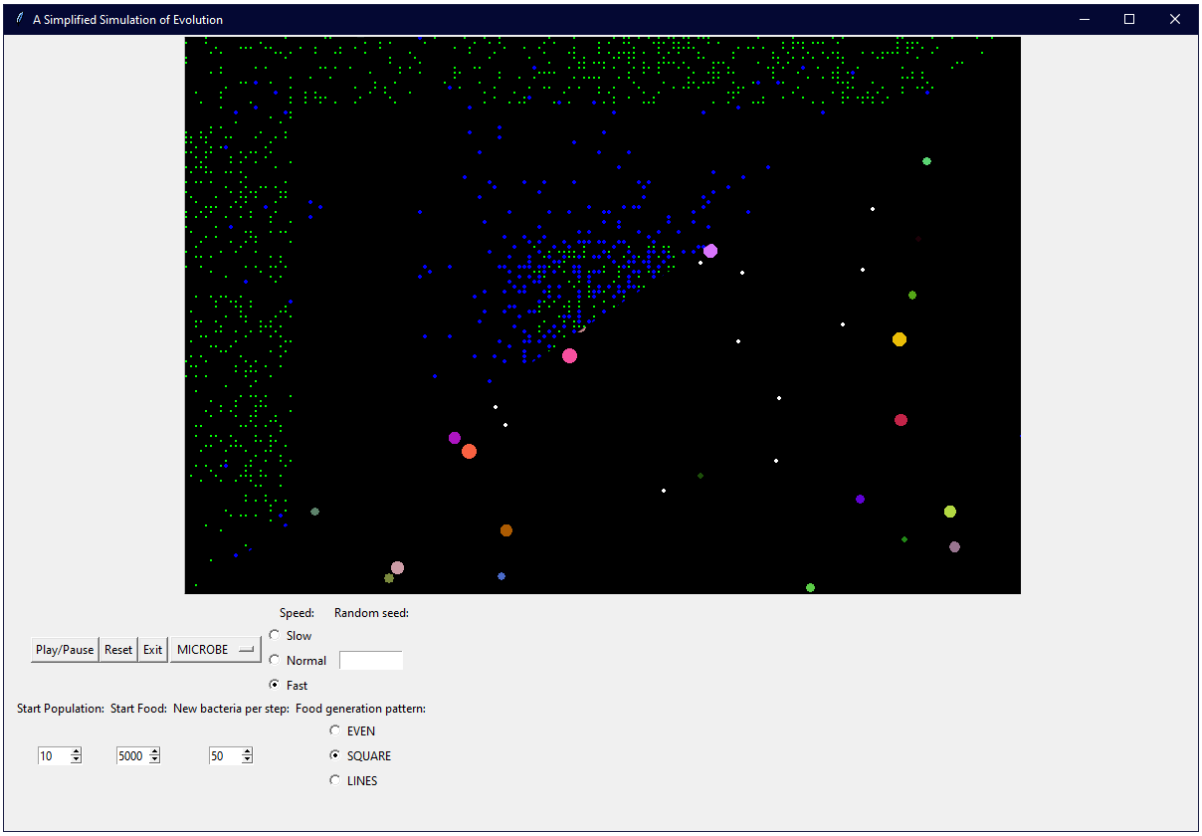
A Simplified Simulation of Evolution



Matura paper by Andreas Ellison

Supervised by Thomas Jenni

05.02.2021 Kantonsschule Zug

Contents

1 Introduction

One of the major goals of science is to understand nature. Our understanding of the world around us is represented in theoretical models, which aim to explain and predict certain phenomena in the real world; phenomena that we can observe or measure with tools. Although models are always an abstraction of reality, reducing nature to a set of well-defined rules, they often succeed in predicting natural phenomena with great precision and we rely on them greatly in everyday life.

The advent of computers gave us the processing power to test our models. To do this we have to formulate and mold our theoretical models or hypotheses into a form that can be understood by a machine: a mathematical model. A mathematical model can then be used to create a simulation. An online dictionary offers the following definition of the word "simulation":

"the representation of the behavior or characteristics of one system through the use of another system, especially a computer program designed for the purpose." [1]

A simulation precisely mirrors a theoretical model, which in turn loosely mirrors reality. By having complete control over a computer simulation, we can extract any piece of information and understand the process much better than any natural phenomenon.

One prime example for testing a theoretical model using a simulation is with evolution. Evolution is a concept that revolutionized our view of the world and is part of our basic education. It is something we cannot directly measure or perceive in most environments because of the vast spans of time over which it operates, and we rely on evidence from fossils to test our theories. Today it is not yet possible to fully represent the natural world's incredibly complex relations at the atomic, chemical, and biological levels on a computer. But, by leaving away the lower levels, using simplified models of evolution in nature and shaping them into simulations we are able to bridge the barrier of time and see what evolution looks like for ourselves.

With modern programming languages and the internet, it has become increasingly simple for anyone to create simple simulations on their computer. This is what I have done. I was originally inspired by a video on YouTube [2], which presents a simple visual simulation of evolution and, through analysis of the resulting data, reflects on some of the basic mechanisms of evolution. The goal of my project was to create such a visual simulation myself and thereby illustrate that using very simple models with some abstract resemblance to the real world, it is possible to show some basic aspects of evolution using a simulation on a computer.

This was achieved by creating a graphical application written in Python, which is able to simulate and visualize models of evolution. Two different models from online sources were integrated into the application. In this paper, the structure of the application will be explained without going into technical details. Subsequently, the two models will be explained and analyzed with data produced by running simulations with different parameters and rules. Using the data some interesting properties of the models will be shown, with which connections can be made to evolution in the real world.

2 Darwin's Theory of Evolution

In 1859, Charles Darwin published his book "On the Origin of Species", proposing a theory for, as the title suggests, the origin of the variety of species that exist in our time. His theory postulates that the individuals of a species tend to differ in their traits over generations. Advantageous traits in individuals that improve their chances of reproduction thrive and may eventually result in the formation of new varieties and subsequently whole new species. Thus, all species would share a common ancestor. This "descent with modification", as Darwin called it, of a species over generations is what we call evolution today.

Evolution naturally occurs through the process of natural selection. Natural selection is a simple mechanism with which individuals of a species can adapt to their environment. Through reproduction, offspring inherit the traits of their parent(s). Due to mutations that occur during DNA replication and/or through genetic recombination resulting from sexual reproduction, the next generation will have slightly different traits than their creators, while still sharing many. After several generations, the differences in some offspring might become more significant and provide survival advantages in the environment, allowing more of this strain of the species to survive in competition with others. Individuals more fit for survival will likely proliferate and sooner or later replace their close relatives or drive them to extinction. Alternatively, they may branch off to form a new variety of the same species.

Darwin's ideas are logical and are arguably concepts that could have occurred to others as well long before. However what Darwin also realized is that the afore-described process takes place across millions of years, impossible for us to witness in our short lifespans. Due to the fact that evolution occurs over such a long span of time, it is mostly unmeasurable (although this is not true for microbial environments, where evolution can be observed after a few hours). The theory is also difficult to test rigorously, as is usually done for other theories. All we can do is see how well it explains trends in the fossil evidence or behaviors that have evolved in animals alive today. Still, Darwin's theory of evolution is very successful at accomplishing its goal, namely, to explain the origins of species and life, and is generally accepted today.

2.1 The Fundamentals of Natural Selection

Although natural selection is a simple concept, it creates very complex relations between species in nature. The aptitude of an individual or a species to successfully reproduce in its environment and generate offspring is often referred to as its *fitness*. The fitness of an individual or a whole species not only depends on the other individuals in the species, but also on other species, the climate and the availability of resources in the environment. When a species adapts to its environment, this is achieved largely through interactions with other species by scavenging their by-products or by preying on them. Through these interactions with other species and the abiotic components of the environment, animals come to form complex ecosystems.

Some species can become incredibly specialized to their environment. Smaller organisms with short lifespans like insects tend to become very specialized and form the base of the complex hierarchy of dependencies between species. Such ecosystems allow for other specialists to develop and have high biodiversity. As a consequence, such species are extremely dependent on the environment. Abrupt change, such as changes to the climate due to global warming, can be devastating to the entire ecosystem.

Although such complex relations are essential for understanding evolution and natural selection, they are beyond the scope of my project. My work focuses on the core principles of natural selection, which are necessary for it to occur. These are the following:

> **reproduction:** Offspring are necessary for the gene pool of a species to be able to change.
> **mutation:** The traits of the offspring must differ from those of the parent(s). Otherwise, no change can occur.
> **competition:** Only a subset of all the offspring should be able to survive, otherwise there will be nothing pushing the gene pool towards more advantageous traits. This means that the environment must be limited in its capacity for serving as a habitat.

All the models for evolution explored in this project rely on these core principles and avoid the more complex aspects of natural selection outlined above.

2.2 An Example: Conway's Game of Life

An example of one of the first simulations on a computer that became very popular is *Life*, devised by John Conway in 1970. The goal of the simulation was not to model evolution, but to create a cellular automaton with some desired properties. A cellular automaton is a deterministic simulation in a grid of square cells. Each cell can have one of a set of predefined states. In Conway's model, the possible states for a cell are *on* and *off*, equivalent to *live* and *dead*, respectively. At the beginning of the simulation, each cell is configured to have one of the available states, after which it is run without any further instruction. The progression of the simulation is processed in individual time steps. In each time step, the next state of each cell is calculated using a set of simple rules, based on the state of neighboring cells. Conway aimed for the simulation to have the following properties, as described in the original article in which his model was presented to the world:

1. *There should be no initial pattern for which there is a simple proof that the population can grow without limit.*
2. *There should be initial patterns that apparently do grow without limit.*
3. *There should be simple initial patterns that grow and change for a considerable period of time before coming to [an] end in three possible ways: fading away completely (from overcrowding or becoming too sparse), settling into a stable configuration that remains unchanged thereafter, or entering an oscillating phase in which they repeat an endless cycle of two or more periods.* [3]

Using these ideas Conway came up with the following simple rules to govern the progression of the simulation, taken from the same article:

1. *Survivals. Every counter [live cell] with two or three neighboring counters survives for the next generation.*
2. *Deaths. Each counter [live cell] with four or more neighbors dies (is removed) from overpopulation. Every counter [live cell] with one neighbor or none dies from isolation.*
3. *Births. Each empty cell adjacent to exactly three neighbors - no more, no fewer - is a birth cell.* [3]

The resulting simulation became very popular and generated a large community of enthusiasts. A fascinating variety of patterns have been continually observed and classified since the creation of the simulation. One category is the so-called *spaceships*, which repeat themselves after a certain

number of steps while traveling on the grid as they shift. Figure 1 illustrates an example of a popular spaceship.

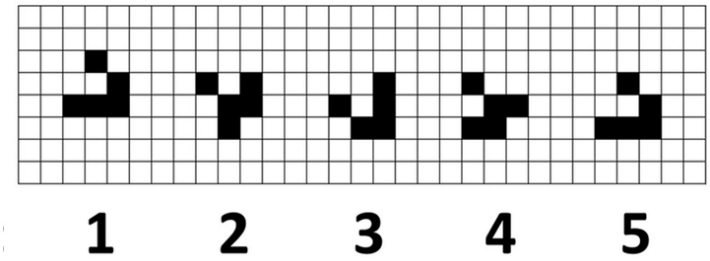


Figure 1: Example of a simple spaceship called a "glider" with all its different states. Source: https://www.researchgate.net/profile/Graeme_Cumming/publication/302393461/figure/fig1/AS:606080598036481@1521512283294/A-glider-as-generated-by-the-rules-specified-by-Conways-game-of-life-This-figure.png

Life uses analogies that mirror the progression of species through birth, death (simplified to over- and underpopulation), and reproduction in nature. The simulation illustrates the fact that deterministic laws can give rise to unpredictable phenomena.

The fundamental principles of natural selection outlined in the previous section are also mirrored in Life. First, reproduction or self-replication can be achieved with some patterns and new patterns are created through interaction with other patterns. Second, mutation of a pattern is possible again through interactions with other patterns. Last, competition is mirrored by rule 3 of the model.

Just as complex organisms can arise from the simple framework of evolution, Life shows how complex phenomena can arise from a simple set of rules. Life is the epitome of how simplicity can arise from complexity and how we can learn about the world around us using abstract models of nature in simulations. Many visualizations of the simulation are available online[1].

[1]For example: https://playgameoflife.com (accessed on 17.01.2020.)

3 Overview of the Application

This project started out as a visual simulation of the same model which originally served as its inspiration. This model will be referred to as the *Primer* model (the YouTube channel where it was discovered is called “Primer”) from this point forward. After the model had been transformed into a working simulation, the architecture of the application for hosting the model was simple to adapt for hosting any other similar model. For this reason, an effort was made for the program to be modular and to accept other models, and hence be able to display multiple simulations. One other model was integrated into the application. However, the project could be extended with any number of additional models with similar principles.

3.1 The Architecture

The whole program is written in Python. The principles of Object-Oriented Programming were applied all over, using *classes* in Python. Classes can be likened to blueprints which define the capabilities of an object (e.g., an animal or a process) in the program. Object-Oriented Programming is a powerful concept about keeping related data and functionality isolated and in the same place. By separating different functional aspects of the program, it becomes much easier to have an overview of the whole and the role of each part of the program can be understood on its own. Additionally, this architecture facilitates extension and maintenance of the program. Object-Oriented Programming is supported in most programming languages and is even enforced in others.

The structure of the application follows the Model-View-Controller software design pattern. This pattern is common for graphical applications and aims to separate the graphical interface (the *view*) and the actual logic or functionality of the application (the *model*, not to be confused with the models for evolution), in this case the simulation itself. To enable communication between the graphical interface and the logic there is the *controller* to take care of this. The view, the model, and the controller all have corresponding Python classes representing them in the program.

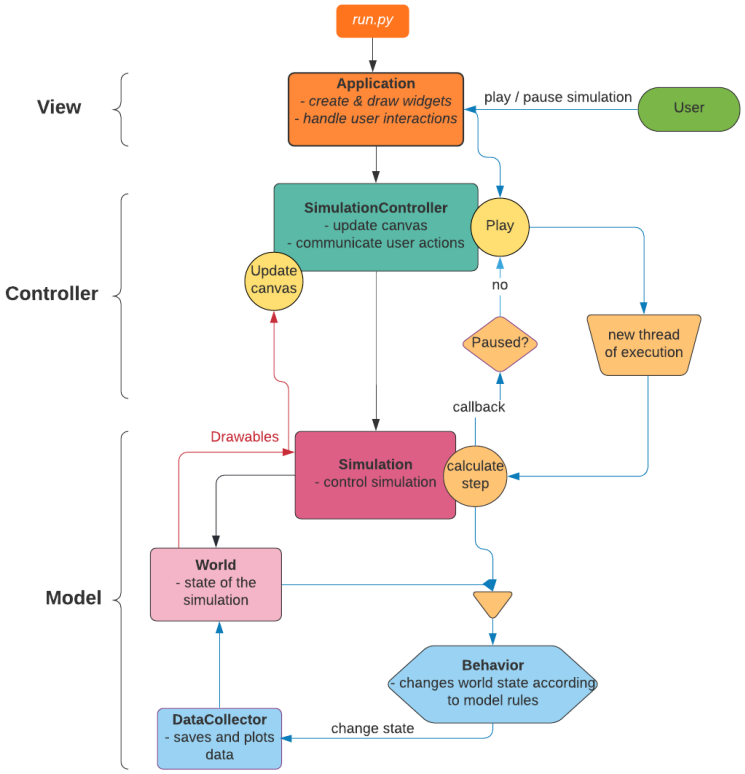


Figure 2: Diagram of a simplified overview of the program architecture and flow. The program is initiated from a short *run.py* Python script, which creates the view component (represented by the Application class). The view then proceeds to create the controller (represented by the SimulationController class), which then initializes the model (represented by the Simulation class). Abstract processes and communication between the components are visualized.

3.2 The View

The view is represented by the *Application* class. This class is at the top of the hierarchy and controls all other components, through communication with the controller. Graphics were implemented using the Python *Tkinter* module [4]. Tkinter is a popular module for creating graphical user interfaces and is very simple to use. The *Application* class *inherits* from a class supplied by Tkinter which serves as the main window of a graphic application. Classes that inherit from another class receive all attributes and functions of the parent class and can define their own. The *Application* class takes care of creating and positioning widgets, such as the canvas or number entries, and handles user actions, which can be actuated through the widgets of the graphic interface. All of this is accomplished using functions supplied by the parent class.

The appearance and composition of the graphical interface are very rudimentary and only consist of the essential widgets needed for controlling the simulation. The interface consists of a canvas for following the simulation visually, a row of widgets shared by all evolution models, and another row for specifying parameters to the simulation specific to the selected model. All models programmed into the application will be explained and analyzed in detail in section 4. Figure 3 illustrates the graphical interface.

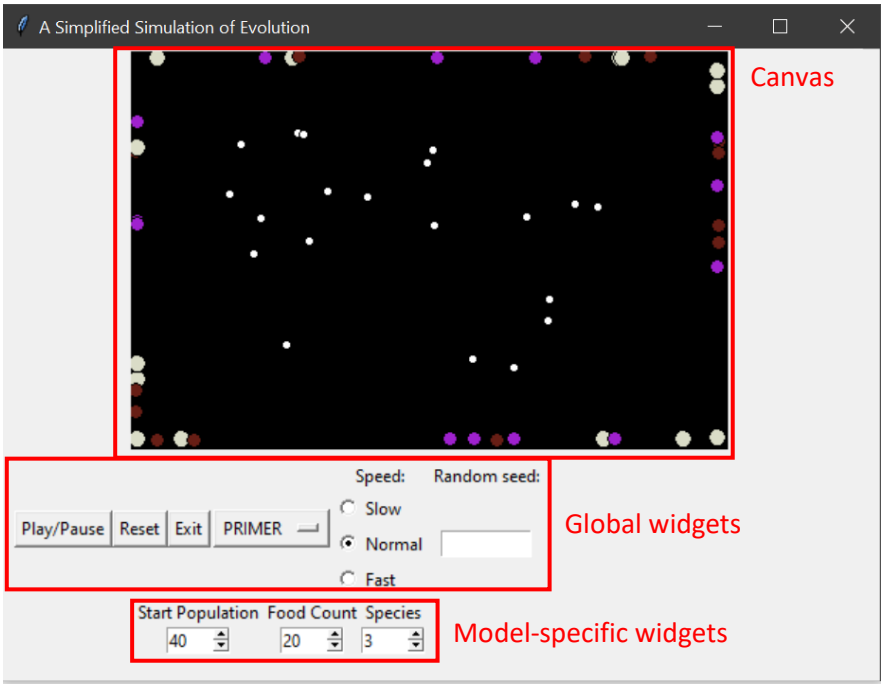


Figure 3: Screenshot of the graphical interface. The Primer model is visible on the canvas in its starting state. All objects on the canvas are circles of different colors. The first row of widgets is shared by all models. The second row is specific to the selected model, in this case the Primer model.

All objects on the canvas are circles of different colors. The Tkinter canvas widget allows other shapes, but only circles were used for simplification. The objects represent concepts such as animals or food of the selected model and are restricted to the visible area on the canvas. Most of the widgets that can be seen in Figure 3 under the canvas are self-explanatory and the ones that may not be will now be explained.

The widget with the text “PRIMER” in Figure 3 allows for selecting between the different models available.

The radio buttons under “Speed” determine how quickly new steps of the simulation are calculated and does not affect the refresh rate of the canvas. The selected model decides the exact behavior of the speed parameter, however all models use it to delay the calculation of steps. This can be used to slow down the simulation to better observe certain aspects.

The value entered in the entry under “Random seed” is used to initialize Python’s random number generator. The module used for this in the program is the built-in pseudo-random number generator in Python. The prefix “pseudo” alludes to the fact that values produced by the generator are deterministic and can be reproduced. They can be reproduced by initializing the generator with the same *seed*, which is the purpose that this entry serves. By populating the entry with a valid integer, a previous simulation for which the seed is known can be replayed. Each time the simulation is run with the same seed and the same parameters, the exact same results are attained. If the entry is left empty, a part of the current Unix time is used as the seed, so that it is essentially unique and not too long.

The use of the widgets specific to the selected model in the second row will be explained in each model’s corresponding section.

3.3 The Controller

The controller is represented by the *SimulationController* class and enables communication between the graphical interface and the simulation. Information about the objects represented on the canvas handled by the class representing the simulation and the controller must know about the attributes of these objects to be able to draw them. To keep this information hidden from the view component, the controller takes care of drawing the objects on the canvas widget and updating them. When the simulation is running, the canvas is refreshed every 17 milliseconds if possible. This means it is redrawn with a frequency of approximately 60 Hz, which is the refresh rate of most computer displays.
Additionally, the controller takes care of setting up the simulation, playing and pausing it, and passing on parameters to it entered from the graphical interface. The controller is notified when parameters have been updated and it immediately communicates the new parameters to the simulation. Thus, updates to the parameters take effect immediately.

3.3.1 Threading

Usually in graphical applications, one encounters the following dilemma: the user interface must be ready for any user actions at all times. However, if a simulation (or some other hidden process) is running in a never-ending loop and effectively blocking other processes, how should the graphical interface be able to react to user interactions and pause the simulation? One solution to this dilemma can be *threading*. Threading allows one to have separate flows of execution in a program, called *threads*, that are handled automatically by Python and the operating system, and give the illusion of the processes running in parallel, although in reality everything is processed sequentially.

Threading is available with built-in libraries in Python and is easy to implement [5]. The simulation runs on a different thread than the graphical interface, to avoid blocking it. Because of the use of threading, calculating a step in the simulation works in the following way: The process of calculating the next step in the simulation is initiated from the controller. The controller sets the process to be run on a different thread and sets a function to be run when the process on the thread has been completed; this type of function is called a *callback*. The callback then does the following: if the state of the simulation is set to “play”, it will initiate the same function that started the whole process of calculating a step, forming a loop. However, if the state of the simulation is set to “pause”, which is done from the thread of the graphical interface through user interaction, nothing will happen and the loop is terminated.

This means that the simulation can be played and stopped as one would expect, but it also means that the simulation will not actually stop until the calculation of the current step is completed. However, individual steps are usually completed in under a hundred milliseconds, so this effect goes mostly unnoticed.

Tkinter also provides ways to run functions without blocking the graphical interface, which is used to refresh the canvas. In Tkinter, jobs like this put up for processing are handled by a so-called *event loop*. The equivalent concept exists for web browsers, which of course is another environment for graphical interfaces.

3.4 The Model

The model is represented by the *Simulation* class. By itself, it is a small class, and its job is mainly to connect the component that stores the state of the simulation and the component that represents the model of evolution and acts on the state to execute a step in the simulation.

Object-Oriented Programming is almost a must when creating simulations, as the subjects therein, referred to as *agents*, should only have knowledge of themselves and should have attributes and behavior they share with other agents, meaning they should all share the same "blueprint". Agents are defined in a class defines the information contained in the agent and its functionalities. Using a class, any number of instances of that class can be created and these instances can be manipulated uniformly.

Packing all information relevant to an agent or a whole component of the simulation allows for abstraction in the communication with other components. Such abstraction was used for all agents, in other words, all entities which have a representation on the canvas of the graphical interface. All agents inherit from the *Drawables* class. Just as for the *Application* class, inheritance allows subclasses to extend the parent class, while sharing a common interface of functions or attributes. The *Drawables* class defines the basic attributes necessary to draw an agent on the canvas: position, previous position on the canvas, color, radius (as all objects on the canvas are circles), and an ID that links to the object's representation on the canvas. The controller uses these attributes to draw or update the agents on the canvas. Because all agents inherit from the *Drawables* class, they can all be drawn in the same way with the same code.

3.4.1 Behaviors

In the program code, a *Behavior* represents the model behind a simulation which determines its progression. The rules of the simulation are separate from the class representing it. All models must define a Behavior as a class, which again inherits from a base class for all Behaviors. This base class defines the interface for communicating with the class representing the simulation. The concept of Behaviors was taken over from the source code of the Primer model [6]. Because a Behavior represents the model, in other words, the set of rules that control the simulation, all Behavior classes are static. This means that they have no state and are not used to make an object. The Behaviors are more like a set of functions which change the state of the simulation. A model simply defines a set of rules for initializing the state of the simulation and then calculating the next step from a given state. By being static, the Behavior mirrors the model not only logically but also architecturally, which is the reason the design was adopted.

However, there must be some way to keep track of the simulation's state, which is achieved with yet another *World* class. The World class represents the state of the simulation. All properties of the simulation are saved in it, such as the agents, the boundaries of the two-dimensional world, and parameters.

Different models also often require different controls for specifying parameters. For this purpose, a Behavior may define a configuration, which describes what kind of widgets (radio buttons or entries) should be drawn on the graphical interface and how they should be labeled. These are the widgets in the second row in Figure 3. The values of the widgets are stored in the World object, where the Behavior has access to them during initialization and when calculating a step.

Although having a visual representation of the simulation is helpful, data is much more useful for analyzing the results. For this reason, all Behaviors must also define a *DataCollector* class. This extracts data from the World object after each step and saves it periodically in the JSON file format and also draws plots using the data, which are saved as images. The *matplotlib* module in Python was used to do this [7]. The raw data saved in JSON also includes the seed used to initialize the random number generator, which can be used to recreate a simulation as explained in section 3.2.

3.4.2 Running the Simulation

First, in an initialization step, the Behavior creates agents and stores them appropriately in the World class. The controller can access them through the model to draw them. The Behavior may also store other data in the World at any time to increase performance in future steps.

After the initial state of the simulation has been set, the simulation is ready to be run. This begins after interaction from the user interface. As explained above in section 3.3, steps are calculated until the simulation is paused again. To calculate one step, the model simply calls the correct function of the selected Behavior and passes on the World object as an argument. The Behavior can then modify the state of the World by acting on the agents to produce the next state. The process of calculating new steps can only end if all agents in the simulation have deceased. The exact method for determining when this is the case is of course defined in the Behavior. As stated above, the *DataCollector* collects data after a step and may write all of its current data to a file and draw plots. The process of saving the data is again done on a separate thread to avoid delaying the simulation.

4 The Models

Two different models were implemented into the application. Both models are from outside sources online. They will be explained to the reader and subsequently analyzed using data collected while running simulations with different parameters. The simulations were run visually using the graphic application and using Python scripts to gather data over longer periods of time or investigate specific aspects.

4.1 The Primer Model

This project was originally inspired by the Primer model and its visualization in a video on YouTube [2]. A comprehensive visual representation of the model exists online [6]. The source code of the online visualization, written in the Rust programming language, is publicly available online [6]. The exact details of the model were understood by reading the source code. The simulation was then integrated into the program based on the original model. The rules of the model as they will be presented in the next section reflect the details of how it was finally integrated.

4.1.1 The Rules

The Primer model represents a population of colored animals scavenging for stationary food. All animals share the same set of variable traits: sight range, radius, and speed. Additionally, every animal has an energy level. When an animal's energy level drops below zero, it dies and is removed from the simulation.
The traits employed in the model are more or less self-explanatory:
An animal's sight range determines the radius within which it can detect food around it. An animal's radius (r) determines its size in the world and its maximum energy (E_{max}), together with a constant (c_E):

$$E_{max} = r^3 * c_E \qquad (1)$$

This relationship is derived from the relationship between volume and mass in physics, omitting any constant factors. Mass was assumed to have a linear relationship with energy capacity.
Lastly, an animal's speed determines the distance by which it can move in a single step.

The traits of an animal determine its color using the RGB color model. The red, green, and blue color values are determined by the radius, speed, and sight range traits, respectively. The intensity of each color value is defined by where the trait lies between the minimum and maximum allowed values for that trait.

The exact minimum and maximum values chosen for the traits and for the constant c_E will not be specified, as they were chosen based on the dimensions of the application and thus depend on external factors.

Each animal starts at a random position at the edge of the world. A variable number of animal species with random traits are created when initializing the world, which serve as templates to create the initial number of animals. Food is randomly distributed across the world, within a certain distance from the borders. Figure 4 illustrates an example of what the initial state of the Primer model might look like.

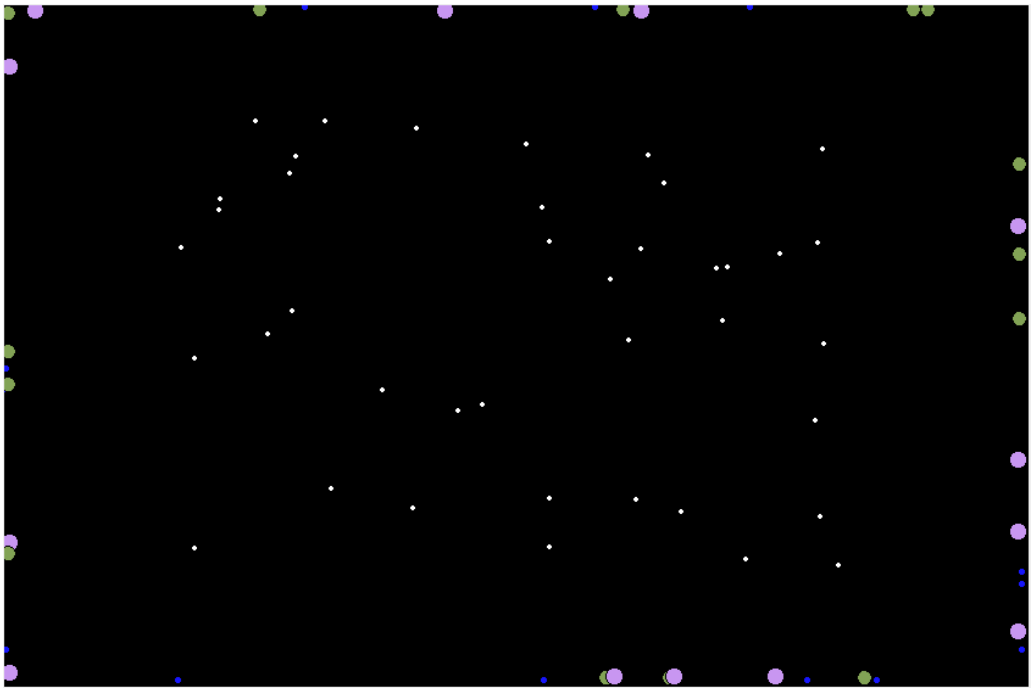


Figure 4: Example of a starting state of the Primer model, with 40 animals, 20 pieces of food, and three different species.

Once the simulation is initiated, a "day" has begun and each animal first directs itself to the center of the world. If nothing is in an animal's range of sight, it changes its direction of travel by a random angle between $-20°$ and $+20°$ and moves at its given speed. This causes the animals to randomly wander around the world. If an animal attempts to wander outside of the world, it is again directed towards the center.
When food comes within an animal's range of sight, the animal moves directly towards it. Once the center of the food is within a distance of twice the animal's radius, it is consumed by the animal. An animal may consume at most two pieces of food. When it has eaten one piece and is running low on energy or has managed to eat two pieces of food, it returns to the nearest edge in the world and "sleeps".

Each step costs energy based on how far the animal has moved (d), its radius (r) and its sight range (s):

$$cost = r^3 * d^2 * s \qquad (2)$$

The first two factors in the equation mimic the classical formula for the kinetic energy of an object:

$$E_{kin} = \frac{m * v^2}{2} \qquad (3)$$

This relation to the real world is arbitrary and does not make the simulation any more realistic.

When an animal is asleep, it no longer loses energy. To decide whether an animal is low on energy and should go to sleep, the energy required to travel to the nearest edge of the world is first calculated. If the quotient obtained by dividing the animal's remaining energy by this value is smaller than 1.5, the animal is low on energy.

One day in the simulation ends when all animals that are still alive are asleep. This means that only animals that have eaten one or two pieces of food survive to the next day. All animals which were

able to eat two pieces of food create a copy of themselves with slightly mutated traits. Each animal starts the next day with replenished energy and new food is generated. This process is repeated indefinitely, unless all animals in the simulation die out.

The parameters available for modification from the graphical interface are the following: the initial population of animals in the world, the number of different species to be used as templates for generating the initial population, and the number of food pieces to be generated after each day.

4.1.2 Analysis

Several different plots were implemented for analysis of the simulation. Several of the ideas for plots were taken from the online implementation of the model, which also allows a similar analysis of the data [8]. Because of the poor performance of the implementation of the model, the simulation was mostly run with less than 100 animals.

Initially, the constant c_E was chosen so that an animal with minimum radius and maximum speed and sight range could make enough steps to cross the world once in its width (not accounting for random changes in direction due to wandering). This was done so that many varieties are able to survive.

The simulation was run for 1000 days, starting with a population of 40 animals, 40 food pieces and 3 species. Watching the visualization of the simulation one can see the color of the animals gradually shift first to green and blue tones and then become brighter and increase in size. This already suggests that due to natural selection, the animals have adapted first to have higher values for the speed and sight range traits, as these are responsible for greener and bluer colors, respectively. Figure 5 shows the average values for the traits in percent of the minimum and maximum allowed value for the respective trait. The trend confirms the observations above. Figure 6 also shows the distribution of the traits after the last step.

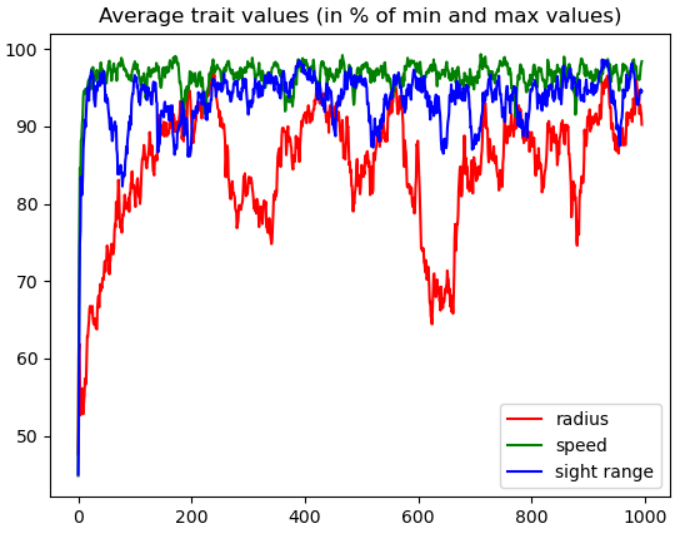


Figure 5: Plot of the average values of animal traits in percent of the minimum and maximum allowed values over time (measured in days of the simulation).

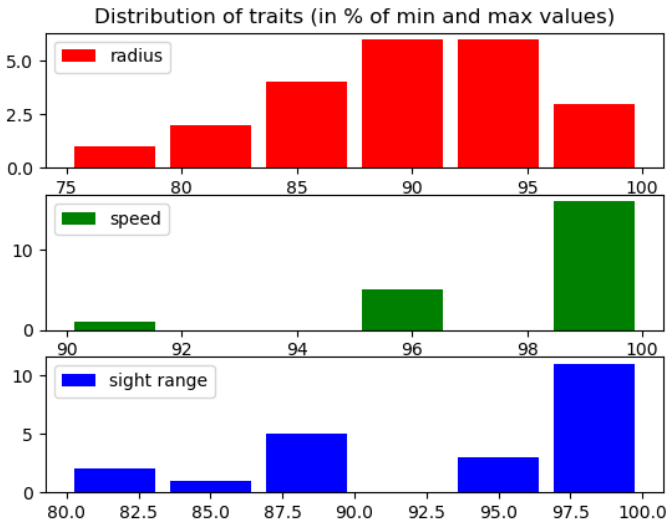


Figure 6: Distribution of traits on the last day of the simulation. The x-axis is in percent of to the minimum and maximum value for the respective trait. The y-axis shows the number of animals belonging to a column.

Although Figure 5 shows the general tendency of the traits, using only this plot it is not clear how spread out the values are. Figure 6 illustrates that maximum sight range and speed values are strongly favored. Thus, having good sight or being fast is clearly worth it, despite the energy penalty this comes with in equation 2.

As for radius, although larger values are still favored, it seems there is a bit more flexibility here. The notable jumps in average radius in Figure 5 could be explained with the spread-out distribution in Figure 6. Larger values are not as strongly favored as with the other traits, which makes it possible for minorities with smaller radii to get lucky in a few rounds and proliferate. This would have a large effect on the average for several days, as in Figure 5, but ultimately values close to the maximum still appear to be favored. It is interesting to note that the big shift in average radius did not seem to have a notable effect on the sight range and speed averages. This means that the animals have a lot of energy to spare, which would also explain why the evolutionary pressure to become larger is less intense than with the other traits. The same trends were seen after running the simulation multiple times.

It is also interesting to note that the population quickly plummeted after the first day and remained stable at around 25, despite the constant daily food supply of 40. Figure 7 shows the population over time.

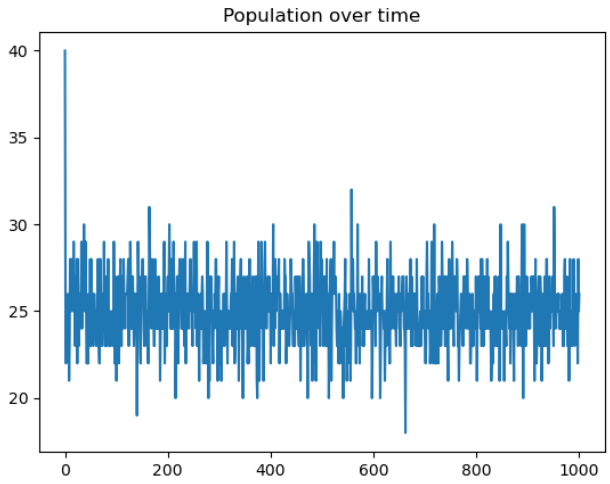


Figure 7: Population over time

From Figure 7 it follows that the average animal eats about 1.6 pieces of food. Thus, even after all animals have close to optimal traits, they are still vulnerable to the element of chance, because of where they wander and how close food is spawned.

To better understand the bias towards maximal values in the traits, the maximum energy level constant c_E from equation 1 was cut in half. The simulation was then rerun for 1000 days with the same parameters. This time the animals' colors gradually shifted to yellow tones. Figures 8 and 9 show the resulting average trait values and their distributions.

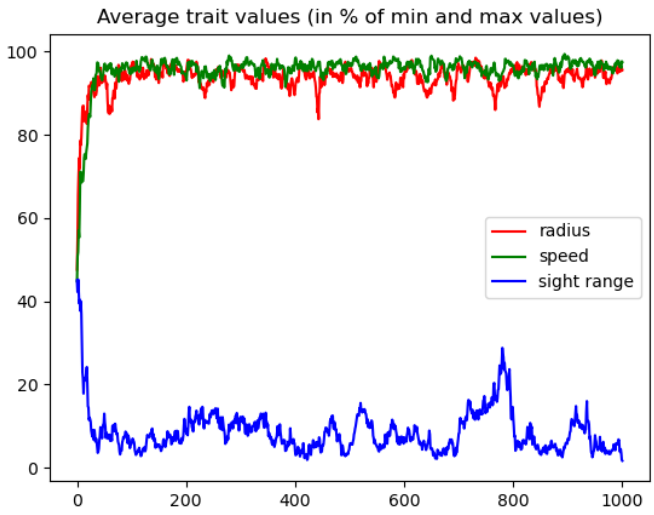


Figure 8: Average trait values in percent after cutting the constant C_E cut in half.

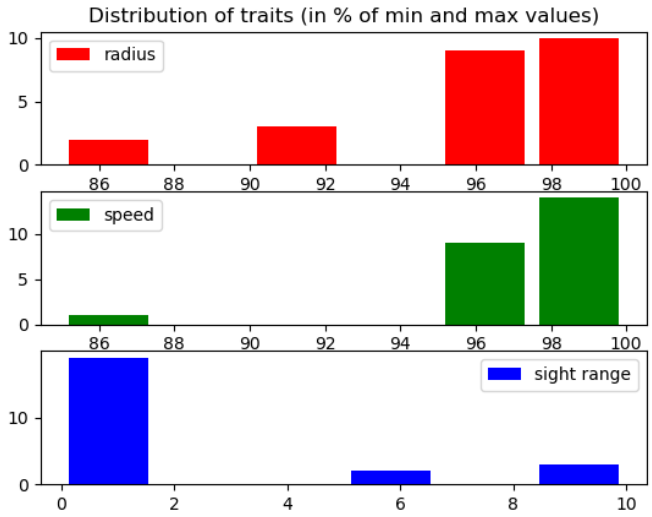


Figure 9: Distribution of the traits in percent on day 1000 after cutting the constant c_E in half.

Interestingly, the sight range trait has ceased to become important and plummeted to values between 0% and 20%. Speed has remained just as important and radius has become more decisive. This demonstrates that the sight range trait is significantly less valuable than the speed trait in general. A larger radius might enable an animal to use more energy. With the limited amount of energy, animals would need a large radius to be able to have a large value for the speed trait. Additionally, a large radius might be enough to find food by randomly running into it, since an animal does not have to have seen a piece of food to eat it. This might also explain why there were more fluctuations in the radius in Figure 5: Because the animals had enough energy to have a high range of sight, they didn't have to rely on a large radius to find food and thus the evolutionary pressure to have a large radius was significantly weaker.

Again reducing the constant c_E, this time dividing it by ten and again running the simulation for 1000 days produces the plot of averages in Figure 10.

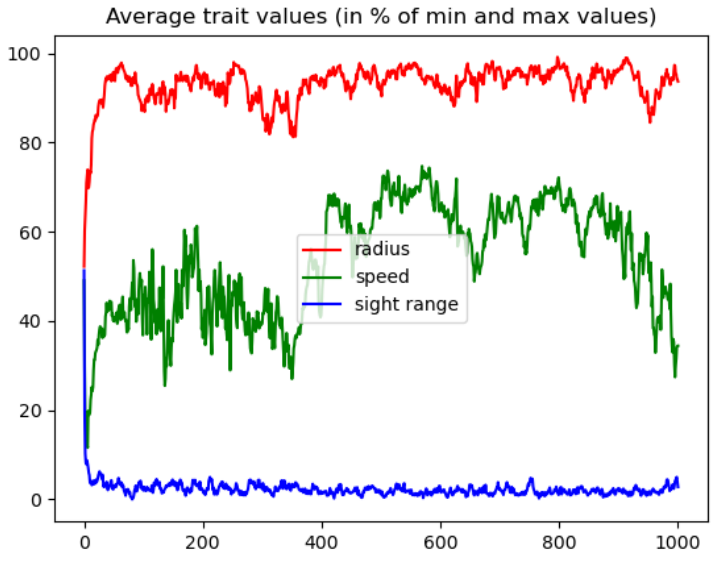


Figure 10: Average trait values over time after dividing c_E by 10.

This further supports the hypothesis that the speed trait is essential and together with the previous results shows that there is no disadvantage to having a large radius.

This observation can be explained mathematically. The following equation combines equations 1 and 2 and shows how many steps an animal can make in total (n), assuming it always moves by its speed (v):

$$n = \frac{E_{max}}{cost} = \frac{r^3 * c_E}{r^3 * v^2 * s} = \frac{c_E}{v^2 * s} \qquad (4)$$

The radius cancels out and thus a larger radius does not result in more energy to spare. The effect of encountering more food randomly might be the sole benefit. Using these insights, a few changes were made to the model in an attempt to balance out the traits and give them more ambiguous meanings.

First, instead of a constant limit of $\pm 20°$ for the change in direction when wandering, a relationship between the speed trait and the limits for change in direction was created. This was done with a quadratic relationship between speed (v, v_{min} and v_{max} being the minimum and maximum allowed speed values, respectively) and the limit for changing direction (α_{max}) using the following equation:

$$\alpha_{max} = \frac{\frac{\pi}{2} * (\mathrm{v} - v_{min})^2}{(v_{max} - v_{min})^2} + \frac{\pi}{10} \qquad (5)$$

This equation translates into larger changes in direction for higher values of the speed trait while wandering. With this equation the minimum possible value for α_{max} is 18° and the maximum value is 118°. The minimum of 18° was introduced to avoid animals simply going back and forth from one end of the world to the other, always taking the same path through the center as animals are redirected towards the center upon reaching an edge. The actual change in direction is a random value between $-\alpha_{max}$ and α_{max}, as before.

Second, a similar effect was introduced for the sight range trait. A linear relationship was introduced between an animal's sight range and its field of view. Before all animals could see 360° around themselves. This time their field of view (φ) was determined using the following formula based on the sight range trait (s, s_{min} and s_{max} being the minimum and maximum allowed values for the trait, respectively):

$$\varphi = \pi - \frac{\pi * (\mathrm{s} - s_{min})}{(s_{max} - s_{min})} + \frac{\pi}{10} \qquad (6)$$

With this equation, better sight results in a smaller field of view. The maximum field of view is 198° and will always be at least 18°. Animals are allowed to see food $\pm \frac{\varphi}{2}$ degrees from their previous direction of motion and inside their range of sight.
The model was also modified so that animals could only eat food that they could actually see and that they were looking for, removing the advantage of having a large radius of simply coming into contact with food by accident.

Third, a simple change was introduced to equation 1. The radius was raised to the power of four instead of three, resulting in the following new formula for calculating the energy cost of a step (with the same meanings for the variables as in equation 1):

$$cost = r^4 * d^2 * s \quad (7)$$

Thus, the number of steps an animal could make in total (n) was now:

$$n = \frac{r^3 * c_E}{r^4 * v^2 * s} = \frac{c_E}{r * v^2 * s} \quad (8)$$

The constant c_E was redefined so that an animal with the maximum values for all traits would be able to travel by the width of the world once, as now a larger radius meant faster energy consumption.

With the new definitions of all traits, it is difficult to predict what combination of traits would be optimal. However, it is reasonable to guess that having a large radius has lost its value as there is no longer any obvious benefit of being large. On the contrary, a large radius should be a disadvantage. Running the model with these new rules and the same configuration as for the previous runs for 1000 days produces the results illustrated in Figures 11 and 12.

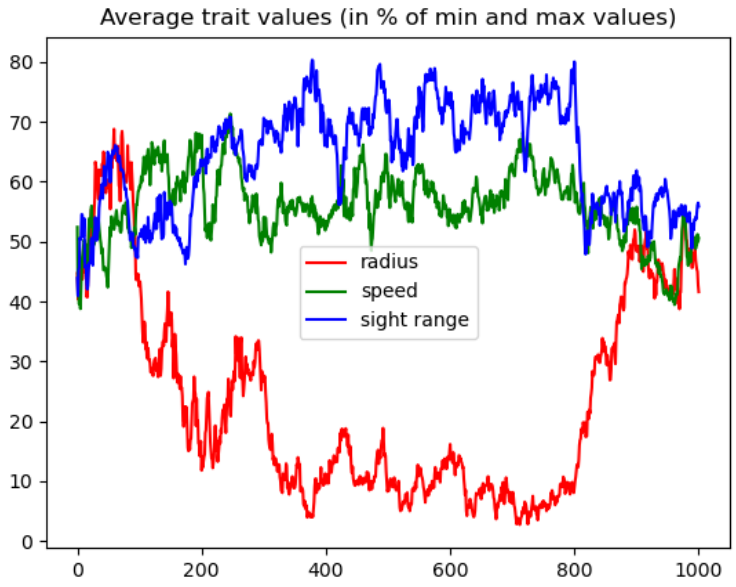


Figure 11: Average trait values over time with the new set of rules and the new definition for constant c_E.

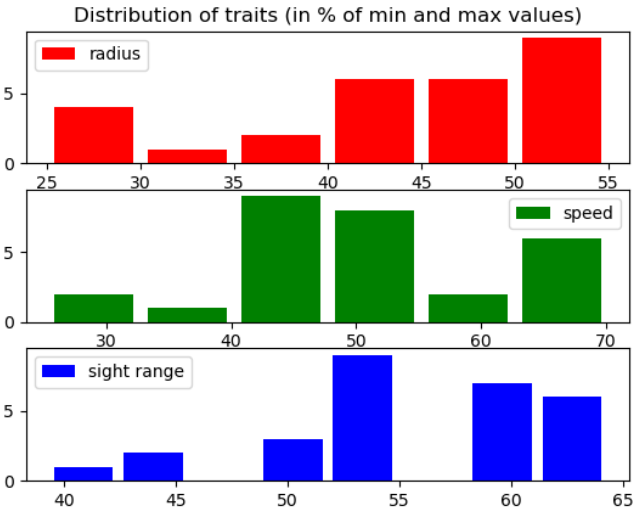


Figure 12: Distribution of traits in percent on day 1000 with the new set of rules and the new definition for constant c_E.

In general, the results are much more unstable and show a high tendency for variation. Figure 12 shows that the range of optimal values is much larger than with the previous model rules. The values are less concentrated in short ranges and much more spread out. This can further be shown by comparing the standard deviations of the traits in these results to the data used for Figures 8 and 9. The standard deviations for the radius, speed, and sight range traits on the last day were 8.7%, 11.4%, and 6.5%, respectively for the simulation behind Figures 11 and 12. The standard deviations in the simulation used for Figures 8 and 9 were 4.6%, 3.5%, and 3.4% on the last day, respectively. Thus, the variance in the traits has increased significantly, especially in the speed trait.

The strong fluctuations of values in Figure 11, especially in the radius trait after day 800, suggest that with the new rules the traits might take longer to stabilize. To verify this, the simulation was run for 5000 days. The resulting average traits can be seen in Figure 13.

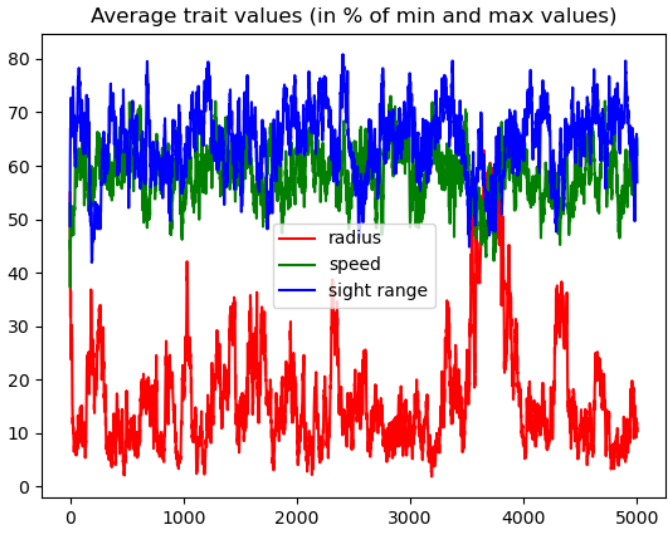


Figure 13: Average trait values over time after 5000 simulation days

More conclusive results can be drawn from Figure 13. As predicted, the radius trait has lost much of its importance, although it frequently fluctuates seemingly at random. In both Figures 11 and 13 there seems to be a relation between the fluctuations in average radius and the fluctuations in average sight range. This can be seen around days 200, 400 and 800 in Figure 11 and around days 200 and 3800 in Figure 13. One possible explanation is that a larger radius, leaves less energy for having good sight. However, there are exceptions to this tendency, for example around day 100 in Figure 11.

One explanation for why the radius trait still seems to have a benefit for survival is that animals with a larger radius are simply able to consume food earlier than other animals with a smaller radius, because an animal can eat a piece of food once it is within twice its radius. This might lead to the emergence of larger animals that have a significant effect on the average and cause the large fluctuations in Figures 11 and 13.

The speed trait shows the most stability in Figure 11, however, at the same time, it had the largest standard deviation on day 1000. The relationship between speed and direction change seems to have a large impact on survival but also seems to allow for more variation.

Sight range has become more important. This is to be expected, as an animal must now see food to be able to eat it. The average value for the trait seems to lie around 60% in both Figures 11 and 13, which would result in a field of view of about 90° according to equation 5.

To test how strong the effect of having to see food in order to actually eat it has in the simulation, the rule was temporarily removed and the simulation was run again for a period of 500 days. Already the results illustrated in Figure 14 are very compelling.

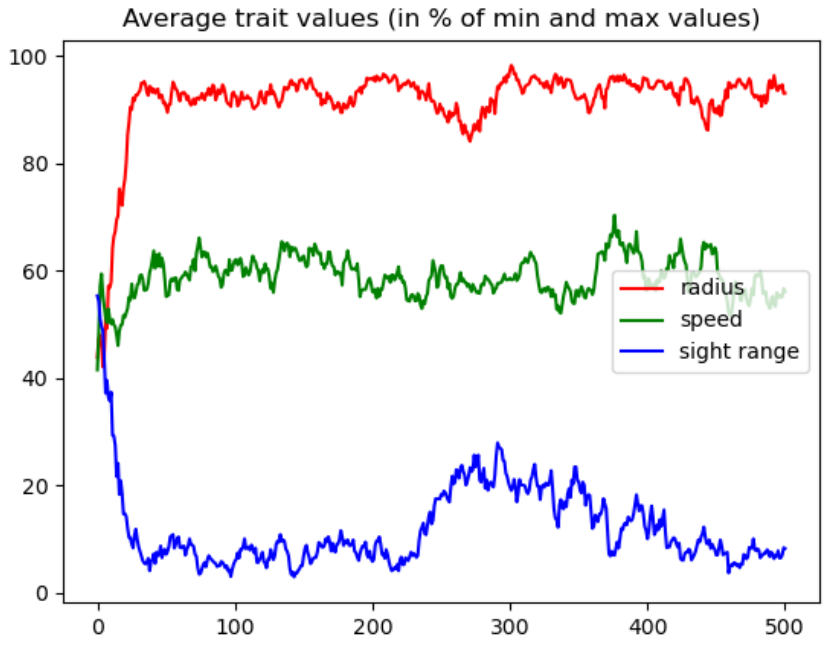


Figure 14: Average trait values over 500 days with animals not having to see food to eat it.

The results in Figure 14 again demonstrate the significance of having a large radius seen in previous simulations. In the variant of the Primer model where animals are not required to see food to eat it, the effect of randomly running into food by accident seems to be a large factor for survival.

4.1.3 Conclusions

The Primer model is a good example of how easy it is to create an environment where natural selection can occur using intuitive rules. With the initial configuration of the model, some values for traits were clearly more beneficial and animals quickly adapted.

However, at the same time, this model shows it is significantly more difficult to create a good balance between traits, which allows for high diversity and does not clearly favor some traits over others.

The model has a poor performance and there is no clear solution to optimizing the implementation. The coordinates and the sight range trait have continuous values and because of this it is necessary to process all food in the world individually for each animal to check whether it can see food. The effect of this is illustrated in Figure 15, which shows the performance of the simulation at each time step.

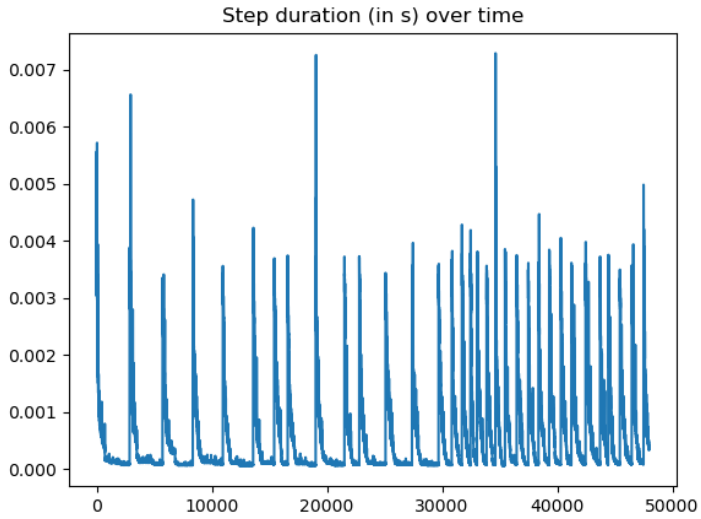


Figure 15: Duration of a step in the simulation over time.

Spikes in the data occur after each new day in the simulation. The amount of time it takes to calculate steps drops drastically as the day progresses, as food disappears and as animals die or go to sleep. The performance of the program is strongly impacted by the amount of food in the world, which limits the program in simulating larger populations. The poor performance of the model limits its capabilities, and perhaps more insights could have been gained with larger populations.

Furthermore, the model simplifies complex aspects of real life, which, while intuitive, restricts the model. The specificity and extent of the rules allow for many different possibilities to extend the model and to create and test interesting relationships between traits. However, such deliberate relationships and all the different constants and parameters embedded into the rules of the model make it difficult to draw any clear conclusions about the real world. The model lives in a very specific world of its own. Apart from illustrating natural selection in action, the model is somewhat limited in terms of offering real insight into natural phenomena.

4.2 The Microbe Model

This model was also discovered online as a simple set of rules for simulating evolution, accompanied by a visualization [9]. The online source provides a good explanation of the model and the source code was again publicly available [10]. The model was explored because its rules seemed much more fundamental, and thus more powerful, than those of the Primer model. Another aim in integrating this model was to prove that the application could house a very different model world arranged into cells. Also, due to the ordering the world into a grid it was clear that this model would be much more performant than the Primer model. The source code was again consulted to understand the details of the model and integrate it into the application.

The goal of the model is to show how agents are able to adapt to different environments using a very simple set of rules.

4.2.1 The Rules

The model is comparable in its simplicity to the one outlined in section 2.2. The following explanation inevitably has a lot in common with the explanation that can be found on the website where it was discovered [9].

The model simulates a population of microbes eating stationary bacteria in a grid world of square cells. Bacteria are distributed across the world and eating bacteria gives a microbe a constant amount of energy. Figure 16 provides an example state of the Microbe model. Microbes are colored blue and bacteria are colored green.

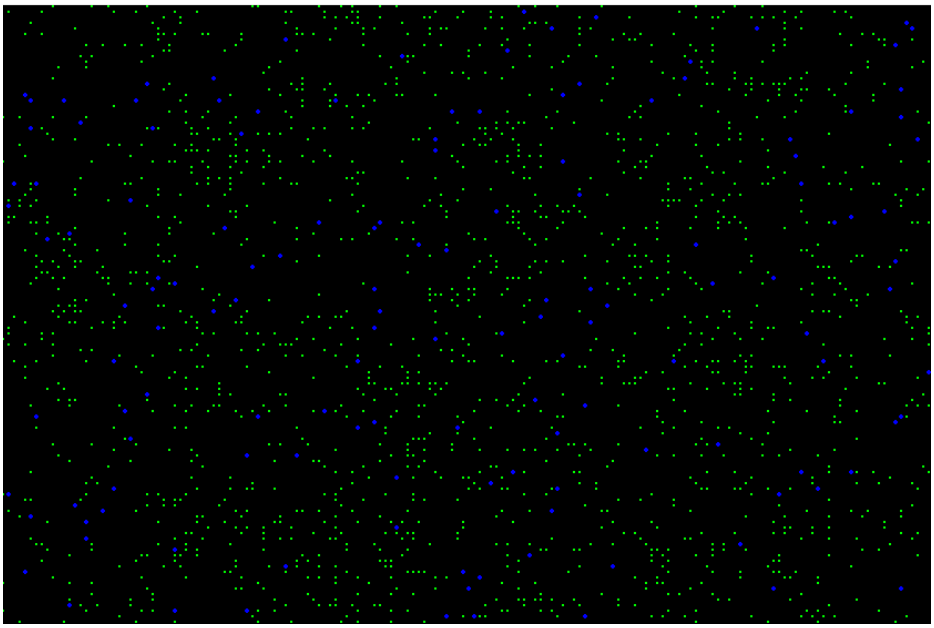


Figure 16: Screenshot of the microbe model in an advanced state. Microbes are visible in blue and bacteria in green.

The microbes generated at the beginning all have the same amount of energy. In each step of the simulation, a microbe loses one unit of energy and can move to one of its eight neighboring cells, depending on its current direction (which starts at a random value). If a microbe leaves the boundaries of the world, it reenters on the opposite side. Every possible direction has an index, starting from the top left and moving around clockwise.

A microbe can change its direction after each step. Changes in direction are determined by the microbe's genes. These are represented by an array of eight probabilities, equal to the number of possible directions, that are normalized and thus sum up to 1. These probabilities (p) determine the change in direction after one step based on the following calculation: A random value (r) between 0 and 1 is computed and the smallest value i is chosen such that the following inequality holds:

$$r < \sum_{n}^{i} p_n \qquad (4)$$

The new direction (d_{t+1}) is determined as follows:

$$d_{t+1} = (d_t + i)\ mod\ 8 \qquad (5)$$

Figure 17 illustrates the meaning of each gene, which follows from equation 5 and the clockwise ordering of the directions.

Gene		Function
Index i	Identifier	
0	p_0	No change
1	p_1	light turn to the right
2	p_2	right turn
3	p_3	hard right turn
4	p_4	reverse
5	p_5	hard left turn
6	p_6	left turn
7	p_7	light left turn

Figure 17: The effects that different genes (p_0-p_7) have on a microbe's direction of travel. Source: https://beltoforion.de/en/simulated_evolution/ (accessed on 17.01.2021)

Each change in direction has a corresponding energy cost, with larger changes in direction resulting in a larger cost than smaller ones. The different costs of the possible changes in direction are visualized in Figure 18.

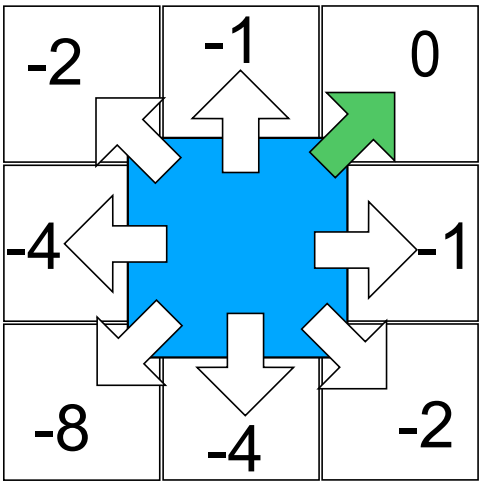


Figure 18: Example of the cost of changing direction with the green arrow representing the current direction of the microbe. Harder turns result in a higher cost than lighter turns. The relationship is analogous for all directions. Source: https://beltoforion.de/en/simulated_evolution/ (accessed on 17.01.2021)

If a microbe moves to a cell with food and has an energy level less than a certain constant, it will eat the food. A microbe can reproduce once its energy has reached a certain threshold. The microbe then makes a copy of itself, with one of its probabilities slightly mutated. The "child" gets half of the parent's energy, leaving the parent with the other half.

After each step in the simulation, new bacteria are generated. The starting number of bacteria may differ from the number of bacteria generated after each step. Three different patterns for distributing new bacteria across the world are possible as a means of pressuring the microbes to adapt their movement to different distributions. These patterns are illustrated in Figures 19-21.

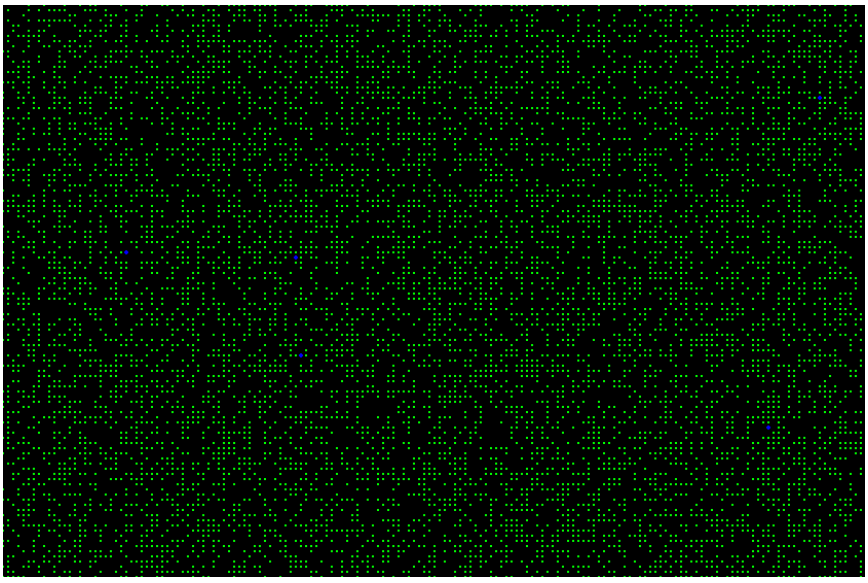


Figure 19: Bacteria are uniformly distributed across the whole world.

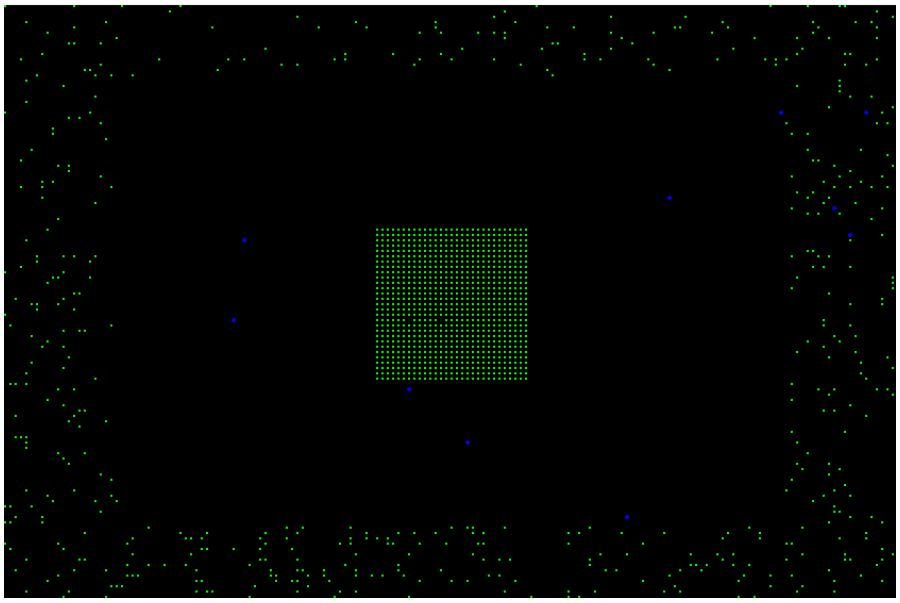


Figure 20: New bacteria are generated disproportionately, concentrated in a square in the center of the world and more sparsely around the edges.

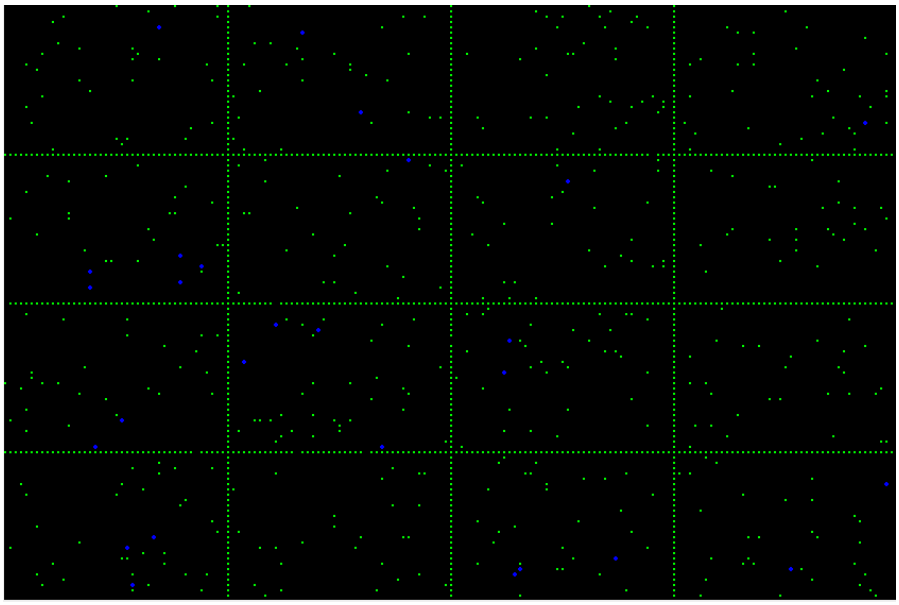


Figure 21: Bacteria are generated along fixed horizontal and vertical lines and more sparsely across the whole world.

The patterns for distributing bacteria illustrated in Figures 19-21 will be referred to as the *Even* pattern, the *Square* pattern and the *Line* pattern, respectively.

From the graphical interface, parameters are available for setting the initial number of microbes and bacteria, the number of bacteria to be generated after each step and the pattern to be used to generate new bacteria.

4.2.2 Analysis

The same constants were used as in the original implementation of the simulation [10]. The initial energy for a microbe was 100, the energy received from eating food was 40, the energy required to reproduce was 1000 and the maximum energy a microbe could have was 1500. The large amount of energy received from eating bacteria and the large energy limit allow microbes to survive for a long time without eating food and to reproduce quickly if there is an abundance thereof.

Due to the use of cells as units in the world, the simulation is very efficient for simulating large populations.

A large initial number of bacteria was necessary to run the simulation, as otherwise the microbes often died out. For all runs of the simulation, the number of bacteria generated was 5000 and the number of bacteria generated after each step was 10.

4.2.2.1 Distribution of Bacteria with the Even Pattern

Without any initial preconceptions, the simulation was run with the parameters above and a starting population of 10 creatures. When observing the simulation, it is apparent that it takes some time for the first few creatures to eat enough bacteria to reproduce. Once some fitter microbes begin to reproduce, with the overwhelming number of bacteria, the population rises exponentially. This continues until food starts to become scarce and, after a while, an equilibrium takes hold, after which no more significant changes in population are visible (as in Figure 16). It is difficult to discern the movement patterns of the microbes only by watching the visual representation as there are many microbes. Figure 22 shows the development of the microbes' probability "genes" after 10^5 time steps.

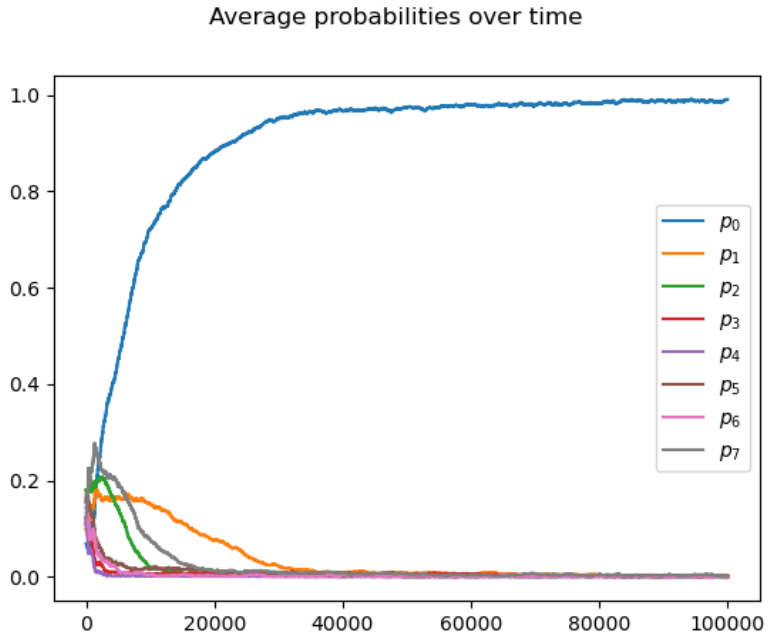


Figure 22: Averages of the probability genes p_0-p_7 during 10^5 time steps

The probability gene p_0 approaches a value of 1 and conversely the genes p_1-p_7 approach a value of zero. The shape of the curve for p_0 in its rise towards 1 can be likened to an exponential decay. This can be explained by the fact that the closer the microbes are to a probability of 1 for the p_0 gene, the less they benefit from having an even larger probability.
Referring to Figure 17, p_0 is the chance of a microbe not turning at all after a step. Thus, most microbes just travel over the map without changing direction and make use of the rule that when they exit the world on one side, they reenter it on the other. The apparent advantage travelling in a straight line provides could result from the microbes being able to travel faster across the world, instead of remaining in the same regions. Because bacteria are distributed evenly, it is more probable that a microbe will find bacteria far away, rather than close-by after having eaten one. Furthermore, the microbes can avoid the penalty of changing direction, as illustrated in Figure 18.

Most of the other probability genes show a similar exponential trend as they diverge to 0. The genes p_1 and p_7 however seem to have slightly less evolutionary pressure to decline and are thus associated with a greater likelihood of survival than the remaining genes (p_2-p_6). Again, consulting Figure 17, these determine the probability of making a slight turn to the left or right, respectively. They do not cost as much as making a bigger turn (see Figure 18) and divert a microbe less in its travel.

Figure 23 shows the distribution of the probabilities for p_0 and p_1 after the last step.

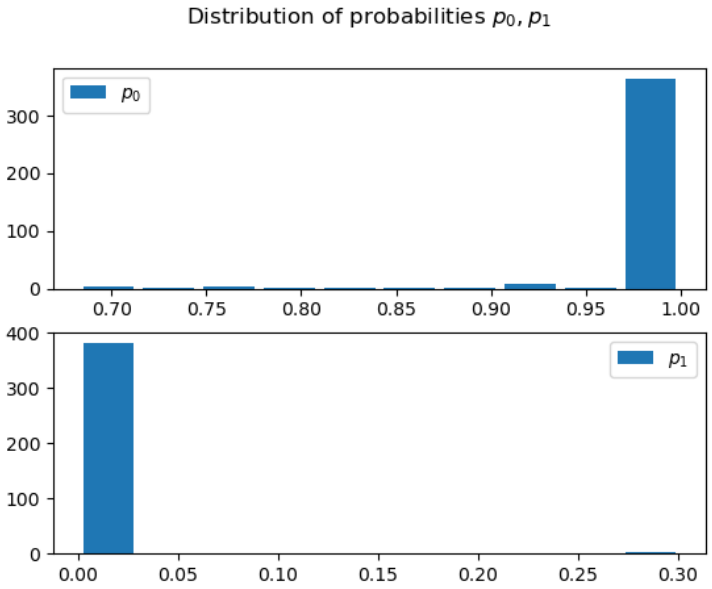


Figure 23: Distribution of the probabilities p_0 and p_1 after the last step.

It is evident that p_0 is very concentrated around the maximal value. This at the same time means that the distributions of the other probability genes must be very concentrated around 0, as is the case for p_1.

Figure 24 shows the population of microbes and bacteria over time.

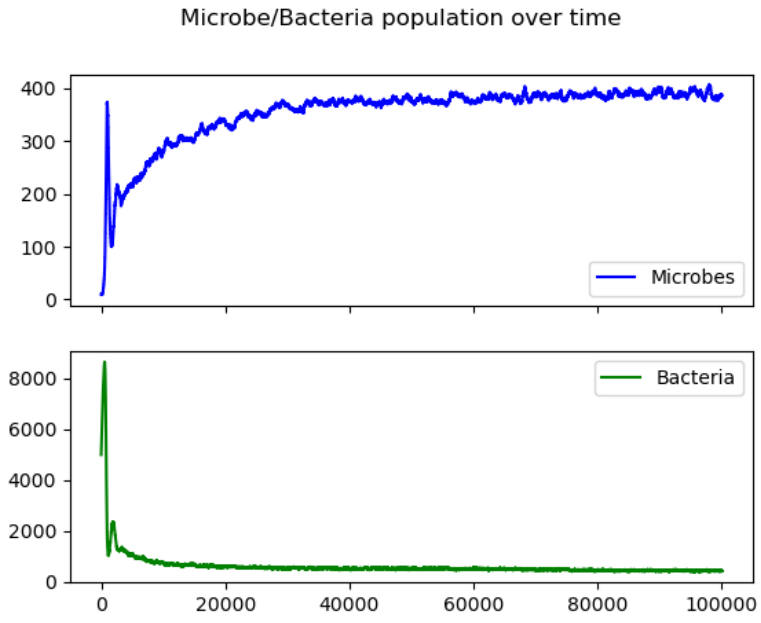


Figure 24: Microbe and bacteria populations over 10^5 steps.

The beginning of the simulation shows an interesting pattern. It shows the initial proliferation of microbes that was seen while following the simulation visually. To have a better understanding of the changes in population at the beginning, the simulation was rerun with the same seed for 3000 steps. The results are illustrated in Figure 25.

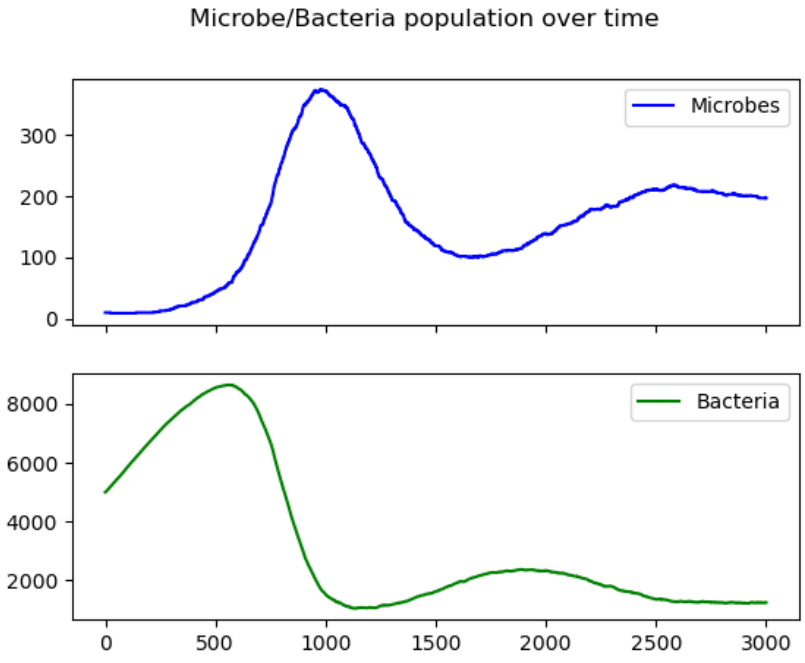


Figure 25: Microbe and bacteria populations during 3000 steps.

The population of bacteria rises with a constant slope of about 10 bacteria per step until about step 500, where the population of microbes becomes significant, and then plummets. The population of microbes rises exponentially until suddenly around step 1000 there are not enough bacteria left and the microbe population drops while the population of bacteria slightly rises again. The microbe

population then starts to rise again more slowly around step 2500 but without such a devastating impact as before.

It is interesting to note in Figure 24 that as the microbes become better adapted to the environment, the population returns to similar numbers as when there was the sudden overflow, as illustrated in Figure 25, and remains stable. The population of bacteria reaches even lower numbers than after the plummet around step 1000, when there was too few resources for the large microbe population to survive.

4.2.2.2 Distribution of Bacteria with the Square Pattern

The simulation was run using the same parameters as above, however with the Square pattern for generating bacteria illustrated in Figure 20. With this pattern it is more common that the microbes die out before they are able to proliferate.

The proliferation starts once one or more microbes have made it to the center, where there is an abundance of bacteria. Most microbes stay around the center while a few also seem to travel straight and reach the edges of the world where bacteria start to collect. Figure 26 shows the described state of the simulation.

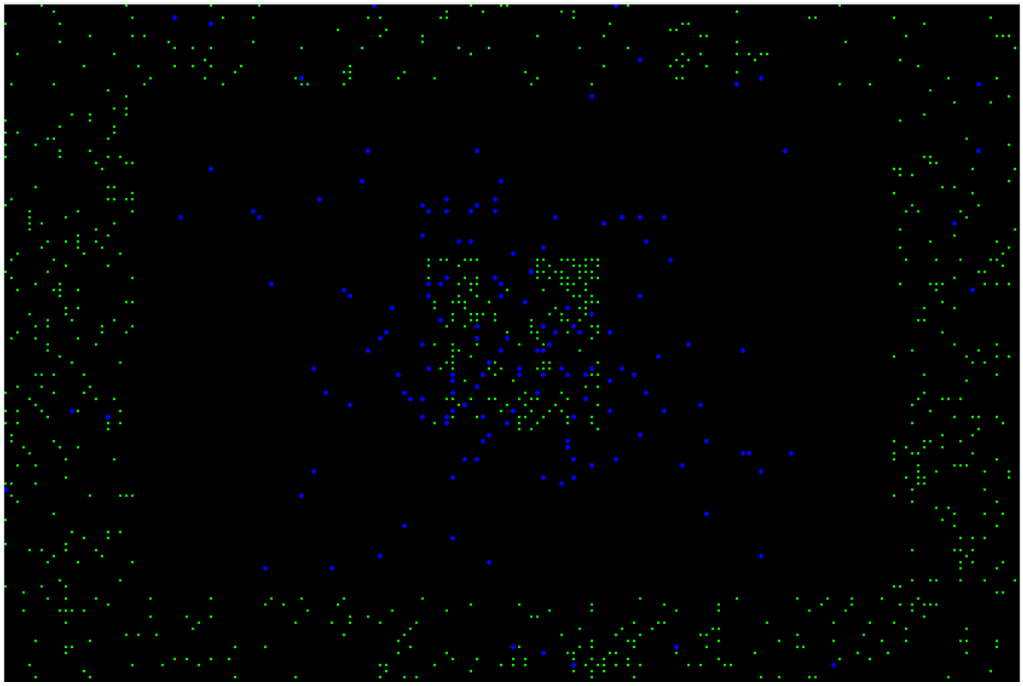


Figure 26: Screenshot of the state of the simulation using the Square pattern.

The populations of microbes and bacteria undergo similar patterns of overpopulation at the beginning of the simulation as when running the simulation with the Even pattern. Figure 27 illustrates the population of the bacteria after only 10^4 steps to be able to see the trend at the beginning.

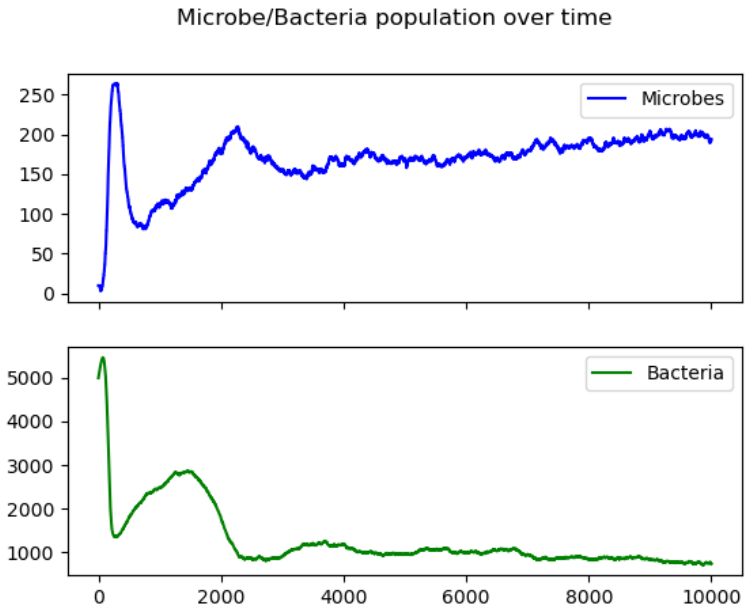


Figure 27: Population over 10^4 steps with the Square pattern for generating bacteria.

The environment houses a much smaller population of microbes, despite the same number of bacteria being available at the start and the same amount being generated after each step. Thus, the microbes are less efficient, at least in the earlier stages, and presumably it is more common that a microbe dies due to not finding any bacteria. Indeed, the average ages of microbes are about twice as high with the Even pattern than with the Square pattern after running both for 10^4 steps.

For the microbes to be able to stay close to the center, as they do in Figure 26, they must have a high probability of changing direction. The probability genes p_1-p_3 and p_5-p_7 would have to be responsible for such behavior. Surprisingly, after running the simulation for 10^5 steps only p_0 and p_7 had relevant values after consulting the data. The remaining probability genes had negligible values, as with the Even pattern, although they persisted much longer. Figures 28 and 29 show the averages of the probabilities after 10^5 steps and the distributions of the probability genes p_0 and p_7 after the last step.

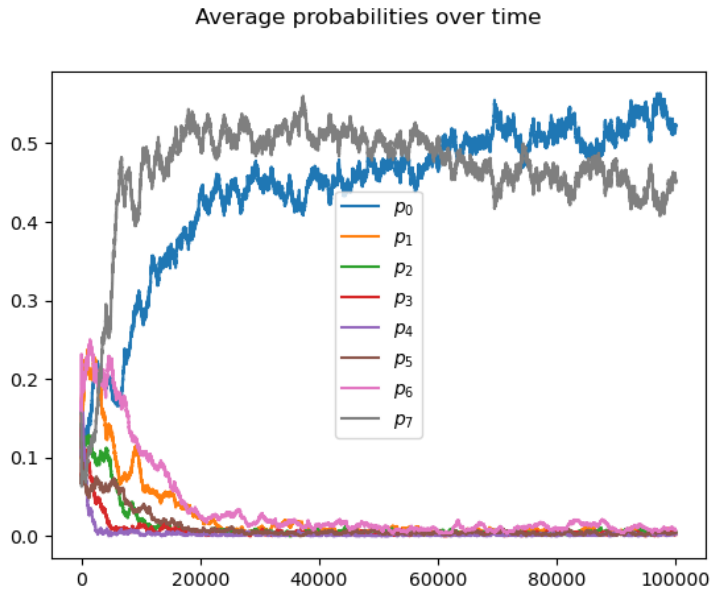


Figure 28: Average values of the probability genes p_0-p_7 during 10^5 time steps with the Square pattern.

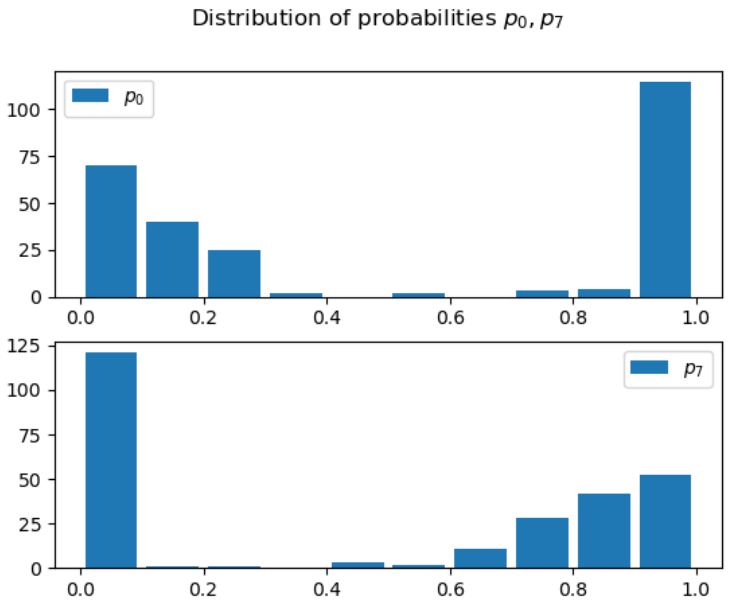


Figure 29: Distributions of the probability genes p_0 and p_7 after the 10^5th step using the Square pattern.

In this case, slight turns to the left are favored for staying close to the center. This raises the question why slight turns to the right are not also favored. Running the simulation multiple times shows that this can also be the case for slight turns to the right, but never both. Presumably, the fact whether right or left turns are favored depends on which first few microbes are able to proliferate and where they first enter the square region and only p_1 or p_7 (not both) ends up being dominant. After running the simulation for 10^4 steps 50 times, both p_1 and p_7 were favored 50% of the time.

Figure 29 shows an interesting symmetry. Because the values of the probability genes must sum up to 1 and the genes p_1-p_6 have values close to zero, the microbes that fall into the rightmost column in the distribution of p_7 must be the same microbes that fall into the leftmost column for the distribution of p_0. The same follows for all of the columns in Figure 29.

Thus, there are two types of microbes that are able to survive in the environment. There are microbes which have a higher probability to slightly turn to the left. These are clearly grouped around the center of the world as in Figure 26. They are most abundant with the p_7 gene close to 1. However, a significant (and declining) amount can also survive with smaller values.
The other type of microbes are the ones that fall into the leftmost column in the distribution of p_7 in Figure 29. From these microbes, only the ones with extreme values and thus a very low probability to change direction are able to survive. They presumably profit from the absence of other microbes on the edges of the world and from the abundance of bacteria in the center.

The microbes of the second type could not yet be seen in the state of the simulation in Figure 26. Figures 28 and 29 show that as time passes, a high probability to turn close to the center ceases to be the most efficient strategy. The averages for p_7 in Figure 28 gradually sink after rising quickly from the beginning and p_0 rises. Although this shift occurs very slowly, it eventually becomes noticeable in the visualization. These two types of microbes that follow from Figure 29 have very different strategies and both are strongly specialized to the environment. Figure 30 shows the state of the simulation after letting the simulation develop significantly longer than the one in Figure 26.

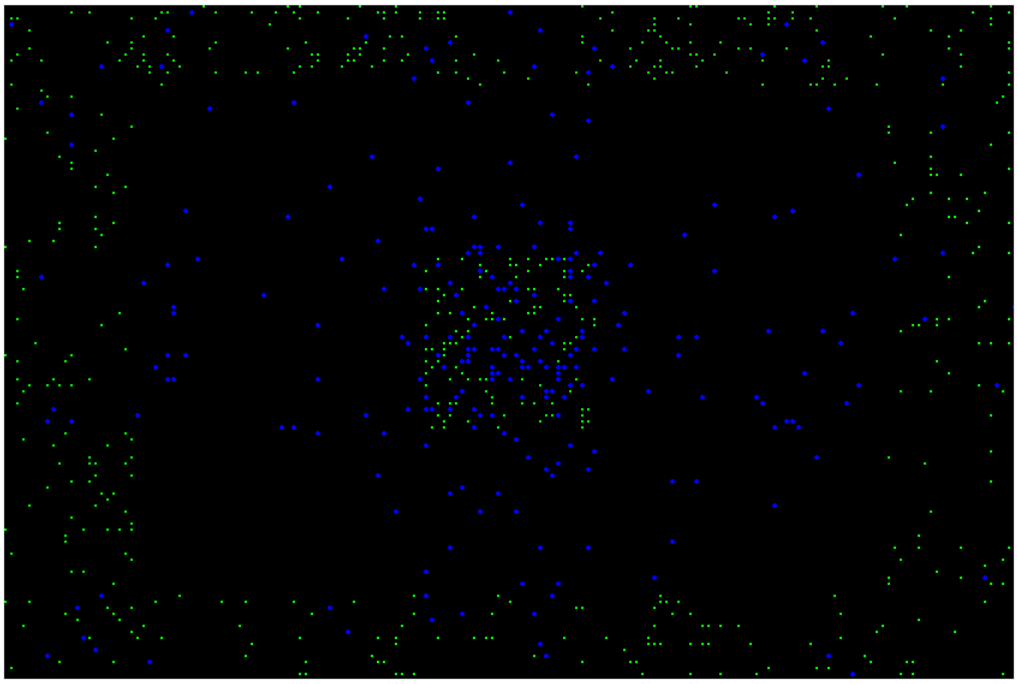


Figure 30: State of the simulation using the Square pattern after a long amount of time.

There is a significant number of microbes left and right of the center square and above and below it. This stands out especially in comparison with Figure 26. These microbes are travelling horizontally and vertically on paths which cross the square of bacteria in the center. These microbes have adapted to travel in a straight line through the square and must be the microbes of the first type with a low probability to change direction. Thus, this strategy is the most efficient with the Square pattern. There are also few microbes travelling straight diagonally and a small population circumnavigating the center, the second type describe above.

The total population of microbes rises very slowly over time, reaching about 250 microbes after 10^5 steps. Thus, the microbes are significantly less efficient than in the Even pattern. Presumably, this is because most microbes around the edges of the world are not successful and the bacteria there cannot be consumed efficiently.

When running the simulation first with the Even pattern and then switching to the Square pattern, the microbes have no trouble adapting to the new environment. Figure 31 illustrates the results obtained from running such a simulation for 10^5 steps, starting with the Even pattern, and switching to the Square pattern halfway through.

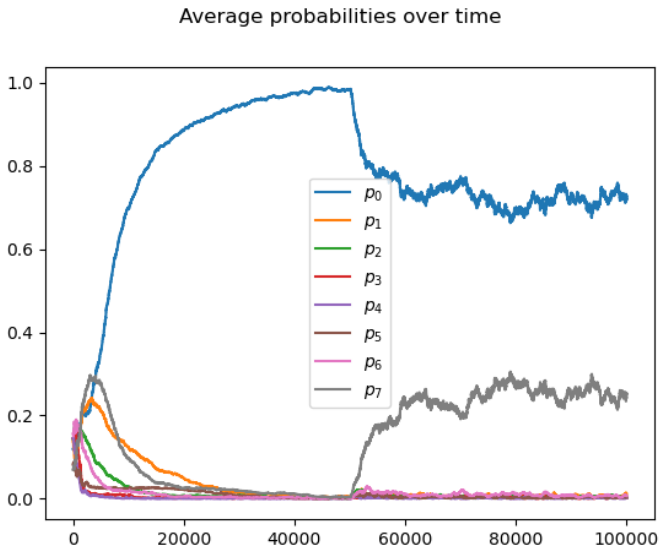


Figure 31: Average values of the probability genes during 10^5 time steps, starting with the Even pattern and switching to the Square pattern halfway through.

Relative to the total time the simulation was run, the microbes adapt very quickly to their new environment. When comparing the results with Figure 28 the following stands out: the value of p_7 stays far further below that of p_0. Running the simulation multiple times yield similar results, with either with p_1 or p_7 being favored around the center. The probability values in Figure 31 seem to have stable values and there is no clear trend of p_0 or p_7 rising or falling after about step 80000. Consulting the distributions for the simulation of Figure 31 shows that the only notable difference between the microbes in Figures 28 and 31 is that in the latter the population of microbes turning is significantly lower and the population of microbes not turning is larger. This suggests that the population of microbes with a high value for p_0 that travel through the center severely limit the population of turning microbes and are clearly more efficient.
This can be further shown by comparing the populations of the two scenarios. When running the simulation as for Figure 31, the population of microbes is notably higher as compared to running it normally. In the first case the microbe population consistently ends up at around 300 microbes, while in the other case it ends up closer to 250.

The trends of the probabilities genes p_0 and p_7 in Figure 28 show no sign of flattening out and perhaps if the simulation would be run for a very long amount of time, the microbes would eventually arrive at probabilities similar to the ones in Figure 31.
Figure 31 demonstrates that the microbes are able to adapt to their environment quicker by switching patterns halfway through, which in this case would make sense as almost all microbes with the Even pattern have a high probability for p_0 and thus less modification is necessary to adapt.

4.2.2.3 Distribution of Bacteria with the Line Pattern

Again, the simulation was run with the same parameters, however now using the Line pattern for distributing bacteria. Remarkably, the microbes are able to adapt to this environment by mostly

travelling along the lines where more bacteria are generated. Figure 32 illustrates what this looks like in the visualization of the application.

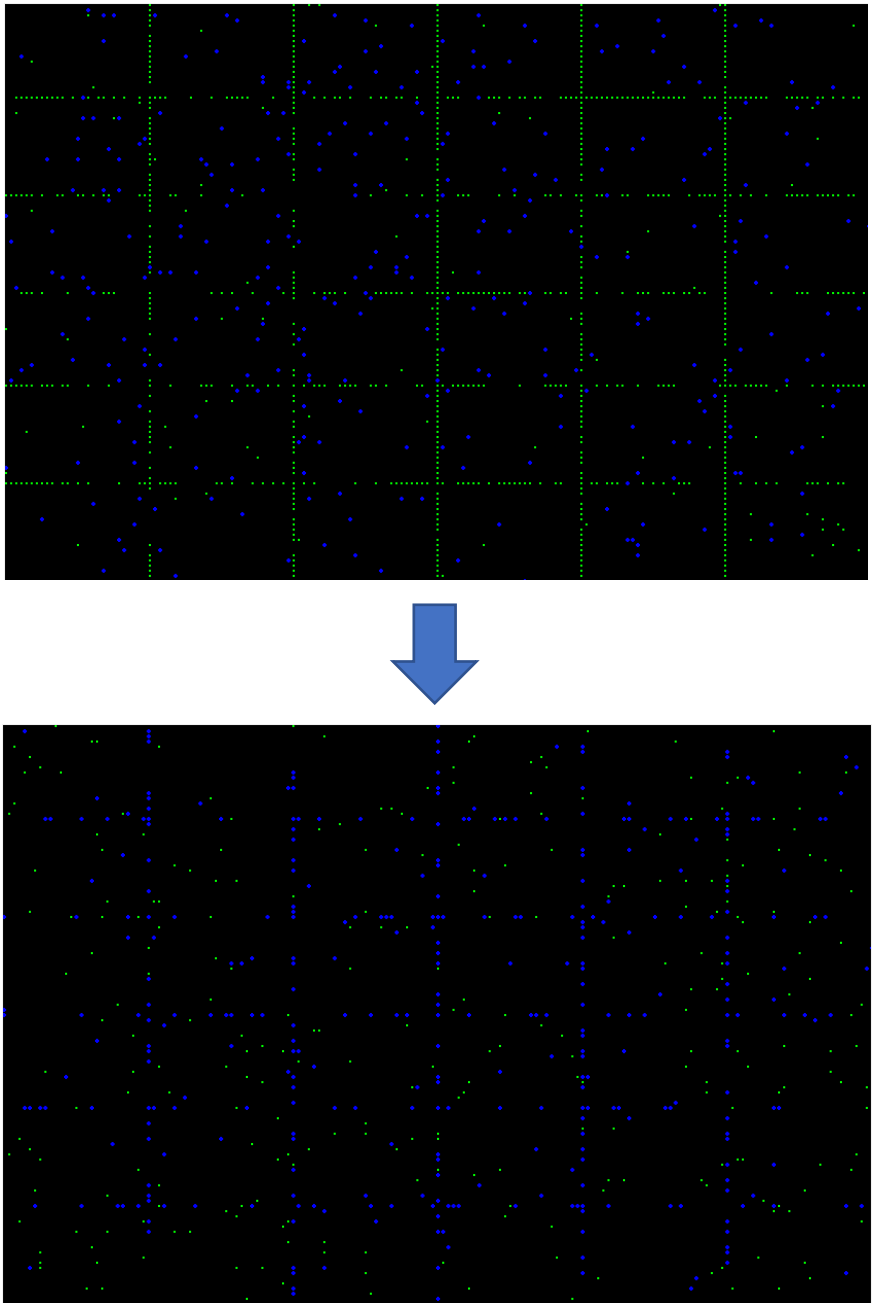


Figure 32: Two screenshots showing the state of the simulation before and after the microbes have adapted to the Line pattern.

Most microbes travel along the lines. But there are also some travelling diagonally across the world and eating bacteria from both the lines and the less fruitful regions between the lines. Probably the bacteria-rich regions cease to become worthwhile if there is too much competition.

The microbes opt for very similar probability genes in the Line and Even patterns. It is not possible to tell the two apart merely by looking at the averages or distributions of the probabilities. However, by looking at the distribution of the directions of travel of the microbes after the last step the big difference is apparent. Figure 33 shows the distribution of directions after running a simulation with the Line pattern for 10^5 steps.

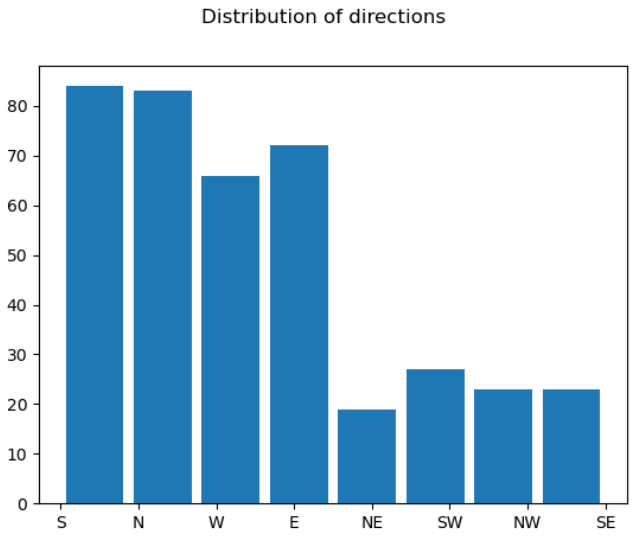


Figure 33: Distribution of the directions of the microbes after 10^5 steps with the line pattern. The directions are labeled as on a compass on the x-axis and the y-axis specifies the frequency of each direction.

Figure 33 shows that horizontal and vertical directions, which follow the orientations of the lines, are significantly more popular. There would be no benefit to travelling horizontally or vertically, unless the microbes are travelling on the lines. For microbes not travelling along the lines, the probability of encountering a bacterium is much higher if they can follow different paths, which is best possible by travelling diagonally.

One could argue that the distribution in Figure 33 might merely be a chance occurrence. Figure 34 shows the number of microbes travelling in each direction over time and provides even more compelling evidence.

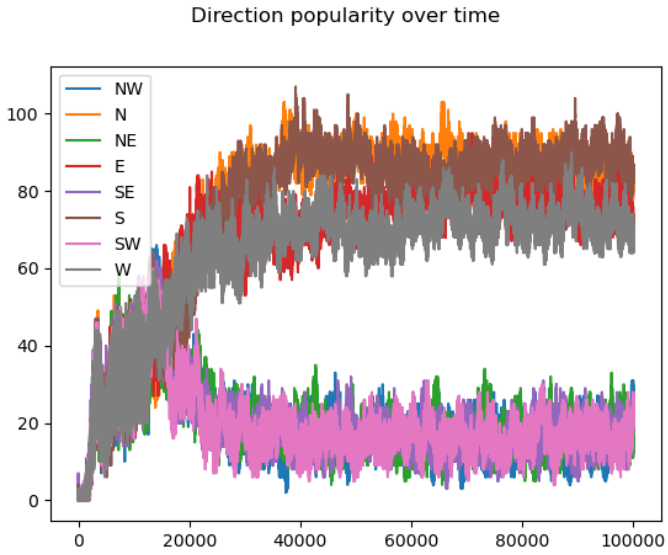


Figure 34: Frequency of all directions over 10^5 steps.

Although the data varies greatly, presumably due to deaths, births and improbable changes in direction, the overall trend in Figure 34 is clear. It takes a while, but gradually horizontal and vertical directions rise in frequency, while diagonal directions become less popular until both reach stable values. The results in Figure 33 are not the result of randomness. They are the result of a trend

which arises due to the distribution of bacteria along horizontal and vertical lines. Through natural selection, the microbes adapt to travel along the lines without ever changing direction, maximizing their chances of finding bacteria and minimizing their energy expenditure.

As in the Square pattern, a minority of microbes opt for a different strategy to find food. These microbes travel diagonally and collect bacteria on and between the lines. With many microbes travelling on the lines and few enough opting for this strategy, it can be worth it.
In evolutionary terms, these two species of microbes can be likened to generalist and specialist species in real life. The less popular species of microbes travelling diagonally would be generalists, collecting sustenance from all over the world, while the microbes travelling on the very specific paths of the lines would clearly be specialists. The latter are more successful but also much more dependent on the environment. If one would switch to the Even pattern, these microbes would surely be the first to die out, while the generalists would not need to adapt at all.

The development of the microbe and bacteria populations are significantly different than with the Even pattern, despite the similarities in the probabilities. Figure 35 shows the microbe and bacteria populations for the same simulation as before.

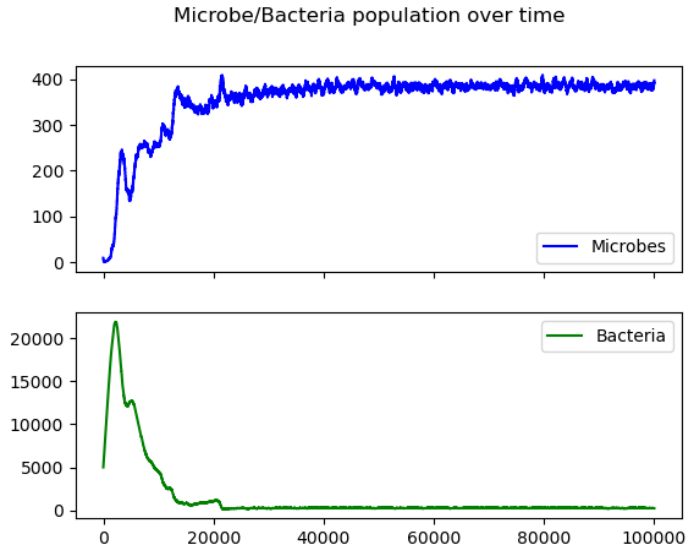


Figure 35: Population over time after 10^5 steps with the Line pattern for distributing bacteria.

The exponential rise in population at the beginning of the simulation reaches far lower numbers than with the Even pattern (compare to Figure 24), and thus also falls much less after overpopulation. Figure 34 shows that it takes a large amount of time for the microbes to adopt horizontal and vertical directions of travel. Presumably, the microbes are not able to proliferate to the same extent at the very beginning, because they are not yet well adapted and are probably not yet travelling along the lines. It is around step 15’000 that horizontal and vertical directions begin to overtake and dominate diagonal directions in Figure 34 and around the same time there is a steep rise in population in Figure 35. This could be the period when microbes begin to travel along lines and as more and more microbes populate the lines, the closer the population approaches about 400 microbes.
The maximum population of microbes achieved with the Even pattern was very similar. Although the microbes are very efficient in collecting bacteria along the lines, the bacteria generated more sparsely across the whole world are somewhat neglected and hard to collect efficiently, which is probably why the microbes are not more efficient overall. In comparison, in the Even pattern all

microbes are collecting bacteria the same way. But since the bacteria are distributed randomly the efficiency of these microbes is also limited.

4.2.3 Conclusions

The Microbe model has many desirable properties, especially when compared to the Primer model. The rules are much simpler and thus far less restricting. The model can be likened to Conway's *Life* simulation described in section 2.2. Because the world is a grid of cells, calculating steps is very efficient. The population of bacteria has no significant impact on the performance of the simulation and thus the relationship between the microbe population and performance is linear. This efficiency allows for the simulation of large populations over a long time and results much more compelling data.

Several connections could be drawn to real world phenomena.
Firstly, the microbes in the Line pattern exhibited behavior similar to specialist and generalist species in the real world. The specialists were far more efficient, but also far more vulnerable to changes in the environment. This phenomenon does not only apply to the Microbe model. It is a well-known fact of the real world and a great dilemma with global warming. Because specialists are so well adapted to specific properties of their environment, slight changes in temperature can have drastic and extinctive effects. The extinction of a specialist species often leads to the endangerment of other specialist species which relied on it and has many other consequences due to the complex relations between different species in an ecosystem. The species in the Microbe model are independent and do not manifest such an interrelation as there are very few species and all of these depend on a single external factor: the availability of bacteria.

Secondly, the fact that microbes would tend to favor either the probability gene p_1 or p_7 in the Square pattern, but never both, based on chance is applicable to the real world. The traits selected for by natural selection are not necessarily the most optimal and are subject to random factors of the environment.

Thirdly, these simulations demonstrate how although a population can suffer from a shortage of food due to overpopulation in an early stage, it is possible for the population to return to similar or even higher numbers and remain stable after becoming better adapted to the environment.

Further, it shows that with a large population, organisms are quickly able to adapt to new environments. Also, organisms coming from a different environment might even be able to better adapt than other organisms which have been in the environment much longer, as was the case when switching from the Even pattern to the Square pattern.

The model was not explored to its full extent and there would still be many interesting scenarios one could test using the Microbe model.

5 Conclusion

With the right model, valuable lessons can be learned by simulating evolution. The Microbe model provides a glimpse of how a simple model can reflect many, important natural phenomena. There are whole subfields of science, often closely linked to Computer and Data Science, in which many, far more sophisticated models have been developed for studying specific and more complex aspects of evolution, or for applying it to real world problems. The evolution of social behavior is studied in the field of Evolutionary Game Theory [11]. In the field of Evolutionary Computation, the principles of evolution are used to find close to optimal solutions to difficult algorithmic problems [12].

The models analyzed in this paper were more proof-of-concept simulations, which deemed simple enough to integrate into the program and analyze. An interesting extension would be to integrate a more comprehensive model into the program and compare the data with results obtained from other sources.

The program served its purpose of running and visualizing simulations and worked reasonably well. It was able to host not only one, but two very different models of evolution and provide meaningful insights into the models and simple aspects of evolution in real life. Moreover, because of the hierarchical structure of the program it was easy to write scripts to run simulations programmatically and over longer periods of time.

However, the program is far from being perfect. The graphical performance of the Tkinter canvas is very poor speed-wise with many agents in the simulation. This became especially apparent with the Microbe model. Further, using a different size for the application window changes the size of the world in the simulations, which has an impact on the data. Other aesthetic properties are also missing from the graphical interface. Programming the plotting of the data for a model is tedious and could do with some automation. The overall legibility of the code is satisfactory, however there is still room for improvement.

Overall, the program is not much compared to some of the more powerful professional tools available for analyzing evolutionary models. However, it served well as the object of a Matura project. The source code for the program is publicly available online on GitHub[2].

[2] https://github.com/HumbleCatcher/A-Simplified-Simulation-of-Evolution

6 References

[1] "Dictionary.com," [Online]. Available: https://www.dictionary.com/browse/simulation. [Accessed 23 December 2021].

[2] J. Helps, "YouTube," 15 November 2018. [Online]. Available: https://www.youtube.com/watch?v=0ZGbIKd0XrM. [Accessed 20 January 2020].

[3] M. Gardner, "Mathematical Games - The fantastic combinations of John Conway's new solitaire game "life" - M. Gardner - 1970," Universität Potsdam, October 1970. [Online]. Available: http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.htm. [Accessed 20 January 2021].

[4] "Python interface to Tcl/Tk," Python.org, [Online]. Available: https://docs.python.org/3/library/tkinter.html. [Accessed 1 February 2021].

[5] "concurrent.futures — Launching parallel tasks," [Online]. Available: https://docs.python.org/3/library/concurrent.futures.html. [Accessed 5 February 2021].

[6] J. Palfree, "minutelabsio/evolution-simulator," 2019. [Online]. Available: https://github.com/minutelabsio/evolution-simulator.

[7] "matplotlib," [Online]. Available: https://matplotlib.org/. [Accessed 1 February 2021].

[8] J. Palfree, "Evolution Simulator," MinuteLabs.io, [Online]. Available: https://labs.minutelabs.io/evolution-simulator/#/s/1/viewer. [Accessed 21 January 2021].

[9] I. Berg, "Simulated Evolution," [Online]. Available: https://beltoforion.de/en/simulated_evolution/. [Accessed 28 January 2021].

[10] I. Berg, "beltoforion/Educational-Javascripts-Typescript/simulated_evolution/," [Online]. Available: https://github.com/beltoforion/Educational-Javascripts-Typescript/tree/master/simulated_evolution. [Accessed 28 January 2021].

[11] "Evolutionary game theory," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Evolutionary_game_theory. [Accessed 1 February 2021].

[12] "Evolutionary computation," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Evolutionary_computation. [Accessed 1 February 2021].