = 0; y < parameters.POPULATIONFIELDSIZE_HEIGHT; y++) {
592 for (int x = 0; x < parameters.POPULATIONFIELDSIZE_WIDTH; x++) {
593 if (pop.obs_pattern[x][y] != 1000)
594 if (rep.get(pop.obs_pattern[x][y]).isEvaluated()) {
595 int normfitness = (int) (rep.get(pop.obs_pattern[x][y]).getFitness() * k + d);
596
597 three_dim[x][y][0] = normfitness;
598 fitnessgrid.repaint();
599 }
600
601 else {
602 three_dim[x][y][0] = 0;
603 }
604 else
605 three_dim[x][y][1] = pop.obs_pattern[x][y];
606 }
607 }
608 // show normalized fitness in fitness grid
609 fitnessgrid.setData(three_dim);
610 fitnessgrid.repaint();
611 iniOK = true;
612 }
613
614 private void createStatistics() {
615 // bfitness = new StatKeeper(true, "Best Fitness ("+ FrevoMain.getCurrentRun() +
```

```
616 // ")", "Generations");
617 bfitness = new StatKeeper(true, "Best fitness", "Generations");
618
619 numSimulations = new StatKeeper(true, "numSimulations" + FrevoMain.getCurrentRun(), "Generations");
620
621 diversity = new StatKeeper(true, "Diversity", "Generations");
622 maxDiversity = new StatKeeper(true, "Max. diversity", "Generations");
623 minDiversity = new StatKeeper(true, "Min. diversity", "Generations");
624 standardDeviation = new StatKeeper(true, "Deviation", "Generations");
625
626 numElite = new StatKeeper(true, "number of Elite", "Generations");
627 numMutate = new StatKeeper(true, "number of Mutation", "Generations");
628 numXOver = new StatKeeper(true, "number of XOver", "Generations");
629 numRenew = new StatKeeper(true, "number of Renew", "Generations");
630
631 effectivityElite = new StatKeeper(true, "effektivity of Elite", "Generations");
632 effectivityMutate = new StatKeeper(true, "effektivity of Mutation", "Generations");
633 effectivityXOver = new StatKeeper(true, "effektivity of XOver", "Generations");
634 effectivityRenew = new StatKeeper(true, "effektivity of Renew", "Generations");
635
636 // Collect best fitness
637 FrevoMain.addStatistics(bfitness, true);
638
639 // Collect diversity
640 FrevoMain.addStatistics(diversity, true);
641 }
642 }
```

Listing A.3: CEA2D.java

Appendix B

Reading text files for generating boxplots

```
1  import matplotlib.pyplot as plt
2
3  f1 = open("my_file1.txt", "r")
4  data1 = []
5  for line in f1:
6  data1.append(float(line))
7  f1.close()
8
9  f2 = open("my_file2.txt", "r")
10 data2 = []
11 for line in f2:
12 data2.append(float(line))
13 f2.close()
14
15 data_to_plot = [data1, data2]
16
17 fig, ax = plt.subplots()
18 ax.boxplot(data_to_plot)
19
20 plt.xlabel('My_x_values')
21 plt.ylabel('My_y_values')
22 plt.title('My␣Box␣plots')
23 plt.show()
```

Listing B.1: boxplot.py

Appendix C

New parameters for FREVO interface

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE zion SYSTEM "..//IComponent.dtd">
3
4  <icomponent>
5  <config>
6  <entry key="classdir" type="STRING" value="CEA2D/CEA2D" />
7  <entry key="classname" type="STRING" value="CEA2D.CEA2D" />
8  <entry key="name" type="STRING" value="CEA2D" />
9  <entry key="description" type="STRING"
10 value="2-dimensional cellular Evolutionary Algorithm (cEA). The
11  cells/candidates are aligned on a NxN torus surface.
12 Note: If you wish a predefined obstacle pattern, please enter '1',
13  '2' or '3' in 'obstacle-pattern' (choose for them at least a
14 10*10 grid, please!).' random-obstacles' means, that you can
15 enter a certain number of obstacles and they will be placed
16 randomly (even 0, so no obstacles). Enter for this mode '4'
17 in 'obstacle-pattern'. The grid-size for 'random-obstacles'
18 is not criticle."></entry>
19 <entry key="image" type="STRING" value="CEA2D.png" />
20 <entry key="tags" type="STRING" value="CEA2D TAG" />
21 </config>
22 <properties>
23 <propentry key="generations" type="INT" value="200" />
24 <!--- <propentry key="populationsize" type="INT" value="10" />-->
25 <propentry key="populationsize_width" type="INT" value="10" />
26 <propentry key="populationsize_height" type="INT" value="10" />
27 <propentry key="obstacle-pattern" type="INT" value="0" />
28 <propentry key="random obstacles" type="INT" value="0" />
29 <propentry key="neighbourhoodmode" type="INT" value="1" />
30 <propentry key="saveinterval" type="INT" value="0" />
31 <propentry key="mutationseverity" type="FLOAT" value="0.3f" />
32 <propentry key="mutationprobability" type="FLOAT" value="1" />
33 <propentry key="percentmutateelite" type="INT" value="59" />
34 <propentry key="percentelite" type="INT" value="11" />
35 <propentry key="percentxoverelite" type="INT" value="30" />
36 </properties>
37 </icomponent>
```

Listing C.1: CEA2D.xml