FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Artificial Life Simulation
Student:	David Pešák
Supervisor:	Ing. Mgr. Ladislava Smítková Janků, Ph.D.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2023/2024

Instructions

The aim of thesis is to create an "artificial life" micromodel (simulation of individual agents). Design a representation of agent properties and actions for agents to interact with the environment and with each other. Apply the principles of evolutionary computing in the simulation.

1. Conduct a survey of the state of research in the field of artificial life simulation.
2. Design and implement a representation of the agents' "life" environment with parameterizable and time-varying resource quantities and locations.
3. Design and implement agents, define agent properties, their representation, rules for property evolution over time, and a set of actions for agents to interact with the environment and with each other. Apply the principles of evolutionary computing (artificial genome, crossbreeding, population).
4. Design, conduct and evaluate experiments.

Literature:

1. K. Kim and S. Cho, "A Comprehensive Overview of the Applications of Artificial Life," in Artificial Life, vol. 12, no. 1, pp. 153-182, 2006
2. M. Komosinski, A. Adamatzky, Artificial Life Models in Software, Springer Science & Business Media, 2009

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 5 December 2022 in Prague.

Bachelor's thesis

ARTIFICIAL LIFE SIMULATION

David Pešák

Faculty of Information Technology
Department of Applied Mathematics
Supervisor: Ing. Mgr. Ladislava Smítková Janků, Ph.D.
May 11, 2023

Contents

List of Figures

List of code listings

I would like to thank to my thesis supervisor Ing. Mgr. Ladislava Smítková Janků, Ph.D who helped me to clarify the subject and provided useful information for this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 11, 2023 .....

Abstract

This bachelor thesis deals with the simulation of artificial life. The simulation model is based on the principle of multi-agent system, meaning that each agent (organism) in the simulation is simulated independently of the others, but the agents can still interact with each other. The evolution of the agents is guided by evolutionary algorithms, more precisely genetic algorithms, using the principle of artificial genome that determines the characteristics and behaviour of the agent. The environment in which agents (or populations of agents) live is interactable and time-varying, placing greater demands on their ability to adapt. The entire program can be easily modified with a wide range of parameters, extending the actions that agents are allowed to perform and adding or modifying the biomes that instantiate the environment. The purpose of this work is to create a program capable of such simulation and to create a platform for future development of this program, i.e. the program must be easily extensible and independent of implementation modules.

Keywords artifical life, simulation, multi-agent system, micromodel, genetic algorithm, population of agents, cellular automata

Abstrakt

Tato bakalářská práce se zabývá simulací umělého života. Model simulace je založen na principu multi-agentího systému, tedy každý agent (organismus) v simulaci je simulován nezávisle na ostatních, nicméně agenti stále mohou interagovat mezi sebou. Vývoj agentů se řídí evolučními, přesněji řečeno genetickými, algoritmy s využitím principu umělého genomu určující vlastnosti a chování agenta. Prostředí, ve kterém agenti, popř. populace agentů, žijí, je interagovatelné a časově proměnné, kladoucí větší nároky na schopnost adaptace. Celý program lze jednoduše upravovat velkou škálou parametrů, rozšiřovat o akce, které agenti smí provést, a přidávat nebo upravovat biomy, které ztvárňují prostředí. Účelem této práce je vytvořit program schopný takovéto simulace a vytvořit tak platformu pro budoucí rozvoj tohoto programu, tedy program musí být lehce rozšiřitelný a nezávislý na implementačních modulech.

Klíčová slova umělý život, simulace, multi-agentní systém, mikromodel, genetický algoritmus, populace agentů, celulární automat

List of abbreviations

AI	Artificial intelligence
ALife	Artificial life
CA	Cellular automata
DSA	Diamond-Square algorithm
GA	Genetic algorithm
MAS	Multi-Agent System
ML	Machine learning
MT19973	Mersenne Twister 19937 generator
PN	Perlin noise

Chapter 1
Introduction

Artificial intelligence (AI) and machine learning (ML) in general have become a great trend in recent times. From language models that allow AI to have a conversation with us, to food sales management combining robotics and AI, or diagnosing patients and predicting their health. With AI, we can also simulate the behaviour of living organisms. For example, this is particularly useful for solving optimisation problems. Another use can be to explore biological principles from real life. One such principle is Darwin's theory of evolution.

Since Charles Darwin introduced evolutionary theory to the field of science, many computer scientists and engineers have attempted to replicate and interpret his ideas in simulations of artificial life. However, the problem is so complex that each simulation is different in some way, as it cannot cover the whole issue in the smallest detail. I will attempt to create similar simulation in my thesis.

The simulation model will be based on the principle of multi-agent system, taking an approach to the problem from a microscopic point of view – each agent can be simulated separately, in parallel with other agents. The model will be built on modules specifying both the types and attributes of the agents, and their interactions with other agents and the environment. The system can therefore be easily expanded and parameterized without the need to modify the original source code.

The environment, as well as the agents, will allow a large range of properties to be parameterized, where even a majority of them will be additionally designed to be time-varying. The environment will also respond to actions of agents. For the simulation to achieve interesting and preferably unpredictable results, the environment must be as diverse as possible. Therefore, as with agent design, the environment can be extended with modules. It's variety and variability will thus expose agents to obstacles that they must overcome in order to survive. At the same time, these changes will allow agents to learn and then more easily adapt.

I will investigate which methods and strategies are most appropriate for simulating artificial life and then to what extent and how the environment will affect the agents, or the agent population. Agents could adapt to their environment using feedback learning, of which there are several types and kinds.

I would like to follow up on this work with a master's thesis, which I have outlined in the thesis objectives.

Aim

The goal of this thesis is to design and develop a reliable and easily expandable program that simulates basic interactions between agents and their environment and between agents them-

selves. Then conduct experiments in this simulation to determine which agents fared best in survival.

This program will be the basis for the enhanced model with 2 more modules. The first one will be a research module predicting the evolution of inventions and technologies, therefore helping agents to adapt to the environment and prepare for future changes in it. The second module will extend the interactions between agents in such way that they will form communities, (dis)classify themselves into social classes and groups and (dis)maintain a certain ”status quo”.

Chapter 2
Related work

This chapter provides comprehensive overview of the models and methods that are used or were used for simulating artificial life. Some of the models are also described more in depth in the following chapters.

Due to it's nature, the study of artificial life (ALife) is divided into many categories, not only with it's implementation, but also with it's usage. As nature itself is exceptionally complex and ALife tries to synthesize portions of it, one computational model simply can not cover the entire issue. Over the years, scientists have devised numerous models solving numerous problems, from one of the first imitation of ALife by John Conway, creator of Conway's game of life based on principles of cellular automata, to advanced ALife models used to diagnose lung cancer in modern medicine. We can get a small glance of the development of research in this field in chronological order:

Martin Gardner [1] examines a mathematical concept designed by John Conway know as Conway's Game of Life. Author describes the beauty of simple game, where individual cells, placed into grid, interact with each other and, by following basic rules about their "evolution", they form exciting patterns. One group formed a pattern that ensures survival for all the forming cell, one group would cycle through more patterns forming an oscillator, or one that even could replicate itself. All these patterns, especially the self-replicating one, demonstrated the power of computational models in simulating life-like phenomena.

Gardner also mentioned the potential of artificial life simulations as a tool for studying emergent properties and foundations of nature in real life. The paper also invites other scientific fields to study ALife and use it for their needs.

Andrew Ilachinski [2] presents a model simulating combat based on multi-agent system (MAS). The paper introduces EINSTein, simulator which tries to provide a realistic representation of combat dynamics. A. Ilachinski describes advanced algorithms and artificial intelligence techniques that simulate agents' behaviour. From strategic planning to situational awareness ability to adapt. The simulations also provides an option to analyze different strategies and tactics agents can choose. The paper also talks about the applications of models like EINSTein – based on a combination of MAS and artificial intelligence.

"Avida: A Software Platform for Research in Computational Evolution"[3], written by Charles Ofria and Claus O. Wilke, is another influential paper in the field of ALife. It introduces the Avida software platform, which provides an extremely vast customization of digital organisms, statistical tools and customizable environment. This paper presents the design, features, and applications of Avida. It also provides in-depth description about how to use the Avida software, mentions it's capabilities and limitations and concluded experiments.

Apart from biology, physics and computer science, ALife also contributes to art. A great

example regarding art as a music can be paper "Putting Some (Artificial) Life Into Models of Musical Creativity"[4] published by Peter M. Todd and Eduardo Reck Miranda.

Authors in this paper [4] describe the integration of ALife into music composition. They analyze 3 main ways of how could this be achieved and what role exactly ALife can play in terms of music. The paper talks about how agents can create music "*not only for their human listeners, but in some cases for each other as well*". In a nutshell, agent's behaviour is converted from non-musical to sounds and songs – melody, harmony, and rhythm that comes from agent's interactions. As the agent is evolving, so is the music he produces. The authors also discuss the benefits, challenges, and potential outcomes of using these approaches to enhance the generative capabilities of musical models.

Lastly, the ALife also helps in modern medicine, for example Cheran, S.C. and Gargano, G. [5] examines the use of ALife as a method to diagnose Lung Cancer from CT images. They describe several algorithms used in this matter, but concerning ALife, they mention a well-known paradigm in the field – simulating Ant colony [6], typically employed for finding the shortest path, in this example Ant colony is used to reconstruct the bronchial and the vascular trees in lungs. The authors also mention the room for improvement in used software and how the Ant colony model can be extend by specialized ants.

Chapter 3

Theoretical and conceptual background

This chapter examines the issue of simulating artificial life, the current state of research in this field and everything related to it.

Since the study of artificial life encompasses a vast range of other scientific fields apart from computer science, like biology, physics, or even sociology, some terms and concepts have to be explained. This thesis however does not cover the whole problematic, only the subjects used for creating the simulator which was used for conducting experiments (see chapter 5).

3.1 Artificial life

The study of artificial life (also known as ALife) focuses on developing models that imitate lifelike systems. However, as it could seem, such models are not strictly interpreted through computers, as mentioned by Komosinski, M., & Adamatzky, A. (Eds.): "*…but also take place in wetware, using techniques from the biochemical laboratory.*"[7], since life itself is incredibly complex, everything cannot be written in computer program. The purpose of these models is to explore the fundamental aspects of life in nature, including evolution, emergence of life, cooperation and intelligence, in order to inspire solutions to complex problems by studying the behavior of organisms in these models. As mentioned earlier, the field of artificial life is very vast and diverse, encompassing many models and its applications, as shown by K. J. Kim and S. B. Cho in "A Comprehensive Overview of the Applications of Artificial Life" in figure 3.1.

Given that describing every model at once can appear very abstract, we can get a better idea about ALife by introducing specific methodologies used in this field.

For instance, agent-based models aim to simulate the behavior of individual agents while introducing a common goal for the agents to accomplish. Another example is cellular automata, which is a grid of cells where each cell can exist in a finite number of states. In each iteration in discrete time, a set of rules is applied to every cell, creating interesting patterns and behaviors. Lastly, genetic algorithms are used to solve optimization problems by representing the solution domain as genetic information and evaluating the fitness function to determine how well a specific solution (genome) performs.

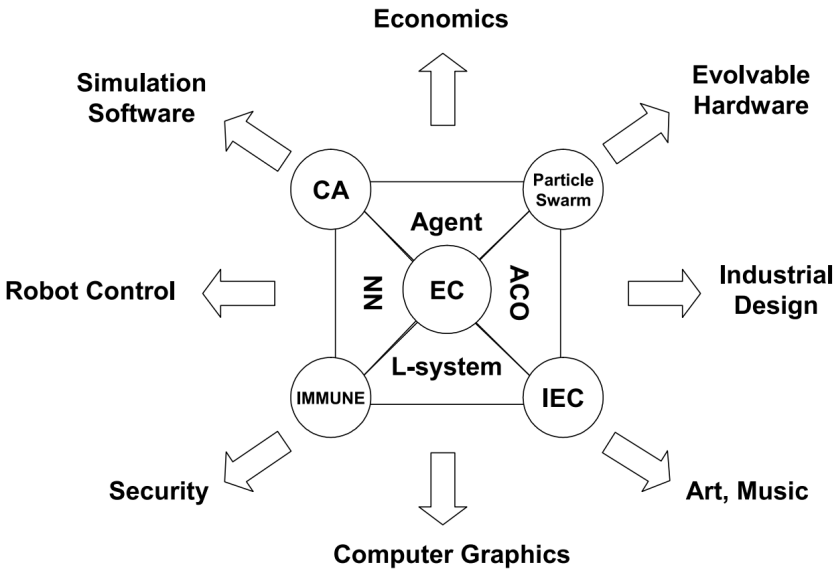

Figure 3.1 "*Relations of the ALife methodologies (CA = cellular automata, IEC = interactive evolutionary computation, EC = evolutionary computation, NN = neural network, ACO = ant colony optimization).*"[8]

3.2 Procedural generation

Procedural generation is a method of creating data with mathematical functions and algorithms. For example, instead of drawing a picture of a valley and then scanning it to the computer, we can create a program that, based on some parameters, draws the valley for us and also provides option to draw different versions of valleys at the same time. Another example is modeling 3D models of trees. Their branches and leaves (see figure 3.2) follow a specific pattern, which can be imitated with a computer algorithm. Then we can easily create entire forests and regions of trees without any additional work needed.

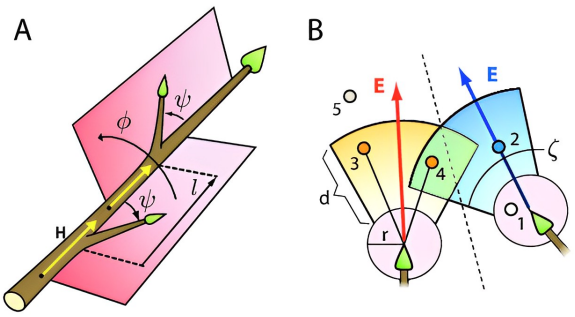

Figure 3.2 "*Buds in space. A) Distribution of buds and lateral branches is defined by the internode length l, phyllotactic angle f and branching angle y. At the branching point, the main branch may deflect in the direction opposite to the lateral bud by angle d (not shown). B) Competition of buds for space. Circles of radius r represent the spherical zones occupied by buds. The colored areas represent each bud's volume of perception, characterized by radius r + d and angle z. Dots represent markers of free space, for which buds compete (details in the text). The average direction towards the markers associated with a bud defines its preferred growth direction E E E.*"[9]

Procedural generation is widely used for terrain generation and texture generation. In computer graphics, procedural textures are very useful for almost any 3D modeling as it provides unlimited resolution (since the texture mainly follows mathematical equations), low storage size and easier texture wrapping (placing and adjusting textures on objects).

For procedural terrain generations exists several techniques, some of them described in the following sections.

3.2.1 Perlin noise

Perlin noise (PN) created by Ken Perlin [10] is an algorithm used for generating digital noise. PN falls into category of gradient noise generators. A gradient represents the gradual change of a quantity over a specific spatial or mathematical domain. In simple terms, gradients provide information about how things change as they move around. For example, when the concept of gradient is used in terrain generation, it shows how quickly the terrain changes in height, giving a perspective on where the terrain is steep and where it is flat.

Noise in computer science refers to random or pseudo-random data usually generated in order to introduce controlled randomness into procedural content generation.

PN at it's core generates values between -1 and 1. It can be used for generating noise in n-dimensional spaces although the original formulation of PN is used mainly in two or three dimensional spaces. The PN is generated in the following steps:

1. Initialization – create a grid of points representing vertexes of grid of cells.
2. Gradients – assign random gradients to each point.
3. Calculate dot products – for each position in the grid, calculate the dot product between position vector and corresponding gradient vectors.
4. Interpolation – interpolate the dot products of all points surrounding each position in the grid. This step generates the result of PN algorithm.

Figure 3.3 Perlin's famous vase, which used PN to generate it's texture. [11]

3.2.2 Diamond-Square algorithm

Another approach used for generating realistically looking terrain is Diamond-Square algorithm (DSA) which comes from a Fractal noise generation family. Fractals (see figure 3.4) are sets of similar-looking patterns usually created by repeated iterations or recursive functions. The DSA is specific method which uses iterative process where the diamond and square steps are applied to smaller and smaller areas. This process then creates fractal landscapes, which can be then additionally processed to create even more realistic looking environment.

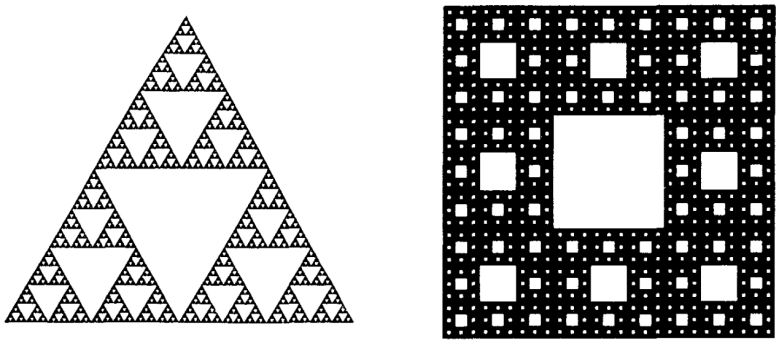

Figure 3.4 "*(c) the Sierpinski triangle or gasket (dimension log 3/log 2=1.585), (d) the Sierpinski carpet (dimension log 8/log 3=1.893)*"[12]

The DSA used in heightmap generation can be described in the following steps:

1. Initialization – create a 2D grid of vertices with initial value of corner vertices which is randomly picked.
2. Diamond step – perform a diamond operation on each initialized square in the grid consisting of four corner vertices. For these squares calculate the average value of their four corner vertices with a random offset added to it. The result is assigned to a vertex in the middle of a square.
3. Square step – for each diamond, assign the average value of its corner vertices with added random offset to it.
4. Repeate – Repeat steps 2 and 3 until desired environment is formed. Each time reduce the size of the squares and diamonds, this reduction usually means halving the previous size.

3.3 Multi-agent system

The concept of Multi-agent system (MAS) was introduced as a possible solution to complex problems where inventing and designing mathematical equations were near to impossible. MAS is based on an idea of distributed intelligence, where a collection of autonomous (individual) agents are places into shared environment. By interacting with each other and their environment, they try to achieve certain goals or objectives. A definition of a goal can greatly vary, from simple survival to solution to complex problems.

3.3.1 Agent modeling and representation

Agent represents an individual who is able to interact and cooperate with other agents, as well as interact with the environment, where he is located. By the definition of Russel S. and Norvig

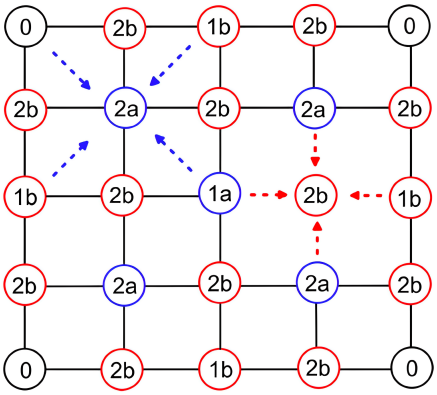

Figure 3.5 "*Diamond-square algorithm (order is 0, 1a, 1b, 2a, 2b, . . .).*"[13]

P., "*An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.*"[14]

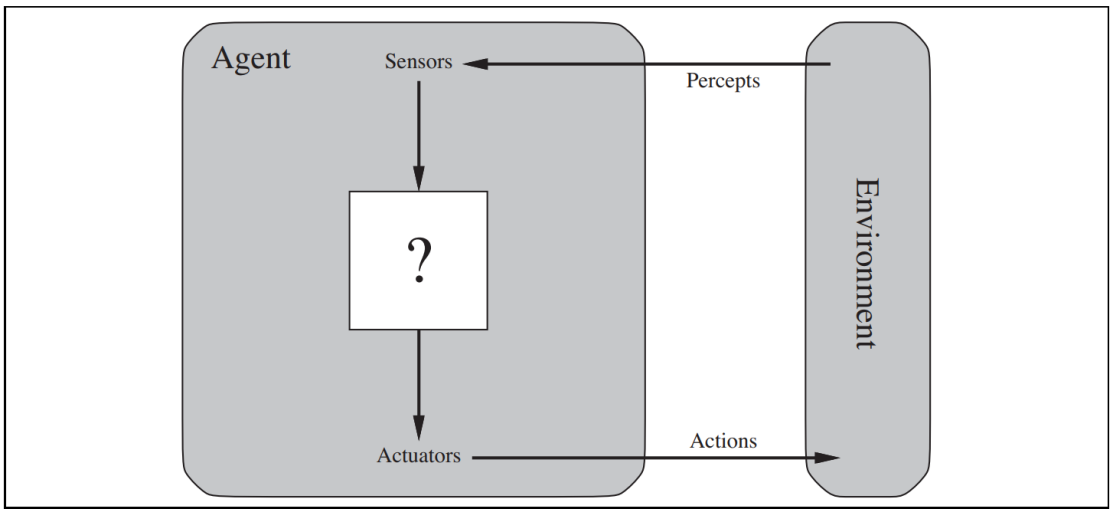

Figure 3.6 Agents interact with an environment through sensors and actuators. [14]

As shown in the figure 3.6, agent perceive his environment through sensors, giving agent a set of sensory inputs. Depending on the chosen type of behaviour and information processing, agent then decides what action he will take. This behavior is defined by the behavioral function b 3.1 that maps sensory inputs to actions.

$$b : I \mapsto A \tag{3.1}$$

There are several methods which implement the agent behaviour. One of them is Goal-oriented behaviour, where agents are introduced to objectives, which they try to achieve. Each step towards the objective rewards agent with utility score or points. Each step in the wrong direction is penalized. This causes agents to learn from the feedback of their actions.

3.3.2 Environment

Environment presents the goals and objectives agents has to face or reach, it can be divided into many categories:

- Fully observable vs. partially observable – depends on whether agents have complete and correct information about the state of the world.
- Deterministic vs. stochastic – actions have fully defined or random effects.
- Static vs. dynamic – the environment changes independently of the agent.
- Discrete vs. continuous –
- Single-agent vs. multi-agent – the behavior of one agent affects the behavior of other agents.

3.4 Evolutionary computation

Evolutionary computation is greatly inspired by work of Charles Darwin – "On the Origin of Species by Means of Natural Selection"[15]. With this book, Charles Darwin introduced a process of evolution of species by natural selection. The idea behind is that, over very long period of time, a species change as they adapt to the environment. If this change is beneficial, a kind of a living organism survives and can introduce another change to it's genetic code. If this change affects the kind negatively, it eventually dies out. To tell whether the change is beneficial or not we often use fitness score or fitness function. This is derived from the concept of how well a species achieved their goal of surviving in the environment.

For example, in "Journal of Researches into the Geology and Natural History of the Various Countries visited by H. M. S. Beagle"[16], Darwin mentions his visit on Galapagos Islands in the Pacific Ocean, were he found various species of finches (which belonged to the genus Geospiza). Each species differentiated by the shape of their beaks (see figure 3.7). Darwin concluded that this was caused by evolutionary processes as a reaction to the different types of food each species consumed as each island offered different kind.

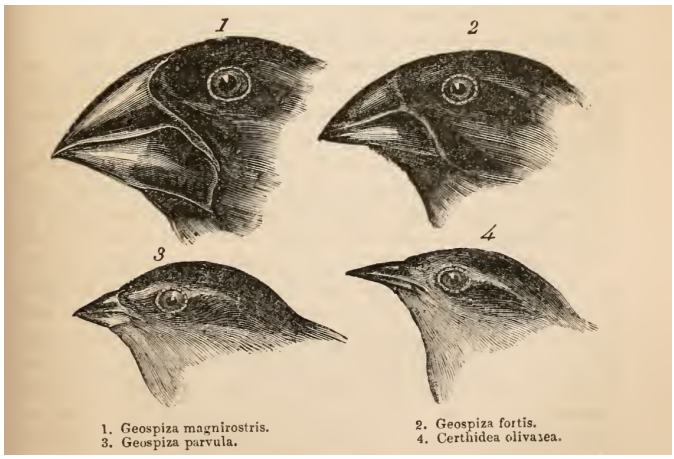

Figure 3.7 Illustration of different species of Geospiza and comparison of their beaks. [16]

Now it was no surprise that this phenomena also inspired computer science. We could see that nature always finds a way and if this was one of the solutions, it clearly has potential.

The fundamental metaphor of evolutionary computing relates this powerful natural evolution to a particular style of problem solving – that of trial-and-error.[17]

As mentioned by A.E. Eiben and J.E. Smith [17], the process of natural selection and evolution can be applied in computer science too. We can represent the problem as an environment, the candidate solution to the problem as individuals that are trying to survive in the environment or achieve any other goal presented to them, and a quality of the solution as the fitness of these individuals.

This kind of problem solving approach is especially useful in optimization. In optimization problems, we know what the desired output of the model, we also know the solution domain (all possible solutions) and the input of the model. Evolutionary computation can find the best or near-best solution from the solution domain by applying the evolutionary processes with natural selection.

3.4.1 Genetic algorithms

Genetic algorithms (GA) is a subset of Evolutionary computation. The main focus for GA is genetic information and it's evolution over time. GA generally follow these steps:

1. Initialization – a population of randomly chosen solutions is generated. Each individual represents one potential solution to the problem.

2. Evaluation of fitness function or fitness score – each individual is tested how well he solves the problem. Based on this the algorithm assigns a score to the individual.

3. Selection – based on fitness score, individuals with higher values are selected for continuation and/or reproduction. The strategy for selection can vary, from simply selection the best half of individuals to assigning a chance to each individual where higher fitness score means higher chance to be selected. This strategy tries to prevent population from reaching local optimum/local maximum. This phenomena prevents individuals to ever reach the best solution as doing so would require individuals to temporally "take worse steps" in regards of evolution.

4. Reproduction and replacement – selected individuals in previous step are staged for reproduction. Reproduction process uses genetic operators such as crossover and mutation in order to create offspring with different genetic information. Crossover operator combines genetic information of parents and mutation operator introduces randomness into offspring's genome. This randomness servers as a form of exploration. Nothing guarantees that mutated genome will prosper however just combining solutions over and over with crossover operator would often result in omission of a vast amount of possible solutions (individuals with different genomes).

   The offsprings then replace the least fit individuals, maintaining a constant or growing population size.

5. Repetition and termination – the steps 2 - 4 then repeat itself until desired goal is reached, a criteria is met, or populations stops evolving.

This procedure can be applied to numerous problem domains, regarding function optimization, robotics, machine learning (by enhancing ML algorithms like hyperparameter tuning or evolution of used neural networks), engineering (by "testing" designed structures and their performance, stability and so on), or image processing by developing the best-performing sequence of filters and other image manipulation techniques, that is interpreted as sequence of genetic information.

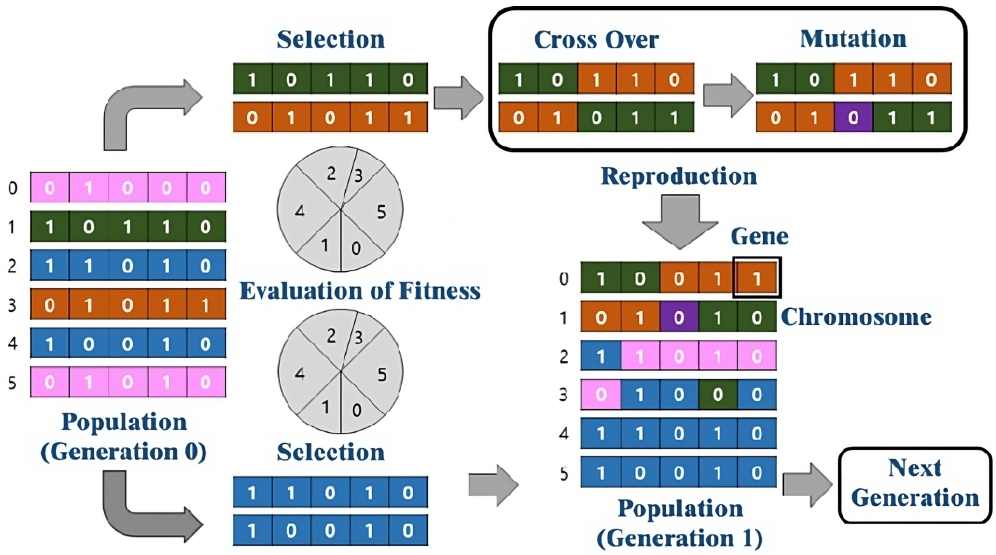

Figure 3.8 Illustration of genetic selection method of GA. [18].

Chapter 4
Simulation

This chapter provides an in-depth description of the approaches used in the thesis program for simulating artificial life and their implementation. Additionally, it offers the reader a general sense of how to expand and operate the simulator.

Simulation is based on an idea of multi-agent system (MAS) (section 3.3), where agents are placed into environment, which is divided into cells, taking inspiration from cellular automata (CA) (section **??**). Both subjects are described in referenced chapters.

When it comes to implementing such simulation it was essential to choose a suitable programming language for the simulator that both supports multi-threading and is also generally fast. Programming languages such as C++ or Python are a popular choice for machine learning and overall scientific purposes, however C++ is faster compared to Python [19]. Therefore, the simulator is written in C++.

The simulation is divided into two programs because the C++ programming language does not provide sufficient flexibility for modularity. The first program, CFLoader, is responsible for importing modules into the simulator by rewriting a portion of its code. The second program is the simulator itself, consisting of two major parts – the environment and agents. Other parts of the simulator are shown in domain diagram in figure 4.1.

These parts could be labeled as "general helping tools" since their main purpose is to either import custom user data or, as previously stated, assist with programming and maintaining organization within the code.

Last important part of the program are Scenarios. Scenario servers as experimenting and statistical tool, initializing populations of specified number of agents with given genome and keeping track of how long each population survives.

4.1 Environment

The environment is a critical part of artificial life simulation. Apart from letting agents interact with each other in it, environment can present goals and barriers to the agents. It can affect their behaviour, provide the necessary food and resources they need, and even represent a threat by its harsh nature, lowering the number of unsuitable agents. All this can shape their evolutionary processes, bringing another diversity into simulation.

There are different options for representing the environment. First option is whether to choose 2D or 3D world. While a 3D world may come closer to reality, it is not always necessary. In fact, in 2D world there is a plenty of room to experiment with any types of actions agents can make. However, a 2D representation may not be suitable for all simulation models. For example,

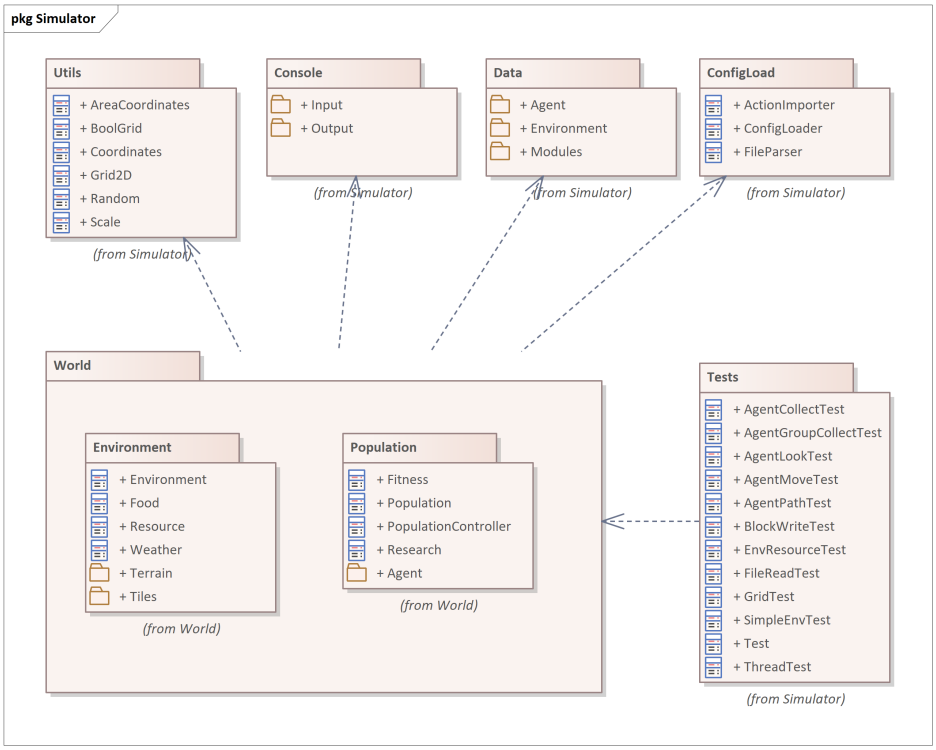

Figure 4.1 Package model of simulator

a 2D world would lack spacial information for a autonomous driving simulation, as it requires not only how the road looks like, but also what signs are near it.

Second option is whether the world should be discrete or continuous. A discrete world is represented by a grid of smaller areas, in which agents can separately or together live. This can make it easier to define, how agents can move, and implement actions. On the other hand, a non-discrete world is represented via continuous space. This approach is crucial for accurate physics simulators like climate modeling or fluid dynamics simulator. Although their represent nature more accurately, those simulations require a lot of computational power and are hard to implement. Overall it all depends on which kind of simulation is being created and what are its goals.

In this simulation the environment is represented as 2D tile-based grid, since the speed of the world generation needs to be very high for this thesis goals. This will allow to create thousands of worlds with different scenarios in a short period of time. Each tile has a few properties. Firstly, since the environment needs to interact with agents (and *vice versa*), it contains information about the agents currently occupying the tile. Secondly, the environment consists of two types of tiles – Land and Water – giving the simulation a more natural look. Moreover, tiles also have their own attributes derived from randomly selected biome, such as temperature, terrain type, which can affect agents ability to move, color, and other properties. Lastly, the tile class inherits from an upper abstract class called “TimeDependent”, forcing it to implement a virtual method “progressTime”. This ensures that the environment changes over time, as required in the thesis instructions.

4.1.1 Terrain generation

For tile-based environment there are several methods that could be used. Given that this thesis aims for mainly fast methods with acceptable portion of realism, i have chosen approach with cellular automata **??**.

4.1.1.1 Noise generation

CA is based on randomly generated noise, in this thesis it is a noise of Water and Land tiles. In terms of memory and speed efficiency, the Water tile is represented as *false* value and the Land tile as *true* value. This optimization ensures that smaller objects are being allocated to heap memory resulting in much faster noise generation.

Noise could be created by randomly generated numbers for each tile in the grid and then assigning a true of false value whether the number exceeds a given threshold. Setting and tweaking a threshold number in noise generation also changes the density of the generated environment. However generating millions of random numbers is inefficient and time-consuming.

Instead, the world generator in this thesis uses individual bits of randomly generated number. Each bit can be directly converted to a true (1) or false (0) value without any additional computation. The density of the world is directed by the bit ratio – how many 1s are in the whole grid compared to all bits in the grid. We can control this ratio by applying bitwise OR or AND operations between the current candidate for the boolean grid and newly generated one. From now on, assume that this grid of 1s and 0s is converted into one long stream of bits making a one large number. Because program uses Mersenne Twister 19937 generator (MT19937)[20], simply repetitively trying bitwise OR or AND operations between two randomly generated numbers in order to get desired 1-bit density would sometimes take a very long time. This effect is caused by the fact, that MT19937 generates 32-bit integers that are uniformly distributed over the range of possible values. It is necessary to firstly generate a second number with slightly higher or lower summary of 1 bits. Only after that we can execute bitwise OR or AND operation with the original number.

Proof. Let us have two numbers a and b represented as sequence of n bits and generated by MT19937. MT19937 generates numbers uniformly distributed, meaning every number in range has a same chance of being generated. This implies that each bit in a and b has the same probability of being either 0 or 1. We can define a Bernoulli random variable X that takes on the value 1 if the generated bit in number is 1, and 0 if it is 0. The probability distribution of X is given by:

$$\begin{aligned} P(X = 1) &= P(bit = 1) = 0.5 \\ P(X = 0) &= P(bit = 0) = 1 - P(bit = 1) = 0.5 \end{aligned} \tag{4.1}$$

From this, we can calculate the expected value of X (this only applies for Bernoulli distribution):

$$\begin{aligned} E(X) &= p \\ E(X) &= P(X = 1) \\ E(X) &= 0.5 \end{aligned} \tag{4.2}$$

Now let us define another variable Y, that represents the summary of 1 bits in sequence of n bits, defining a random number. We can express Y as a summary of n variables X:

$$Y = X_1 + X_2 + \cdots + X_n \tag{4.3}$$

where X_i are independent variables representing individual bits. Finally, we can calculate the expected value of Y, since we know from 4.2 that expected value of X_i is 0.5:

$$\begin{aligned} E(Y) &= E(X_1 + X_2 + \cdots + X_n) \\ &= E(X_1) + E(X_2) + \cdots + E(X_n) \\ &= 0.5 + 0.5 + \cdots + 0.5 \\ &= n/2 \end{aligned} \tag{4.4}$$

Therefore approximately half of the bits in random number consisting of n bits generated by MT19973 have value of 1. ◀

From this, we can deduce that executing bitwise AND operation between two randomly generated numbers will reduce the number of 1 bits approximately by half.

Analogically, we can deduce that executing bitwise OR operation between two randomly generated numbers will increase the number of 1 bits approximately by half.

In implementation, the density of 1s in the main number (representing the boolean grid) is increased by executing bitwise OR operation with another number, that is beforehand modified with bitwise AND and after that one more time with bitwise AND operation, rising the number of 1s in the main number by ~12.5 %. Same goes analogically for lowering the number of 1s.

This process repeats until desired density is reached with 5 % tolerance.

4.1.1.2 Terrain and biomes

Since the noise is created, we can start shaping the terrain with Cellular automata. Let us assume that the generated grid of true and false values is a matrix of 1s (true) and 0s (false). Now for each tile we apply the following formula 4.5:

$$M = \sum_{n \in N} (n - 0.5) \tag{4.5}$$

where N stands for all tiles in neighborhood (the original tile included), which is defined as Moore neighborhood (see figure 4.2) of radius $r = 1$ and where M stand for majority of tiles in neighborhood.

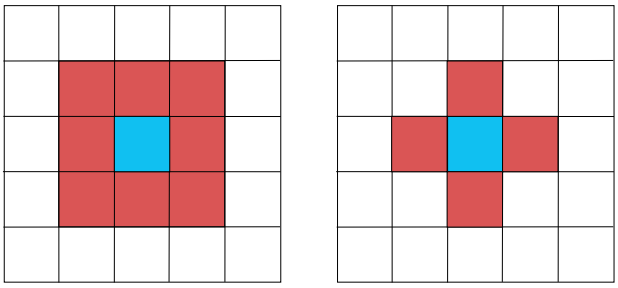

Figure 4.2 Moore neighborhood (left) and Von Neumann neighborhood (right)

After that we assign a value to the iterated tile T based on 3 rules:

- if $M < 0$, then T will be a Water tile
- if $M > 0$, then T will be a Land tile
- if $M = 0$, then T will have a random type assigned

This process is then repeated 7 times, making the landscape look more smooth.

Biomes are defining feature of the environment. They define a tiles temperature, distribution of food and resources and other properties of the world. After a landscape is generated, the environment starts to generate biomes. This can be achieved by several ways. First, the biomes could be assigned after introducing another property to the world, for example temperature, making areas with higher temperature desserts or with lower temperature mountains or frost biomes. Another example would be spreading vegetation across the world and assigning biomes to the areas based on its density. This thesis implements a method, where based on program parameter *biomeChunkSize* the simulator splits the environment into smaller areas in which it randomly chooses a tile and assigns it a random biome. After every area has *biome seed*, each seed then spreads, imitating a growing cells. This process is implemented in following steps:

1. Add *seed tile* to a *opened* buffer.
2. For each opened tile in *opened* buffer, add each tile in Von Neumann neighborhood [21] (see figure 4.2) to a *candidate* buffer.
3. Clear *opened* buffer.
4. Merge *postponed* buffer into *candidate* buffer.
5. Choose a random tile from *candidate* buffer and mark it as a *reference point* (RP).
6. Sort tiles in *candidate* buffer by Euclidean distance [22] from the RP.
7. Move half of the closest tiles to RP from *candidate* to *postponed* buffer.
8. Move the rest of the tiles in *candidate* to *opened* buffer.
9. Repeat steps 2 - 8 until *opened* buffer is empty.

Thanks to this process biomes can grow in any random directions, not just in Up, Down, Left, and Right directions.

4.2 Agents

Agents are second critical part of artificial life simulation. They are individual entities trying to accomplish either common or personal goal. In this thesis, the goal is to survive as long as possible.

4.2.1 Agent properties

Agent properties are divided into 3 parts. First one is the current state of the agent, keeping track of its energy, health and hunger. Second one are *weights*. For each available action agent assigns a weight of how important that actions is. This can change over time in agents life. Last part is **genome**, the most important part of the agent. Genome defines agents traits, which influence its actions. For example *endurance* trait with high value will enable agent to travel long distances without consuming much energy. Another example is *vision* trait, allowing agent to see further distance, spotting food and resources sooner than other agents.

4.2.2 Population

Population consists of group of agents with the same starting genome. this hierarchy makes it easier to implement interaction between more than two agents however this is not yet fully implemented in this thesis. Population entity keeps track of its agents and also servers as a mediator between evolutionary processes and agents. Whenever a new generation should be created, population entity reports to the population controller how well each agent performed and whether he is suitable for reproduction or elimination.

There are actually two ways on how agent can reproduce. First option is, as mentioned earlier, through population controller. This is described in more detail in following section 4.2.3. Second option is via reproduction action. This action is called whenever an agent has more food than he needs. It is possible to implement all kinds of reproductive actions, but in this thesis only simple clone was added.

4.2.3 Life cycle

This section describes agents life, behaviour, reproduction, and genome propagation.

4.2.3.1 Initialization

At the start the simulator initialize n populations of m agents with given genomes. Program places each population on randomly selected Land-type tile and then starts the simulation. Everything apart from the initial position of population can be defined by user.

The simulation runs in discrete time, meaning every action (not only of agents) have specified how many time iterations it takes to execute it. For example apple can regrow on tile every 3 iterations or agent can move from one tile to another in 5 iterations. Running simulation in continuous time wouldn't bring any advantages apart from more flexibility in time duration of actions. However this advantage is over weighted by the cost of implementing it.

4.2.3.2 Action choosing

Deciding on what action should agent choose can be implement in many ways. That is why agent only calls one method to tell him, what to do. This can be easily substituted with different implementation, for example agent could choose their actions with neural networks or planners and schedulers. The current solution will probably be replaced with other mentioned options. For this work agent chooses actions based on their score which is computed with custom action rules in mind. The score is then multiplied by associated action weight. Agents can also choose random action with positive score whenever a discovery parameter is set to more then 0.

4.2.3.3 Action execution

Each action have access to all agents properties, especially traits. Usually, time for executing specific action is derived from agents traits. Whenever agent chooses an action, he sets himself for a period of execution time into "sleep" and waits until action is finished. Action on its last "time iteration" proceeds to do what it is designed for. Then agent can repeat the process of choosing and executing other actions.

4.2.3.4 Selection

Each generation (duration specified by user, default value is 50 time iterations) only the most *fit* agents are selected to continue with their life and staged for reproduction. The rest of them are

eliminated. Deciding which agents should continue is done by *fitness function*. One example of fitness function looks like this:

$$f = \sum_{i=1}^{3}(w_i * s_i) \tag{4.6}$$

where each agent state s_i (energy, hunger, health) has its assigned weight w_i.

Simulation in this thesis also provides an option not to choose fitness function to stage, which agents should continue and reproduce. There is also an option to eliminate agents whenever their hunger or health reaches critical value. This can be set with program argument "-F <health/hunger/fitness>".

4.2.3.5 Reproduction

After selecting the most *fit* agents, we have to define which of them will be reproduced and how. General approach is to reproduce all *fit* agents, which happens here too. The crossbreeding process randomly selects half of parent A traits and half of the parent B traits, creating an offspring with new type of gene. There is also 5% chance for mutation for every trait in the offsprings gene. This process replaces the selected trait value with new random value.

4.3 Implementation

As mentioned earlier, environment is implemented as a 2D grid of tiles. Environment holds information about Weather, Biomes, available actions and the base grid of tiles. Each tile has a a following properties:

- a vector of pointers to agents that are currently occupying it,
- a pair of numbers representing a position of the tile in the grid,
- a reference to the Environment instance that the tile is in,
- a pointer to a leading population (representing a property assigned to one population),
- a pointer to assigned biome,
- information about temperature and terrain influence on agent's mobility,
- vectors of food and resource,
- and lastly vector of prescription of food and resources (from which the food entities are generated)

During world generation, tiles undergo a conversion process from individual bits to boolean values, then to enumeration values, and finally to instances of child classes that implement the abstract class *Tile*.

Weather in this thesis is not implemented, however is ready for additional implementation by user or me in the future. The idea behind this is same as for *Behaviour* module, or *Tile* implementation, where program has prescribed the header files which are separated from implementation files. The program should work without knowing which implementation a user has chosen.

On the other hand, Biomes are defined in configuration files. This means that user can change any aspect of the file without any required knowledge of C++. Biomes define the shape of the environment, it's hostility, food, food distribution, resources and their distribution. They also define the temperature and other aspects affecting agents.

The second part of the simulations is agents, who are/can be controlled by population entity, which is controlled by *PopulationController* entity, managing population creation and deletion. It also propagates the information about an advancement in time, sending signals randomly to all the agents in the environment. *PopulationController* also defines, how fitness function is evaluated for each population and how are agents selected for reproduction and termination. The *Fitness* entity (in *PopulationController*) always has to provide functions for selection and fitness score, however the implementation of these functions can be completely substituted to the user's liking.

Agent solely is implemented as a hybrid regarding a micro-model and macro-model design. It can act either in the best interest of the entire population in which it is placed or in it's own self-interest. Agent also inherits *TimeDependent* class (as all entities that are dependent on time, like tiles, food, resources, weather, actions, scenarios, or *Population* and *PopulationController* entity), forcing him to do something every iteration of simulation. Agents properties are defined as follows:

- a module for agent's stats – energy, health, and hunger,
- a pointer to behaviour module defining the decision-making process upon receiving information through sensors
- a sensory inputs (as updating it each iteration would cost a lot of computational power, hence it is stored here and updated only if it is convenient),
- a reference to Environment entity, this way agent can have an impact on the environment and it is easier to exchange information between these two,
- a **Genome**, defining traits of an agent,
- an ID,
- a population ID, defining who agent belongs to,
- a pointer to a tile the agent stands on,
- a pointer to a tile which has been picked as agent's home,
- a map of food and resources agent has in his inventory,
- a pointer to an action that is currently in progress,
- an information about whether an action is finished and is staged for deletion,
- and lastly vector of weights used for evaluating next action.

Users can add their own actions into the simulation by following the class specifications:

```
class Action : public TimeDependent{
protected:
    Agent * agent = nullptr;
    virtual unsigned int timeToPerform() = 0;
    void done() const;
public:
    unsigned int timeRemaining = 0;
    std::string name;

    explicit Action(Agent * agent);
    virtual ~Action();

    void start();
    virtual double getScore(Agent * agent) const = 0;
    virtual double energyDrain();
    virtual Action * copy(Agent * agent) const = 0;
};
```

Code listing 1 Class specifications of action

Chapter 5

Experiments and results

Before doing any experiments, it is worth mentioning how was the environment set and how agents behave in these tests. The environment, for most of the tests, is 64 tiles wide and 64 tiles long (4096 in total). The land density was set to 60 % and food density to 20 % with regenerative rate of ~1. This means the current environment produces around 500 food per generation. In practice, the terrain generation changed land/water ratio to around 80 %, resulting in food production around 700 per generation.

As for the agent behaviour, the goal for each agent is to bring any kind of food to home in g iterations. If agent brings 1 food, he survives for another generation. If he brings 2 and more food, he can reproduce. Lastly, if he brings no food or does not come home at all, he will be eliminated.

At the start, agent tries to find food, then takes it/harvests it, and then he can decide, whether he has enough time and energy to look for another or whether he has to go home. This decision is also based on the distance from home.

The first experiment only tests if single population of agents can survive without crossbreeding or genome mutation for 100 generations. This test also shows how many agents the current environment can support without any competition between different populations or agents with different genome. Food was distributed in two different ways. First one was "cumulative", each generation adding food to the environment. In second experiment food amount is reset to randomly chosen number in each generation, meaning if there was tile with food at the end of one generation, the food was deleted.

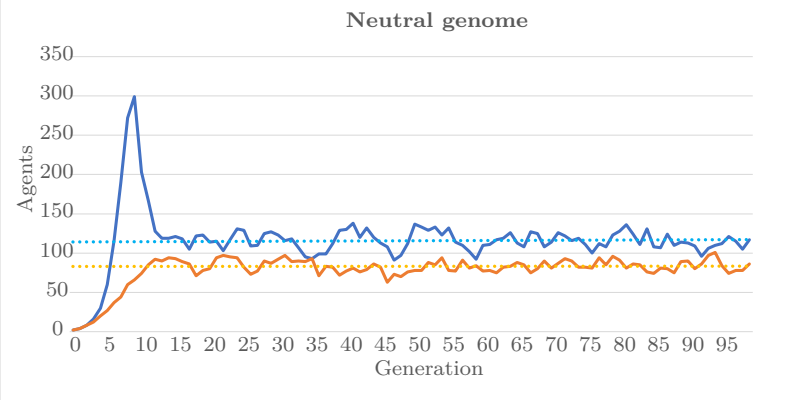

Figure 5.1 Normal agents with no mutation. Blue curve – cumulative food distribution; Orange curve – food distribution always set back to certain value.

The results in figure 5.1 show that after few generations the number of agents stops growing, as it reaches a maximum capacity, where agents do not have enough food to sustain the current population. The spike in the beginning of the experiment is most likely caused by accumulated food in the area far off the population base (where agents bring food). As the population grew, agents had to travel more further away, where food was accumulated in the meantime. Once they reached the travel limit, as they did not have enough time or energy, the only food, which was available to agents at that moment, is the one that regenerated after one generation.

All the following experiments use non-cumulative food distribution for more balanced results.

5.1 One-trait genome mutation

Next experiment focuses on "speed" trait in agent's genome and it's evolution over time. This trait reduces the time needed for any movement-related action, such as "GoFromHome" (forcing agent to move in a direction from home so he does not wonder around in circles), "GoToFood" (whenever there is any kind of food in agent's vision range) or "GoHome" (whenever agent is done with finding food). This advantage comes with a cost of much greater energy usage, following the formula:

$$E = \frac{s^2}{d} + c \tag{5.1}$$

where E stands for final energy cost of moving 1 tile, s for the value of *speed* trait, d for a movement energy divisor, which serves as a scaling factor, and c for any constant defined by user.

If agent brings 2 pieces of food to home, he is staged for reproduction. He needs another agent to combine his genetic information. If there is no second agent, he only survives for another generation. When agent finds other agent to reproduce with, they create 2 offsprings. Both of them have genome composed of traits of their parents with equal chance that certain trait comes from either parent A or parent B. However one offspring also has a chance, that the value of any trait does not come directly from any of the parents but is rather derived from one of them and slightly modified in a random way. This type of randomness introduces *mutation* to the simulation.

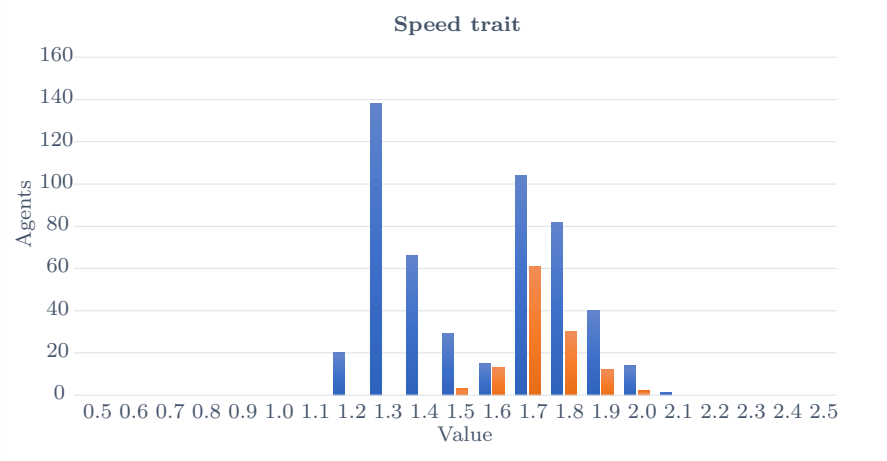

Figure 5.2 Agents with *speed* trait by it's value. Orange – normal amount of food; Blue – 3 times the amount of food.

Reproduction for all the following experiments is population-closed (if not stated otherwise), meaning agents from one population do not reproduce with agents from another population.

Because the chance to get a different value for *speed* trait in agent's genome is not that great, evolution would not have enough time to make any significant difference. For this reason, the number of generations was risen to 500. After some tests this number came out as a good compromise between accuracy and simulation execution time.

First test regarding one trait evolution (figure 5.2 – orange) showed that agents with higher speed did win the "evolution race" compared to the standard agents with speed trait value of 1. But, interesting things happen if we add more food to the environment. By tripling the amount of food, suddenly simply being faster than others is not the best strategy and being efficient with energy overweights the advantage of speed (figure 5.2 – blue). This phenomena is caused by the fact, that having a speed trait with higher value is inefficient for longer distance. Agent with this trait has to sleep more often so competing for food in far away distances results in less to none collected. But even if agent is efficient with his movement, when he tries to find food far from home, there is no time to scout the area for it. This means that when there is a very limited amount of food, the competition for it in local area is fierce and speed becomes crucial for survival. However when there is enough food and agents do not have to look for it for a long period of time, populations splits into 2 half. One that quickly collects food near home and one that slowly collects food in greater distances.

By analyzing the results more deeply, we can see a trend that agents tend to follow (figure 5.3). Agents discovered that their time of executing a movement-related actions is rounded down (due to a discrete nature of the simulation). This means values that bring the final time to a number approaching the next whole number $(x + 1)$ (before rounding it down), for example $x.8$ or $x.9$, present a highly efficient ratio between energy cost and speed.

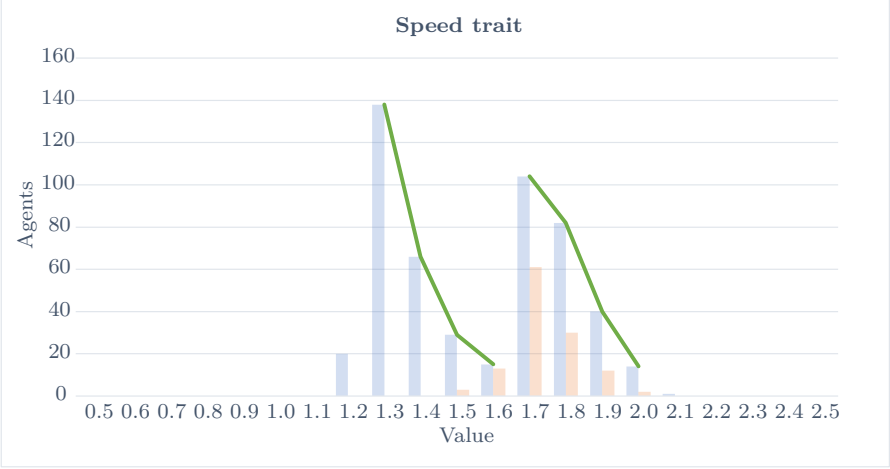

Figure 5.3 Emergent trend of speed values that take advantage of rounding down the time of action.

Now let me introduce another trait to the simulation – *bravery*. This trait represents the amount of risk an agent is willing to take. It controls how much agent prefers finding more than one food over coming home safely (in time). For this experiment there are couple options how to test which value for *bravery* trait works the best. We can simulate the difference between two environments, one with limited food and one with plenty of food. We can also test how brave are fast agents and how brave are slow agents. Ultimately we can test if agents with best speed trait value behave compared to others. For example, as the results in figure 5.2 show, agents with genome containing speed trait of value 1.3 and 1.7 make the majority of the population. This can also effect the bravery trait evolution.

So for the experiment i have decided to make 4 tests in total. First two tests simulate the difference between fast and slow agents in an ideal environment with a lot of food (about 70 % of land is covered with it). The prediction for slow agents would be that the value of bravery trait will rise almost to the maximum because it is in their nature to cover more distance further away from home. Fast agents are more of the opposite in terms of distance from home, so the value of bravery trait should not change much. However quite the opposite happened.

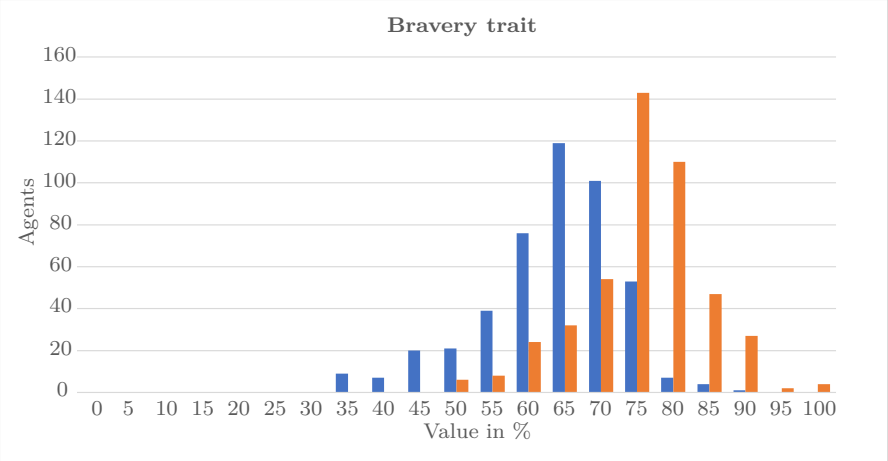

Figure 5.4 Agents with *bravery* trait by it's value. Orange – fast agents; Blue – slow agents.

This test (figure 5.4) showed that the optimal strategy for slow agents is not to risk that much. Afterall their speed is not great and because their area of interest is usually far from home, knowing when to stop collecting food is a good predisposition for their survival. On the other hand, fast agents tend to collect food in close proximity of their home. And since the distance is also part of decision-making process behind action "GoHome", bravery affects this score in a way that distance does no longer bother fast agents.

The other two tests involve agents that have the most efficient value of the speed trait. These agents are placed into environment with plenty of food and then they encounter it's shortage. The previous test suggest that agent will be more brave and with their advantage of being efficient, they will not misjudge the distance and time needed for the way home.

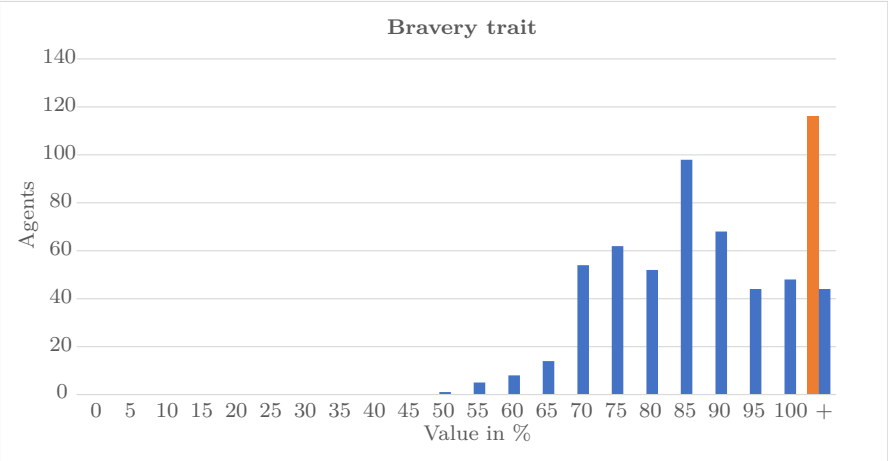

Figure 5.5 Agents with *bravery* trait by it's value. Orange – normal amount of food; Blue – 3 times the amount of food.

However, the results in figure 5.5 are more unexpected. Apart from confirming that agents have higher values of bravery in environment with sufficient food, when it comes to the environment with less food, agents suddenly choose very aggressive strategy and forbid themselves to come home with less then 2 pieces of food completely. This behaviour eliminates the possibility of an agent "just surviving" through another generation and forces him to only reproduce.

5.2 Two-trait genome mutation

Before letting two different populations compete with each other, we can test multiple trait evolution over time simultaneously. Since any more traits than two would be hard to picture, this experiment tests exactly that. The simulation provides an option to include any number of traits into agent's genome, but doing so would require more specific application than what is of this thesis.

The environment was chosen to have limited food, pressuring agents into faster adaptation and more aggressive strategy. This can show how far an agent will go with the speed trait value, how much is he willing to sacrifice energy and how desperate he is to find enough food to replicate.

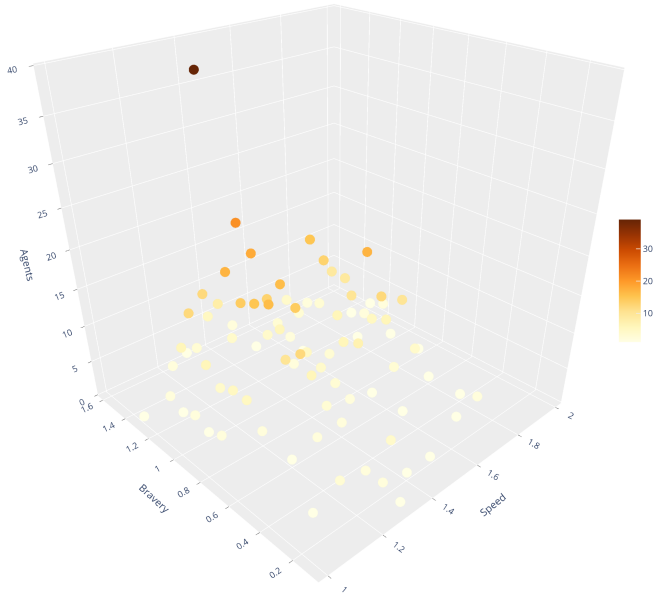

Figure 5.6 Bravery and Speed trait evolution

Here the results are not that surprising. As before, bravery pays off greatly, the majority of agents are beyond bravery value 1 – again ignoring an option to come home with only one piece of food. As for the speed value, this time carefully handling energy usage won over saving time during action execution.

Idea for other traits could be *vision*, affecting how far can agent see or even what can agent see. For example agent with great vision could also see other agent's inventory or his energy, health and so on. Other traits could even define basic economical interactions, like dividing agent's into buyers and sellers or, when cooperating, into producers and sellers. In prepared scenarios agents already have defined additional traits like mentioned *vision* or *endurance*.

5.3 Two competing populations

Up until now, experiments tested only one population at a time, letting agent's collectively evolve one or more traits. In this experiment, agents have fixed genome, but are divided into two populations. Apart from setting the food amount in the environment to higher value, the rest stays the same.

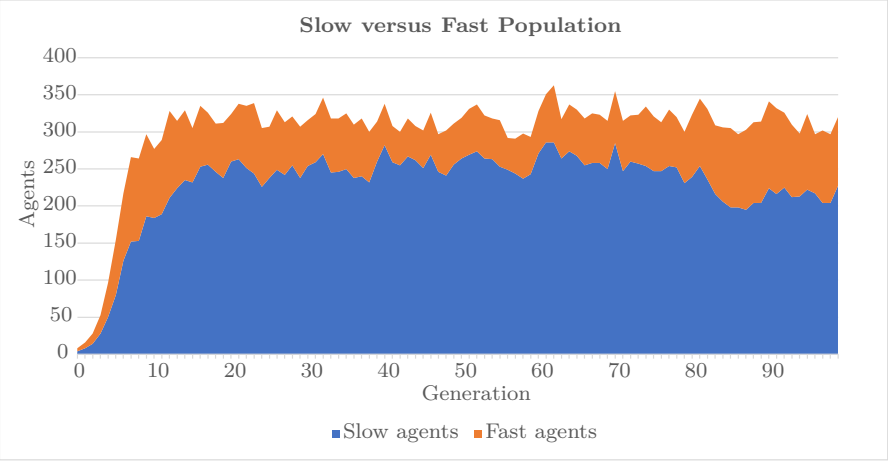

Figure 5.7 Agents with *speed* trait of value of 1 (Orange) and 2 (Blue) competing against each other.

Here unfortunately nothing interesting happened. The population ratio mimicked the previous tests (see figure 5.2) where one population was split into fast and slow agents, also with majority of slow agents.

5.4 Agent interaction – aggression trait

Following experiments introduces first interaction between two agents. This test implements an *aggression* trait.

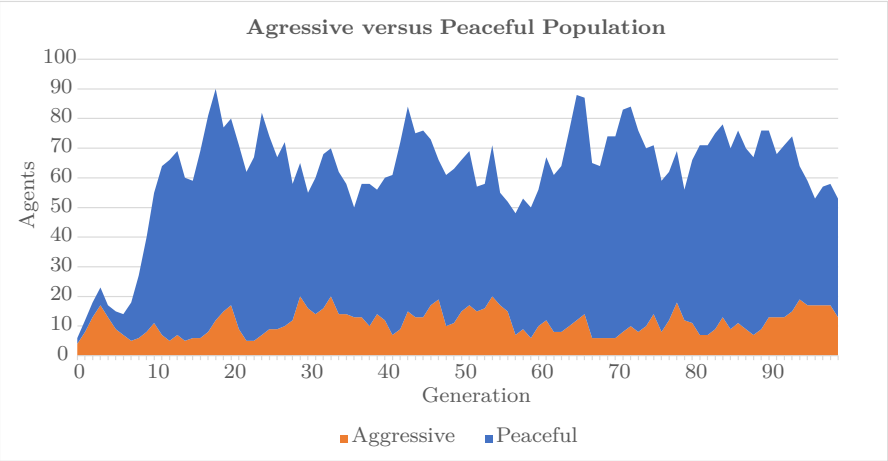

Figure 5.8 Two competing populations of agents, one with *aggression* trait, one without it.

Whenever agent has this trait, he is not strictly fixed to strategy "Do not come home without any food" as agents who are peaceful. In fact, if he does not find any food in a certain amount

of time, agent returns home and tries to rob other agents. This trait comes with a disadvantage: if an agent tries to rob other agent, who also has an *aggression* trait in his genome, they fight each other and both of them are eventually eliminated, since fighting costs a lot of energy and health (after all, it is literally fight for survival).

Graph in figure 5.8 shows that being aggressive does not pay off. This disproportion of agents is caused by one rule peaceful agents follow: "Do not return home until you have at least one piece of food". Aggressive agents give up much sooner and head home with a vision that they acquire food by mugging other agents. However because aggressive agents return home sooner than peaceful agents, they end up mugging each other resulting in their death.

The graph also shows how introducing aggressiveness into simulation affects the total number of agents in one generation. Whenever population of aggressive agents grows, the total amount of agents (not only in the peaceful population) drastically goes down. Promptly after that aggressive population starts "fighting" itself and the total number of agents rises again, even tho the number of agents in aggressive population drops.

5.5 Summary

All these experiments served as a showcase of the flexibility the simulator can provide. From one population to many, from one trait in genome to whatever user want. There is also an option to choose between two types of agent reproduction: one that artificially selects *fit* agents and reproduces a specified portion of them or one where agents reproduces through custom action. Last method was not demonstrated, however when implementing custom actions, it is very easy to do so for anyone with basic knowledge of C++. The reproduction can also be parameterized by user to include (or to not include) mutation, it's chance and it's rate.

The simulator also provides flexibility in world creation. From defining a world size to custom food, resources and whole biomes. This was not shown because without proper user interface it lacks purpose. This however brings an idea for future improvement.

Chapter 6
Conclusion

In my thesis I analyzed the current progress in the field of artificial life research, proposed my own solution to the problem and its implementation.

Although research in this area has already moved on a great deal, it is still such a complex topic that there is still plenty of room for improvement and exploration in some areas of ALife. I have found that there are already many interesting and even extensive simulators and programs dealing with this topic, and with the large variety of methodologies applied to ALife development, they all differ a lot from each other. Each methodology is suitable for solving different problems and therefore it is not possible to create a one-size-fits-all simulator that solves everything.

In my research, I found that the very early days of ALife produced unpredictable and interesting results. An example was the previously mentioned Conway's Game of Life, where cells began to form interesting shapes, forming groups that differed from each other in behavior and shapes that began to replicate. Another example, also already mentioned, was the Avida program, where organisms suddenly developed cooperative behaviour. These programs became a great inspiration in the ALife field and gave rise to many other and more advanced simulators.

In my design for the simulator, I chose a 2D discrete environment represented by a grid of cells. Each cell could then hold all sorts of information, such as resources or its effect on an agent. Agents could then move around this environment in a discrete time and interact with others.

Experiments have shown that agents can respond correctly to their environment and to others, while providing a relatively high speed of execution for these experiments. This can still be improved by adding the ability to run the program on multiple threads, which will make the program several times faster. On the other hand, picking out relevant actions and balancing their effects to make the experiment to have some added value and be unpredictable in some way was very exhausting. After all, the simulator is mainly meant to serve as a platform for various implementations and to make it easy to use.

The program is now in a state where it is very easy to run experiments in it, modify things as needed, implement additional actions and additional biomes, however, during testing I came across a major problem, namely that experiments are very difficult to evaluate without a good graphical interface and/or statistical tools. So this will be another goal in future work on this simulator.

Bibliography

1. GARDNER, Martin. MATHEMATICAL GAMES. *Scientific American* [online]. 1970, vol. 223, no. 4, pp. 120–123 [visited on 2023-05-10]. ISSN 00368733, ISSN 19467087. Available from: http://www.jstor.org/stable/24927642.
2. ILACHINSKI, Andrew. EINSTein: A Multiagent-based Model of Combat. In: *Artificial Life Models in Software*. Ed. by ADAMATZKY, Andrew; KOMOSINSKI, Maciej. London: Springer London, 2005, pp. 143–185. ISBN 978-1-84628-214-0. Available from DOI: 10.1007/1-84628-214-4_7.
3. OFRIA, Charles; WILKE, Claus. Avida: A Software Platform for Research in Computational Evolutionary Biology. *Artificial life*. 2004, vol. 10, pp. 191–229. Available from DOI: 10.1162/106454604773563612.
4. TODD, Peter; MIRANDA, Eduardo. Putting Some (Artificial) Life Into Models of Musical Creativity. 2004.
5. CHERAN, S.C.; GARGANO, G. Computer aided diagnosis for lung CT using artificial life models. In: *Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)*. 2005, pp. 1–4. Available from DOI: 10.1109/SYNASC.2005.28.
6. DORIGO, M.; GAMBARDELLA, L.M. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*. 1997, vol. 1, no. 1, pp. 53–66. Available from DOI: 10.1109/4235.585892.
7. KOMOSINSKI, Maciej; ADAMATZKY, Andrew. *Artificial Life Models in Software*. 2009. ISBN 1848822847. Available from DOI: 10.1007/978-1-84882-285-6.
8. KIM, Kyung-Joong; CHO, Sung-Bae. A Comprehensive Overview of the Applications of Artificial Life. *Artificial Life*. 2006, vol. 12, no. 1, pp. 153–182. Available from DOI: 10.1162/106454606775186455.
9. LONGAY, Steven; RUNIONS, Adam; BOUDON, Frédéric; PRUSINKIEWICZ, and. TreeSketch: Interactive Procedural Modeling of Trees on a Tablet. In: 2012. Available from DOI: 10.2312/SBM/SBM12/107-120.
10. EBERT, David; MUSGRAVE, F.K.; PEACHEY, D.; PERLIN, Ken; WORLEY, Steve; MARK, W.R.; HART, John. *Texturing and Modeling: A Procedural Approach: Third Edition*. 2002.
11. PERLIN, Ken. An image synthesizer. In: *International Conference on Computer Graphics and Interactive Techniques*. 1985.
12. FALCONER, Kenneth J; FALCONER, KJ. *Techniques in fractal geometry*. Vol. 3. Wiley Chichester, 1997.

13. ZHANG, Ling; ZENG, Zhaofa; LI, Jing; LIN, Jingyi; HU, Yingsa; WANG, Xuegang; SUN, Xiaodong. Simulation of the Lunar Regolith and Lunar-Penetrating Radar Data Processing. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*. 2018, vol. PP, pp. 1–9. Available from DOI: 10.1109/JSTARS.2017.2786476.

14. RUSSELL, Stuart; NORVIG, Peter. *Artificial Intelligence: A Modern Approach*. 3rd. USA: Prentice Hall Press, 2009. ISBN 0136042597.

15. DARWIN, Charles. *On the Origin of Species by Means of Natural Selection*. London: Murray, 1859. or the Preservation of Favored Races in the Struggle for Life.

16. DARWIN, Charles. *Journal of Researches into the Geology and Natural History of the Various Countries visited by H. M. S. Beagle*. Cambridge University Press, 2009. Cambridge Library Collection - Darwin, Evolution and Genetics. Available from DOI: 10.1017/CBO9780511693342.

17. EIBEN, A.; SMITH, Jim. *Introduction To Evolutionary Computing*. Vol. 45. 2003. ISBN 978-3-642-07285-7. Available from DOI: 10.1007/978-3-662-05094-1.

18. PARK, Gun; HONG, Ki; YOON, Hyungchul. Vision-Based Structural FE Model Updating Using Genetic Algorithm. *Applied Sciences*. 2021, vol. 11, p. 1622. Available from DOI: 10.3390/app11041622.

19. ZEHRA, Farzeen; JAVED, Maha; KHAN, Darakhshan; PASHA, Maria. *Comparative Analysis of C++ and Python in Terms of Memory and Time*. 2020. Available from DOI: 10.20944/preprints202012.0516.v1.

20. MATSUMOTO, Makoto; NISHIMURA, Takuji. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.* 1998, vol. 8, pp. 3–30.

21. KARI, Jarkko. Theory of cellular automata: A survey. *Theoretical Computer Science*. 2005, vol. 334, no. 1, pp. 5–10. ISSN 0304-3975. Available from DOI: https://doi.org/10.1016/j.tcs.2004.11.021.

22. DEZA, Michel; DEZA, Elena. *Encyclopedia of Distances'*. 2009. Available from DOI: 10.1007/978-3-642-00234-2_19.

Contents of the attached media

- `readme.txt` ..... brief description of the content of the medium
- `src`
  - `impl` ..... source codes of the simulation
  - `thesis` ..... source code of the thesis in format LaTeX
- `text` ..... text of the thesis
  - `thesis.pdf` ..... text of the thesis in format PDF