2019

Hunter gatherer simulation

ARTIFICIAL LIFE AGENTS COMPETING IN A 2D WORLD

AXEL WICKMAN

729G43: ARTIFICIELL INTELLIGENS

EXAMINATOR: ARNE JÖNSSON

Abstract

In this project an artificial life simulation was built in order to demonstrate and explore the dynamics that emerge from the rules of this kind of system. The simulation consists of humanoid agents moving in a 2D world, able to eat food to sustain themselves, and being able to reproduce asexually. Along with the graphical simulation, tools were added in order to be able analyze the system statistically. This report explains the motivation behind the project, how it was made, how it can be used, and lastly provides an analysis of the systems common behaviors.

Introduction

The ability to run models and algorithms on our computers provides an opportunity to understand some of the fundamental mechanisms of our surroundings, and why they arise. By building simulations, a programmer can observe behavior of systems given varying levels of complexity of the rules and preconditions of this system. Depending of the goal of the researcher, the purpose of the simulation can be to closely approximate the real world, to only mirror some essential process of the real world, or to model something entirely theoretical. In which case, the simulations can bring attention to complex interactions and emergent behavior that otherwise wouldn't be obvious through purely mathematical modelling. This project is a free exploration of how to build a system which gives rise to this behavior.

Background

Artificial life

Understanding human and animal behaviors is of large interest to many fields of science. While a lot of behavior can be understood through models of psychology and reward mechanisms, some of it stems from our evolutionary past. To understand this, one approach is to simulate the evolutionary process for virtual agents in a virtual environment. Most commonly this uses the principles of natural selection; discards what doesn't work, elaborate on what does. The progression/evolution of this kind of system is a tendency for what is being elaborated on to move towards states with the properties that makes them stable. For virtual agents this is typically behaviors which promote survival and reproduction, which can be defined or incentivized by the programmer designing the system. What is interesting about this system isn't only what has to be done to survive, but also the often complex dynamics of the behaviors that help to accomplish this.
In the famous Conway's Game of Life, emergence and self-organization can arise from rules that are very simple. From only three rules (governing survival, births and deaths) this system is capable of states that not only are entirely unpredictable, but also are capable of simulating Turing Machines (Rendell, 2001), as well as self-replicating structures (dvgrn, 2013). Self-replication in systems is one way that competition can arise. Competition can give rise to certain traits being selected for by applying selective pressure to some parts of the system. Natural selection is an example of this, and so is competition between companies. Another example where competition doesn't equal existence of a self-replicating structure, but rather stability of the system, is a modern, today commonly used within machine learning type of network called General Adversarial Networks (Google, 2019). Here the competition is between a generator and a discriminator network, which both try to outsmart each other, giving rise to better models.

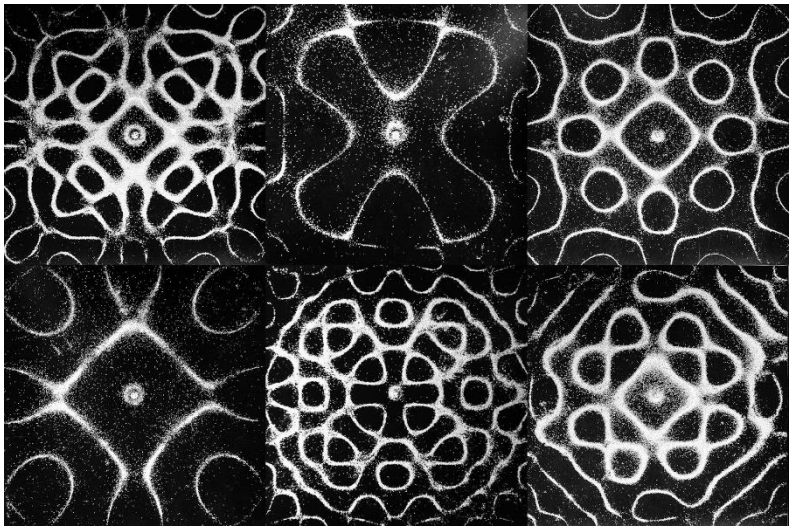

Figure 1. Day 19: Chladni Plates, by Chris Smith, 2013.

Example of a natural selection system. Grains of salt are placed on a paper with a loud speaker below. When tone is played, some parts of the paper oscillates, while some are at points of equilibrium. The grains of salt, which were originally evenly distributed, bounce in a random direction if they are resting on an oscillating part of the paper, but otherwise stay put. The grains in "good" positions are kept, while the grains in "bad" positions are resampled. From this process coherent patterns emerge.

Artificial life has been used as a method for inquire within for example the field of biology. Taylor and Jefferson (1993) provides a comprehensive summary of how artificial life in the form of software, hardware and in the lab can be used to understand the complexities of living systems on the level of the molecular, cellular, organism and population-ecosystem. The molecular level provides insight about the grey-area between living and non-living systems. When can a reaction involving an RNA-molecule be seen as fulfilling a purpose and not just an accident? This view of *chemical evolution*, which considers how simple molecules can self-replicate, is distinguished from *organic evolution*, which considers the replication of already organized, clearly defined units. This kind of evolution can consider life on an organism level, with single individuals consisting of these units having emergent behavior (functionality of control) to perform tasks using some kind of low level reasoning. Organic evolution can also consider life at the population level, where the clearly defined units instead are coexisting individuals with their own incentive to spread their own genes. This kind of system produces a kind of ecosystem, with emergent intelligent agent behavior arising from competition and symbiosis, as well as evolutional phenomena. In this project, the decision was made to investigate this kind of artificial life system.

Purpose

The purpose of this project is to build a system simulation that through the process of natural selection and competition between artificial agents create emergent behaviors. The overall nature of these behaviors should be investigated though mainly qualitative observations of the system, over quantitative observations.

Method

Environment design decisions

To fulfill the purpose of the study the environment should be constructed in such a way that it allows for the competition between agents, while also delegating the decision reproduce and other behaviors to the agents and the evolutionary process itself. This means that the agents must live in the same world in order to interact – a multi agent system. This also rules out using a genetic algorithm that programmatically selects and reproduces agents according to certain behaviors. Instead, a more hands off approach was chosen, where a natural selection process emerges from the environment controlling the conditions of survival and reproduction. It therefore makes sense to use an environment which is both continuous (for both time and space) and sequential, where each agent has to make decisions based on non-discrete input throughout its life, where previous actions affects later outcomes. This means that small changes in decisions can have a large effect on later outcomes, and that this system is chaotic and that the environment therefore in practicality is stochastic (Russell & Norvig, 2009, ss. 42-44).

Agent design decisions

The agents were chosen to have only partially observable percept of their surroundings, and while actions are performed constantly the environment is technically static because a new percept is calculated, and a new decision is taken between every frame with environment not changing during this decision time. A classical shallow neural network is suitable for agent decision making. Among other reasons, this is the case because they are universal function approximates, in theory being capable of calculating any mathematical function (behavior) necessary for the agent's genes to last (given sufficient size). Usually neural networks are trained using supervised gradient decent algorithms. This is however not an option here since the optimal outputs for the network aren't known. The network structure and weights are therefore determined exclusively from the agent's genes and are thereby controlled by the natural selection.

The agent's percept is configurable. The percept values always range between 0 and 1. The agent can have six different modalities to sense:

- Collision – if the agent is colliding with the wall.
- Visual receptor – lines orientated at different angles to the agent, which increases in value if the line intersects any object in the world.
- Color – red, green and blue value that corresponds to the average color of what the visual receptors intersect with.
- Energy level – Ration of current energy level to maximum possible energy level.
- Mushroom count – Ration of current mushroom count to the maximum possible mushroom count.
- Memory – values that change with a certain reactivity towards a value of a corresponding network output value. This allows for recurrence in the neural networks.

Each agent can perform at most six different actions, depending the configuration and on if conditions allow it. Actions that constitute instantaneous events have a cooldown-period where they can't be triggered again. The possible actions are:

- Reproduce – “willing” if activation above a threshold. Needs at least 60 in energy in order to be able to do this. Energy between the parent and child are split according to set proportions.
- Walking – at certain speed and backwards if the value is close to zero. There is an energy penalty in proportion to the absolute speed of the agent.
- Turning – Left or right. Turning rate is the delta between two output perceptrons. Energy penalty in relation to the turning-rate.
- Eat – consume one mushroom from the inventory to turn it into energy. Energy from eating is capped at set level.
- Place – place one mushroom from the inventory in front of itself in the world.
- Punch – Remove energy set amount of energy from all other near agents, with set energy penalty.

Implementation

General programming

The software is required to simulate and render many elements, which means there is a risk of hitting performance bottlenecks. To mitigate this, C++17 was chosen as the main programming language because of it being performance orientated and providing low level control over the hardware, while still having many modern features to speed up the development process. Compiling was done with the GCC MingW compiler, and CMake for managing the build process. C++ has extensive support for object oriented programming. This is most commonly accomplished by creating one header file, where all the class declarations are stored in such a way that it is clear to the programmer and compiler how to use the classes in the file. Then there is a source file, which contains all the definitions, or actual code, for the classes that were declared in the header. One class can access and use another class by telling the compiler to include the other header file, which makes the header file’s scope accessible, and makes it possible to instantiate an object of the class and call the member functions inside it. This means that there usually is a hierarchical organization of ownership between the classes, with the “main” function at the top, acting as a starting point for execution of the program.

There are times when one object must access another object which it doesn’t own. This can be done by the object previously having been given some kind of reference (a pointer, smart pointer, or a reference to the object) which has been stored, or by accessing a third object which knows about the desired object. This last technique is for example used when the *WorldObject* class needs to check the config file, which can be globally updated during runtime. Every *WorldObject*-object has a pointer (a variable with a memory address as a value) to the World-object which spawned them, which they can as for the Config-object using a getter function, which is returned and read. Another case is when an object is managed by a shared pointer (a kind of smart pointer). In this case it practically is owned by all objects which knows about it and is deleted when it is no longer referenced by anything else. This is for example used when an agent dies but has been selected by the user and therefore still referenced by the GUI-object. In this case it isn’t deleted from memory until the user deselect the agent.

Rendering

The output of the program consists of the prints produced in the terminal, as well as a window with a real time, interactive rendering of the world and statistics about it. To create one of these windows require interaction with the operating system, and a lot of low level programming for graphics and input. This is therefore handled by the library SFML, which has several advantages. It provides a high level interface for creating windows, graphics, audio and networking. It also provides interoperability between operating systems, which means the same code can talk to Windows, Mac and Linux. On top of this, SFML provides utility classes and functions, which speeds up development and decreases complexity of the code. One commonly used example of these is the *sf::Vector2<T> class*, where the "T" can be different numerical types such as floats and integers. This class is convenient for storing two numerical values in one variable and provides rudimentary but very useful vector algebra member functions.

Rendering isn't accomplished through one specific class in the project. Instead, all objects which can be rendered have a *draw* member function, which is called every frame with a pointer to the SFML window object and a float of the passed time as parameters. While rendering is executed sequentially with the rest of the program, which for simplicities sake is single threaded, there happen at the same time as OpenCL is processing agent decisions. This means that there on most systems is a double load on the GPU, but this likely isn't a problem because this isn't a processing bottleneck.

Quadtree

Because the system needs to calculate collisions between objects in the world, and there will be hundreds of objects which can collide every frame, a smarter solution is needed than simply calculating the Euclidian distance between every object constantly. This solution is not viable because it has a time complexity of $O(n^2 - n)$, and quickly becomes too slow for even a modern computer running C++. Instead, a Quadtree is used to only calculated the exact distance between object which are already known to be close to each other in memory. This tree is a kind of container with shared-pointer references to all world objects which can collide world objects. These world objects have themselves references to the Quadtree and can perform lookups in to get a container of all other close objects. The tree structure is recursive, with the top tree rectangle encapsulating the whole map. If two objects are in the same rectangle, that rectangle is then split into four quadrants, and the reference to each object is moved down to their respective subtree. This process continues until every object has its own rectangle, or the subtree size limit is reached. Every time an object moves outside the current subtree its reference is moved up the tree until its position is once again within the current tree's rectangle, where it is then inserted.

Genes

Every agent has genes which defines their characteristics. The implemented system for managing these genes is very general (which provide flexibility), and does not differentiate much between genotype (the genetic structure) and phenotype (the resulting characteristics). This has the disadvantage of not allowing for modularity and regularity (Huizinga, Mouret, & Clune, 2014) arising from genetic structure but has the advantage of not adding extra complexity and uncertainty in the simulation. The gene representing the number of layers in the network is for example simply

represented as an integer number. There are five different types of gene-classes all inheriting from the same abstract base class Gene. The numeric types FloatGene, and IntegerGene stores one value of their respective datatype which can vary between a given range. The LambdaGene is a so called template class, meaning that it can store any kind of value, which is then evaluated using a user-defined lamda-function. The map genes are containers which have strings as keys and any kind of Gene as value. These are used to give names and easily look up values in the genome. The last kind are ListGenes, which clone a given template Gene either a static number of times, or the number of times defined by an IntegerGene.

Using these types of classes it is possible to recursively define the structure of a genome, which values then can be easily generated and mutated randomly as well as cloned. The significant advantage with this is that the system automatically adapts the number of generated random numbers. This can be seen in the code snippet in Appendix B, which shows how the system is used. The system will generate a new completely random layer, the layer count increases with one. After generation the values can be accessed through a function where needed to then actually translate the genes into characteristics.

Agents AIs

The agent AI consists of a traditional feed-forward neural network (also called a multi-layer perceptron) which is ran every frame. This is a large computational workload, but the running of these neural networks do not depend on each other in between frames. This means that it is possible to perform this task using the GPU, which is designed for running computations on its thousands of cores. If a given problem is suitable, which it is in this case, it is possible to get multiple times the performance. It is however required to write special code to program the GPU. The choice was made to use OpenCL for, mainly because it is a low level language with little performance overhead, and because it is a standard that automatically works on mainstream GPUs and also CPUs (which works as a fallback in case there is no GPU).

To implement this, an OpenCL-wrapper object is told every time a new agent spawns. This new agents genes is then translated into memory buffer representations of the neural network, and this data is written to the GPU. These buffers are then manually allocated when the Agent-object is destroyed (because the agent died, the program shut down, etc.). Every frame, after world physics simulation but before rendering, a loop loops through every agent, first updating its percept, then asking the OpenCL-wrapper to think for it. This writes the percept data onto a buffer specific to the current agent, called netActivationA. The GPU is then instructed to start a kernel, which is the function that runs on the GPU, which computes the neural network activations of the next layer given the network and the previous layer. As a parameter, the kernel is given netActivationB as the buffer to write the output activations to. This is simply an instruction for the GPU to start the kernel, and doesn't happen immediately, and the main program doesn't wait for it to finish. Instead it goes to the next layer and queues an instruction to compute this layer when last layer is done, but this time with netActivationB as input buffer and netActivationA as output buffer. This procedure is repeated, with the buffers flip-flopping, until the last layer is reach. The last instructions for this agent is to first to get the last activations buffer's data from the GPU, and then call a callback to the Agent object, setting its computed actions. This is then repeated for all agents, every frame. At the start of every frame, the main program freezes and waits for all GPU computations to finish up, after which it can apply the actions.

As activations the network is fed values ranging between 0 and 1 from the percept. The first hidden layer perceptrons then first multiplies previous layer activations with the connection weights, and sums them. The previous layer's bias is then added, and the value is passed through the arctangent activation function to provide this layer's activations. This is then repeated for all hidden layers and the output layer. The output layer activations are then passed through the sigmoid function (on the CPU) to get a value between 0 and 1 to be used as an agent action. The default network values can be found in the configuration file (Appendix C).

Agent colors and names

Agent colors was added as a tool to track which agents are related and how much their genome is changing, every agent is given a color and a name generated from their genome. The agents' colors also allow agents to see the species species of other agents if color vision is activated. Because the names and the colors need to reflect differences in the whole genome, they aren't generated from specific genes, but rather the value of all numeric genes are normalized and collected into a vector. This vector is then split into several parts (depending on how many values are needed) and an average number is calculated for each part. These averages are then used as inputs for generating the colors and names. The color comes from rescaling three extracted values linearly and simply making these the red, green and blue components.

For constructing the names, a Markov chain is used. This Markov chain is built from a database of 100 000 American surnames and saved to a json file using a Python script. This json file is loaded into memory when the program starts. The model has a max lookup of 4, meaning that it will look at the previous 4 characters (at most) both when generating the chain and choosing the next character. When generating the characters, all previous instances where the same lookup characters have appeared are considered as alternatives. To make the decision the numbers extracted from the genome are used. This system means that children most likely will have similar colors and names to their parents, but that unrelated agents will have vastly different characteristics, and also that small changes in the genome have a small and for names probabilistic effect.

Populating the world

Agents and mushrooms have a chance of being spawned in the world every frame if their current count doesn't exceed their respective predefined limit. This is done by interpreting these events as random and independent, and using a poisson distribution and a given rate multiplied by the frame time to generate a random number which represents how many of this object should spawn this frame. This creates a forever lasting supply of food resources for the agents, while still allowing for food running out in the short term. Mushrooms can reproduce if their total count is less than the limit and there aren't too many mushrooms in the local vicinity. Every frame every mushrooms as a certain probability of spawning a new mushroom somewhere near itself.

Agents can also reproduce at will, when certain conditions are met. This isn't dependent on the current count of agents. This means that the agent count can exceed the limit, although population explosions aren't problem because of the limited resources in the world at any given time. Any sudden rise in population count will lead to there being less food to eat, and the agents starving, thus balancing the system.

Analysis tools

In order to get a grasp of the complex state of the simulation, as well as changes during the simulation, a number of tools was added into the graphical user interface. In the lower left corner, the current number of frames per second, time factor, agent count, and mushroom count are listed. This gives a very basic overview of the whole world. A more in-depth population screen can be acquired by selecting a specific agent. This will show the agents name in the agents color (clicking the name will make the camera follow the agent), an energy bar, a representation of the percept vector, a representation of the action vector, and text information about network layers, perceptron count, age, generation, children, murders, and mushrooms in inventory. Each perceptron in the percept and action vectors can be clicked, which changes the values of the other vector to represent their on-line correlation with the selected perceptron.

On the right there is a list of toggleable settings that has some effect on the simulation or the GUI. The “agentSpawning” toggle turns on and off the process of agents being populated into the world. The “reloadConfig” toggle switches off directly after being triggered, effectively making it a button, which updates the world configuration depending on the config-file. The toggles “showWorldObjectBounds”, “showQuadtree”, and “showQuadtreeEntities” are all used to debug and make sure the object collision detection is working correctly. “showVision” shows the agents visual receptors and is also helpful for determining agent direction. “showPaths“ renders the walked paths of all agents, which is also done regardless if the agent is selected. The path is a dashed line where every dash represents 1 simulation second graphLine” and “graphSpectrogram” are mutually exclusive toggles, which both have sub-toggles (revealed when hovering), the latter of which’s are also mutually exclusive. These are described in more detail further down. “renderOnlyAgents” prevents rendering of the world. All toggles starting with “visualize” are mutually exclusive, and sets the agents opacity proportional to their corresponding attributes, effectively only showing agents where these values are high. “showSquare” renders a coloured square which also can changes opacity and makes all agent more visible.

The line graph (toggle “graphLine”) renders color-coded normalized line graphs for each of the activated sub-toggles. The x-axis is time and the y-axis is the attributes value. Note that there is currently no smoothing or simplification of the curve, which can cause the program to run slow and change the agent behaviour, if not viewed while the program is paused.

The spectrogram renders the distribution of the values (y-axis) of certain attributes over time (x-axis), where the agents’ colours are used to indicate which agents are where in the distribution. Where there is overlap, the average colour is instead shown. This tool give much insight into the evolution of the system, and is useful for identifying what are the dominant species, how they have changed over time, how long a certain species have existed, and the genetic diversity within species. Unlike the line graph, the spectrogram is robust to be viewed while running the simulation, even after recording data for a long time.

Usage

Building on Windows

For Windows x64 machines, a prebuilt binary can be downloaded from the in Appendix A, where one also can find the source code. To build HunterGatherers from scratch the first step is to download or clone the source code onto the local machine. Also required is the SFML library (version 2.5.1 or above), and an OpenCL SDK specific to the hardware of the machine. During installation of these, it is

important for the relevant environment variables are added to the system (for SFML the SFML_DIR needs to be set). To compile the GCC MinGW (SEH) 64-bit compiler (version 6.1.0 or above) is needed to be installed and added to the system path. In order to simplify the process of building, CMake (version 3.12 or above) is highly recommended.

With all the required dependencies installed, building can be done by running the following commands, with “PATH_TO_MINGW” changed to the directory where the compiler was installed:

```
mkdir build

cd build

cmake -D CMAKE_C_COMPILER="PATH_TO_MINGW/bin/gcc.exe"
-D CMAKE_CXX_COMPILER="PATH_TO_MINGW/bin/g++.exe"
-D CMAKE_MAKE_PROGRAM=" PATH_TO_MINGW/bin/mingw32-make.exe”
.. -G "MinGW Makefiles"

mingw32-make.exe
```

In order to run the program the SFML .dll-files “sfml-graphics-d-2.dll”, “sfml-system-d-2.dll”, and “sfml-window-d-2.dll” needs to be placed in the same folder as the generated executable.

Running

Simply running HunterGatherers.exe will start the simulation with the first found OpenCL-device, although all available devices will be printed in the console. To use a specific device one can use the argument “CL_DEVICE” followed by the name of the device.

The settings of the simulation are defined by a configuration JSON-file. By default, the program will use the “Config.json” in the same directory, but this can be changed using the argument “CONFIG” followed by the name of the file. These configuration settings can be changed without having to rebuild or even restart the simulation.

Observations

Initial development

The following observations are a description of a typical way the system evolves, in this section is made using the default configuration file. Using these spawn rates, when first starting the simulation, the world is empty but quickly get populated by agents, and them more slowly, mushrooms. In the beginning there aren’t enough density of mushrooms to sustain a single agent, so they are all doomed to die. Because mushrooms can self-replicated, their population count increases exponentially, and soon the whole world is populated with hundreds of mushrooms in clusters. The generation zero agents often move in random turning patterns at stable but different speeds (see Figure 2). It is common for agents to not turn when they hit the wall, and therefore destroying their chances of finding food.

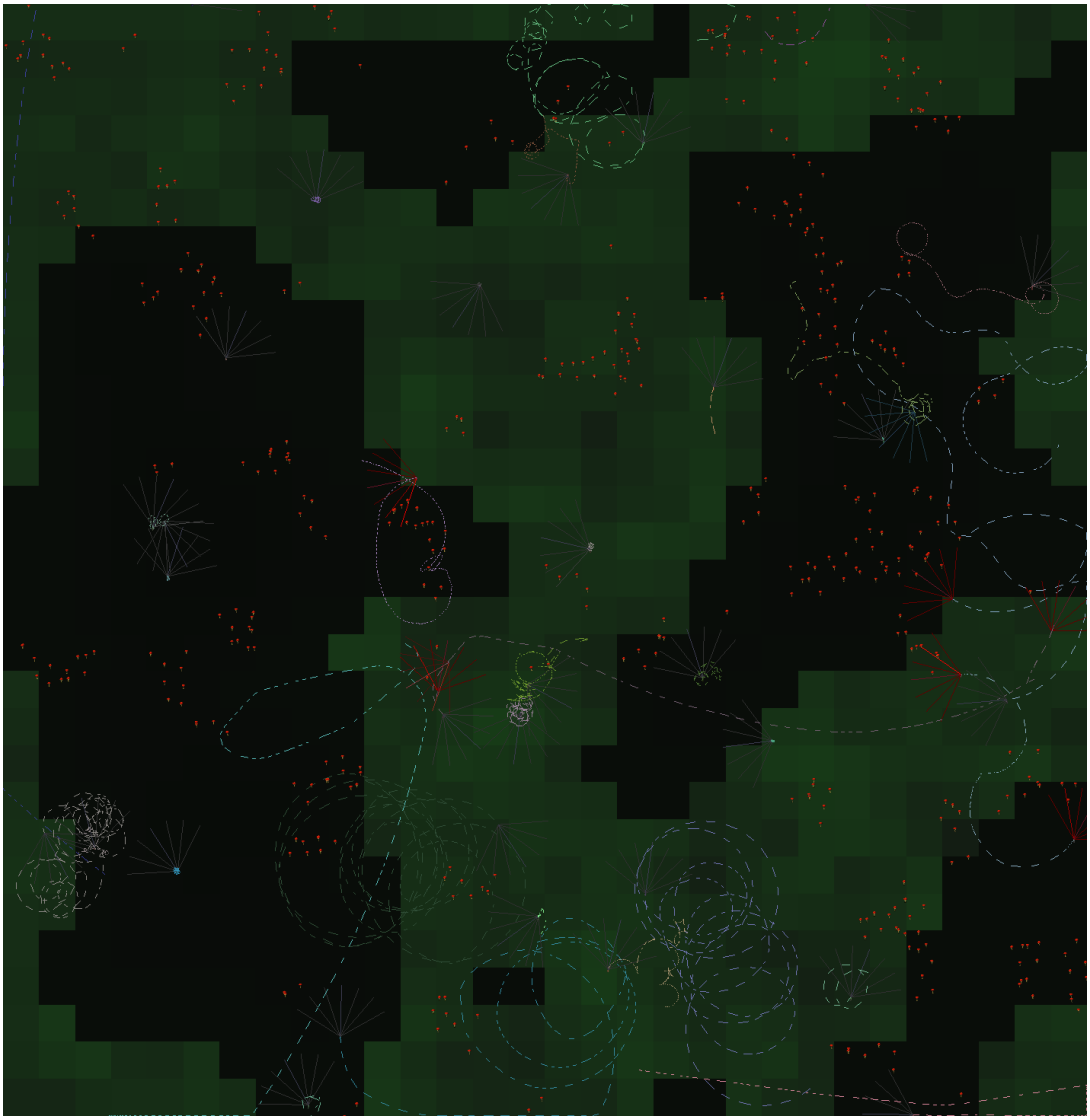

Figure 2 newly created world. Mushrooms are clustered. Agent population is low, hand with high genetic variation. The behaviours are equally varied, with some agents only going in circles and others turning at seemingly random points. Note that the background texture at this point of the development doesn't influence the simulation, and therefore can be disregarded completely.

After a short while one or multiple agents in the world walk into a cluster of mushrooms and choose to quickly reproduce, creating children which do the same (Figure 3). This often creates a cascade of new agents and is the beginning of a new species. Because these agents are so close to each other, it is essential that they quickly spread out as to not compete, which is detrimental to the survival of the genome.

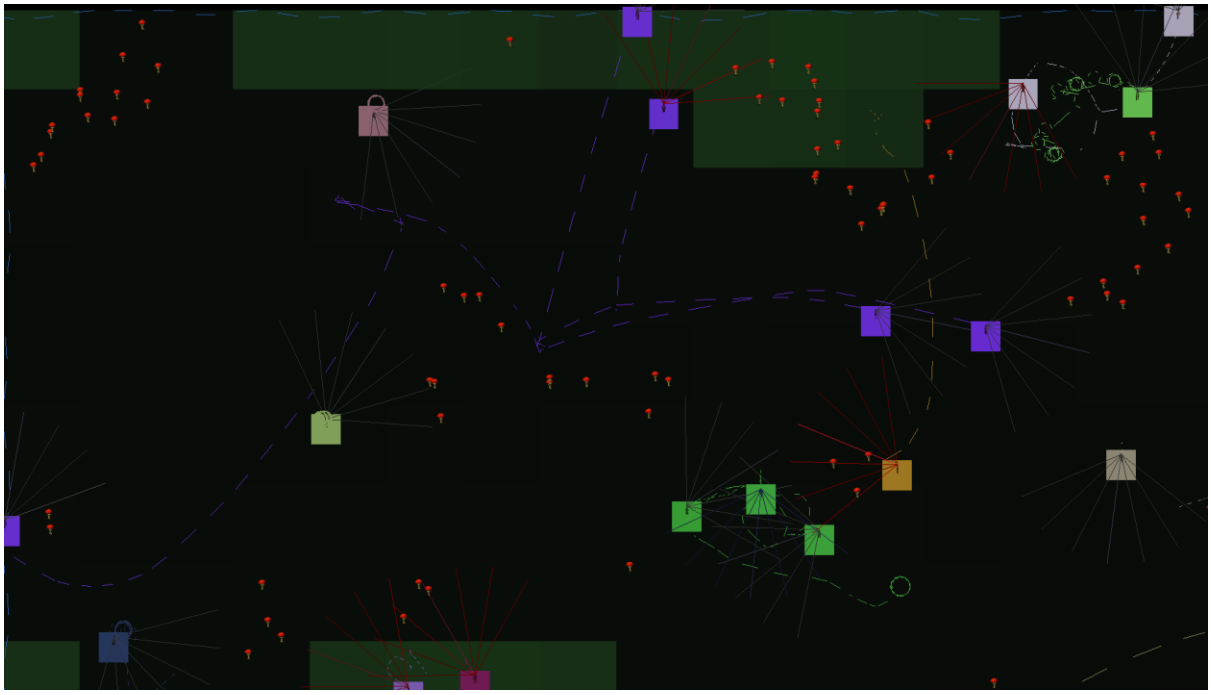

Figure 3 One blue agent finds a group of mushrooms and uses the energy to reproduce. This creates children that behaves the same, causing a cascade. Here this happens within a very short time span.

Species and completion

From here, if the species is skilled enough, it often doesn't take a long time for the species to dominate the whole world. As seen in the spectrograms in Figure 4 it is not uncommon for new species to die out naturally. In Figure 5 this instead seems to happen because of competition from new species. This likely means that there is some kind of weakness in the behaviour of the previous species, that gets eliminated by the process of natural selection not by selecting for genetic variations within the species but rather by selecting a whole new species. The news species can get a competitive edge over its opponents, as long as there is there is one. This means that these sequential species aren't independent from each other. The fact that the system doesn't seem to be stable with multiple species existing at the same time indicates there only exists one evolutionary niche in the environment.

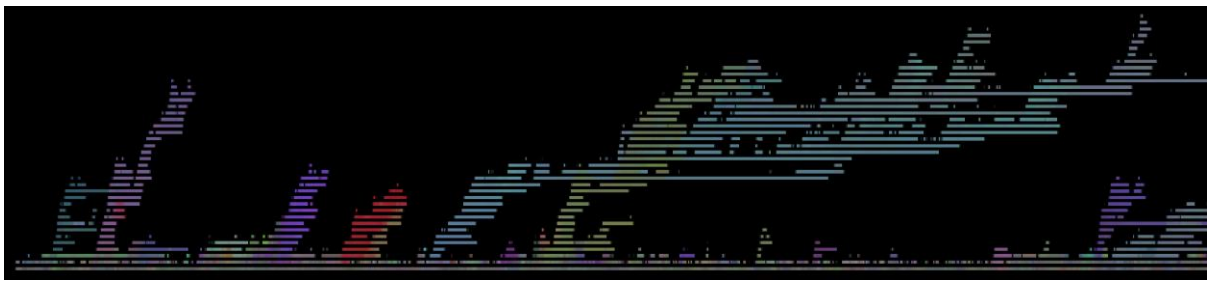

Figure 4 Depending on the nature of the environment, it can be hard for new species to sustain themselves. With this configuration the new species quickly die out naturally. This image also illustrates how multiple species can exist at the same time.

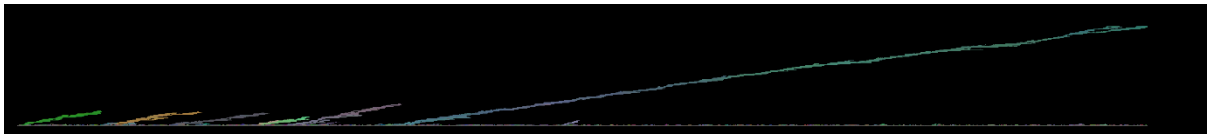

Figure 5 An environment where the 6th species to spawn is the first that's able to sustain itself over time. Looking closely one can see that most new species establish themselves despite there already existing a dominant species in the world. Also notices the lack of new species being created when the stable species has established itself.

In Figure 6 one can see that there can be branching of species, and that these species compete for resources, causing one branch to prevail. This branching doesn't necessarily come from different traits competing at being successful at reproduction. Sometimes single agents being extremely skilled at surviving but not at reproduction, can reserve resources from the species as a whole. One speculation is that this "stalling" might be one of the reasons species die out naturally.

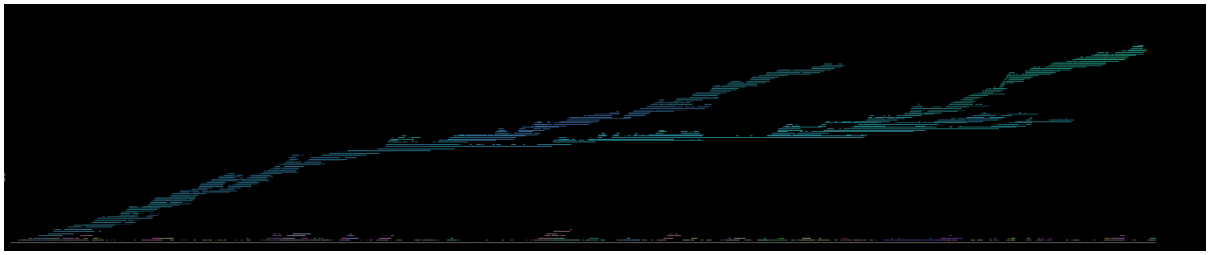

Figure 6 Branching of species, and genetic drift indicated by color gradient.

It is in fact relatively common for species to die out, despite the environment staying constant. It isn't always clear why these extinctions happen (Figure 7), but it likely always has to do with the amount of resources in the system. The fact that the world is relatively small and generally only contains up to a hundred agents makes random chance likely to play a role. Another likely contributor is that the amount of mushrooms available varies in waves. Individuals that are quick at scanning for food and reproducing but bad at surviving periods of scarcity, might for example quickly spread over the whole world and consume all available food, which causes the decline on the whole species.

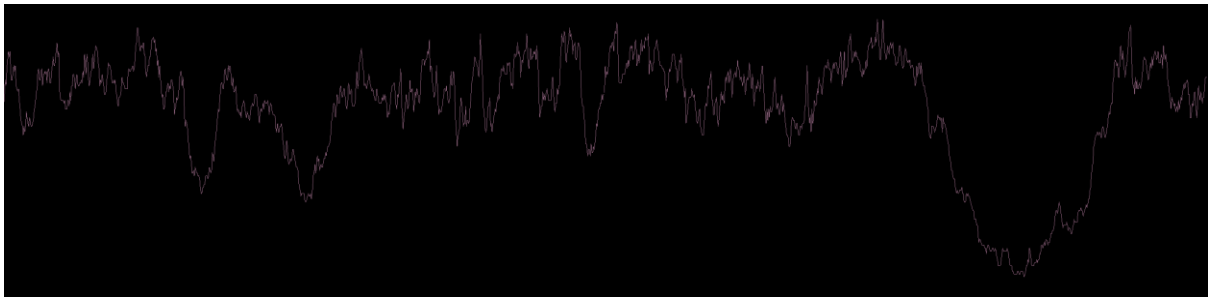

Figure 7 Population count collapses, and then recovers.

Emergent strategies

Foraging

The most basic skill needed for surviving in this world is the ability to find food. This skill can arise from the simple behavior of increasing the rate of turning if something red is detected within the visual feed. It is in other words very easy for a new species to be evolved simply doing this, and sometimes whole species can sustain themselves by for example only making left-turns. Most sustainable species are however capable of turning both left and right. There is however a quality to

this skill, and not all visual receptors are always wired up to cause turning. The longer a species has existed for, the better they get at turning towards the mushroom and the more sensitive they become at detecting mushrooms even at the edge of their visual field.

When letting a species evolve for a long time the agents also learn to change their walking speed depending on if they see mushrooms. The standard walking rate seems to be moderately high for these agents, increasing slightly if the food is right in front of them. They can however also stop completely, allowing themselves to have time to turn towards the food before walking towards it. This ability can be very effective for picking up almost all mushrooms that the agent stumbles upon, but also often backfires. For example, this makes the agent's children sensitive to genetic mutations causing them to stop if they have food in front of them. It can also cause the agents to get confused, as seen in Figure 8, where the agent seeing multiple mushrooms causes it to get paralyzed and starve to death.

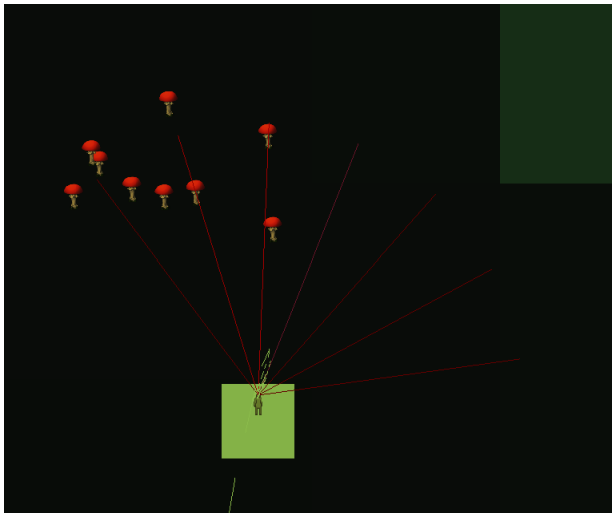

Figure 8 Highly evolved agent getting confused and paralyzed by seeing too many mushrooms.

Reproduction

As described earlier, many generation zero agents reproduce as much as possible as soon as they have the opportunity. This is good for quickly establishing a new species (see Figure 9), but also creates a lot of local competition and is in general a dangerous thing for the agent to do in terms of it's own survival. Because of the mutation rate, and because there is no kind of age limitation on the agents, the most valuable life to an agent is its own. A child might greatly increase the probability of a set of genes to survive, but there is no guarantee that the child will survive, or that it won't even decrease the chances of survival for the parent through competition or through murder.

For this reason, highly evolved species tend to reproduce much more conservatively and only when the right conditions are meet. A common strategy is for example to only reproduce when the agent has mushrooms in front of it, quickly grabbing them right after having reproduced and thereby ensuring it's own survival above all.

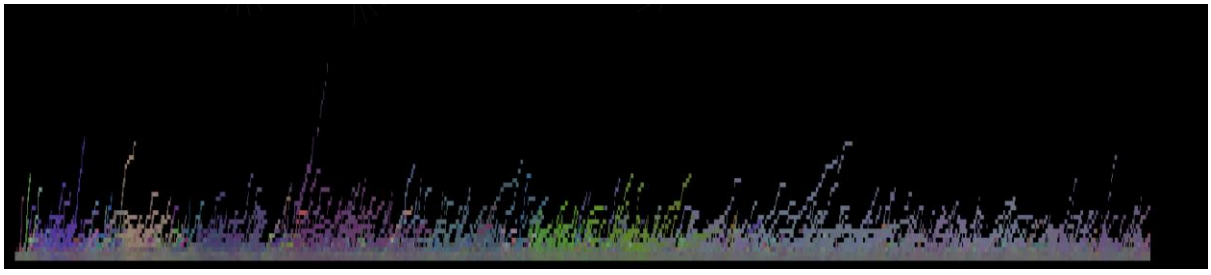

Figure 9 A spectrogram showing how many children different species and individuals tend to have. One can see how a species often become dominant as a result of a single individual having many children. Otherwise birth rates between species seem to be similar.

Saving for later

When designing the system, it was expected that the ability to save mushrooms in order to eat them later would be well used. It seemed it would be easy to evolve, with it simply being a matter of not eating, and indeed many examples of individuals were found doing just this effectively as part of an evolved strategy. This strategy usually just consists of eating when the energy level passes below some threshold. However, no matter the scarcity of food or energy consumption rates were changed, this strategy never became the part of any stable species. As can be seen in Figure 10, this ability does become common during certain times of evolution, but is then selected against, and the behavior dies out.

As with many other behaviors there is a risk to having this ability. It's safer to always eat as a heuristic, instead of making an active judgement which has the risk of being wrong and leading to starvation. This does however seem to be a problem that can be overcome and shouldn't be enough for the ability to be selected against. below it is obvious how the population seems to be inversely correlated to this behavior. This is because when the food is stored instead of eaten, it limits the systems capacity to sustain life. Saving the mushrooms might however not be advantageous to the gene. Even if an agent has a buffer of mushrooms, it still risks dying, and in this case it would be better for the genes to reproduce.

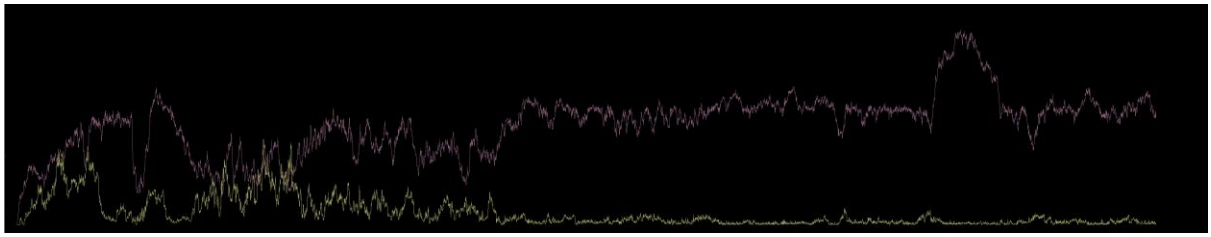

Figure 10 Graph showing the population of agents (purple) and the mean amount of mushrooms in the agent's inventory (yellow).

Opportunistic murder

Murder rate seems to stay constant during evolution. Some species are more murderous than others, but no species depend solely on murdering other agents to survive as can be seen in (Figure 11). What does seem to change during time is however how murders are done. At first, they're often random and detrimental acts, often between parent and child. This can make reproduction dangerous. With time, it does however seem to become more of a rare occurrence which is common to some members of a species, and rarer to others. It seems to be an opportunistic act, that is something the agent sometimes chooses to do when the able. For example it might be when colliding into another agent, although some intelligence is required in order to discriminate between if it is the wall or an agent.

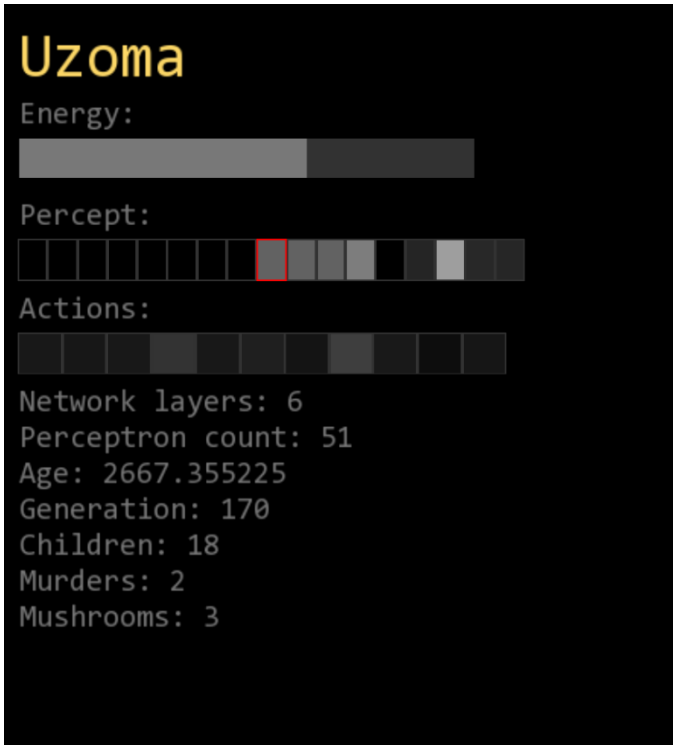

Figure 11 A highly developed and successful agent. It's network is relatively small, and it is good at reproducing and surviving. It has the ability of saving mushrooms for later as well as murdering other agents.

Conclusion

The observations and analyses presented in this report are merely a starting point for the exploration of the model. This has been a discussion of the overarching phenomenon that tend to appear when running it, but the deeper and longer one looks, the more details in behavior one can find. To aid in the analysis of these behavior a sweet of tools are supplied to the user, and the open source nature of the software allows for further development and modifications to the system. The graphical metaphors of humans (hunter gatherers) and mushrooms makes the model intuitive to understand. The implementation of the model is done in a cross-platform-friendly way, and makes efficient use of the available resources of the system.

Future avenues for development/experimentation for HunterGatherers could be ways of creating more niches, allowing for multiple species at the same time, learning during the life time, allowing the agents to have other items in their inventory, and aging.

References

dvgrn. (2013, November 23). *Re: Geminoid Challenge*. From conwaylife: http://www.conwaylife.com/forums/viewtopic.php?f=2&t=1006&p=9917#p9901

Google. (2019, June 17). *Overview of GAN Structure*. From developers.google.com: https://developers.google.com/machine-learning/gan/gan_structure

Huizinga, J., Mouret, J.-B., & Clune, J. (2014). Evolving Neural Networks That Are Both Modular and Regular: HyperNeat Plus the Connection Cost Technique. *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, 697-704.

Rendell, P. (2001). A Turing Machine In Conway's Game Life.

Russell, S. J., & Norvig, P. (2009). *Artificial Intelligence: A modern approach.* Harlow: Prentice Hall.

Smith, C. (2013, May 22). *Day 19: Chladni Plates*. From flickr: https://www.flickr.com/photos/cjsmithphotography/8800645088/

Taylor, C., & Jefferson, D. (1993). Artificial Life as a Tool for Biological Inquiry. *Artificial Life*, 1-13.

Appendix A

Source:

https://github.com/Axelwickm/HunterGatherers

Prebuilt Windows x64 binary:

https://github.com/Axelwickm/HunterGatherers/releases/tag/v1.0

Appendix B

```
void Agent::constructGenome(size_t inputCount, size_t outputCount) {

    // Function for finding out how many weights every peceptron should have
    auto previousLayerPerceptronCountLambda = [](LambdaGene<int> &l, float mutationFactor) {
        auto layers = l.getOwner<MapGenes>()->getOwner<ListGenes>()->getOwner<MapGenes>()
                      ->getOwner<ListGenes>();
        auto thisLayer = l.getOwner<MapGenes>()->getOwner<ListGenes>()->getOwner<MapGenes>();

        // Find which layer is calling this function
        auto itr = layers->getList().begin();
        for (auto &_ : layers->getList()) {
            if (itr->get() == thisLayer){
                break;
            }
            itr++;
        }

        // If this is the first layer, the count depends on how many inputs the network has
        if (itr == layers->getList().begin()){
            auto count = layers->getOwner<MapGenes>()->getGene<IntegerGene>("InputCount");
            count->evaluate(mutationFactor, l.getEvaluationCount());
            return count->getValue();
        }

        // Else, the count depends on the perceptron count in the previous layer
        itr--;
        auto lastLayer = ((MapGenes*) itr->get());
        auto count = lastLayer->getGene<LambdaGene<int> >("PerceptronCount");
        count->evaluate(mutationFactor, l.getEvaluationCount());
        return count->getValue();

    };

    // Function for finding how many perceptrons this layer should be
    auto perceptronCountLambda = [](LambdaGene<int> &l, float mutationFactor) {
        auto layers = l.getOwner<MapGenes>()->getOwner<ListGenes>();
        auto thisLayer = l.getOwner<MapGenes>();

        // Find which layer is calling this function
        auto itr = layers->getList().begin();
        for (auto &_ : layers->getList()) {
            if (itr->get() == thisLayer){
                break;
            }
```

```
        // If this is the last layer, then the count depends on how many outputs the
        // network has
        if (++itr == layers->getList().end()){
            auto count = layers->getOwner<MapGenes>()->getGene<IntegerGene>("OutputCount");
            count->evaluate(mutationFactor, l.getEvaluationCount());
            return count->getValue();
        }

        // Else, the count depends on the MutatingPerceptronCount gene,
        // and is therefore random
        auto count = l.getOwner<MapGenes>()->getGene<IntegerGene>("MutatingPerceptronCount");
        count->evaluate(mutationFactor, l.getEvaluationCount());
        return count->getValue();
    };




// Create a perceptron map which has a weight count, and a list of the weights.
    auto perceptron = std::make_shared<MapGenes>();
    auto weightCount = std::make_shared<LambdaGene<int> >(previousLayerPerceptronCountLambda);
    perceptron->addGenes("WeightCount", weightCount);
    auto weight = std::make_shared<FloatGene>(settings.weightMin, settings.weightMax);
    auto weights = std::make_shared<ListGenes>(weight, "WeightCount");
    perceptron->addGenes("Weights", weights);

    // Create a layer map which has:
    auto layer = std::make_shared<MapGenes>();
    // a random mutating integer gene which might be used to define count of perceptrons,
    auto mutatingPerceptronCount =
std::make_shared<IntegerGene>(settings.perceptronPerLayerMin, settings.perceptronPerLayerMax);
    layer->addGenes("MutatingPerceptronCount", mutatingPerceptronCount);
    // a lambda gene which decides if the mutatingPerceptronCount should be used,
    auto perceptronCount = std::make_shared<LambdaGene<int> >(perceptronCountLambda);
    layer->addGenes("PerceptronCount", perceptronCount);
    // the bias for this layer,
    auto bias = std::make_shared<FloatGene>(settings.biasMin, settings.biasMax);
    layer->addGenes("Bias", bias);
    // a list of perceptron maps
    auto perceptrons = std::make_shared<ListGenes>(perceptron, "PerceptronCount");
    layer->addGenes("Perceptrons", perceptrons);

    // The top gene is a map gene containing a layer count, predefined input and output
counts,
    // and a list of layer maps
    genes = std::make_shared<MapGenes>();
    auto layerCount = std::make_shared<IntegerGene>(settings.layersMin, settings.layersMax);
    genes->addGenes("LayerCount", layerCount);
    auto inputCountG = std::make_shared<IntegerGene>(inputCount, inputCount);
    genes->addGenes("InputCount", inputCountG);
    auto outputCountG = std::make_shared<IntegerGene>(outputCount, outputCount);
    genes->addGenes("OutputCount", outputCountG);
    auto layers = std::make_shared<ListGenes>(layer, "LayerCount");
    genes->addGenes("Layers", layers);

    // The genome is then generated according to this structure
    genes->generate();
}
```

Appendix C

```
{
  "_seed" : "seed is either an unsigned int, or TIME",
  "seed" : "TIME",

  "WorldSettings" : {
    "worldWidth" : 7500, "worldHeight" :  7500,

    "mushroomReproductionRate" : 0.02,
    "mushroomReproductionDistance" : 120,
    "mushroomReproductionNearLimit" : 3,

    "terrainSquare" : 30,
    "quadtreeLimit" : 60,
    "PopulatorEntries" : [
      {
        "type" : "Agent",
        "targetCount" : 100,
        "rate" : 0.25
      },{
        "type" : "Mushroom",
        "targetCount" : 750,
        "rate" : 0.14
      },{
        "type" : "BouncingBall",
        "targetCount" : 0,
        "rate" : 0.75
      }
    ]
  },

  "AgentSettings" : {
    "mass": 1,
    "friction" : 0.01,
    "maxSpeed" : 1000,
    "turnFactor" : 65,
    "punchTime" : 1,
    "actionCooldown" : 2,

    "energyToParent" : 0.45,
    "energyToChild" : 0.3,
    "energyLossRate" : 0.2,
    "movementEnergyLoss" : 0.015,
    "turnRateEnergyLoss" : 0.00375,
    "punchEnergy": 6,
    "punchDamage" : 30,
    "mushroomEnergy" : 20,
    "maxEnergy" : 100,
    "maxMushroomCount": 20,

    "canReproduce" : true,
    "canWalk" : true,
    "canTurn" : true,
    "canEat" : true,
    "canPlace" : true,
    "canPunch" : true,

    "memory" : 4,
    "memoryReactivity" : 0.18,

    "perceiveCollision" : true,
    "receptorCount" : 7,
    "perceiveColor" : true,
    "perceiveEnergyLevel" : true,
    "perceiveMushroomCount" : true,

    "FOV" : 120,
    "visibilityDistance" : 350,
    "visualReactivity" : 1,
```

```
    "mutation" : 0.03,
    "layerMin" : 4, "layerMax" : 6,
    "biasMin" : -1, "biasMax" : 1,
    "weightMin" : -2, "weightMax" : 2,
    "perceptronPerLayerMin": 6,
    "perceptronPerLayerMax": 10
  },

  "Controls" : {
    "pause" : "Space",
    "close" : "Escape",
    "showInterface" : "D",
    "clearStats" : "C",

    "up" : "Up",
    "down" : "Down",
    "left" : "Left",
    "right" : "Right",

    "slowDown" : "Comma",
    "speedUp" : "Period",

    "keyboardCameraMove" : 15,
    "timeFactorInitial" : 9,
    "timeFactorDelta" : 0.05,
    "timeFactorMax" :  20,
    "scrollFactor" :  0.05
  },

  "Rendering": {
    "windowWidth" : 1920,
    "windowHeight" : 1080,

    "graphLine" : false,
        "graphPopulation" : true,
        "graphMeanGeneration" : false,
        "graphMeanPerceptrons" : false,
        "graphMeanAge" : false,
        "graphMeanChildren" : false,
        "graphMeanMurders" : false,
        "graphMeanEnergy" : false,
        "graphMeanMushrooms" : false,
        "graphMeanSpeed" : false,

    "graphSpectrogram" : true,
        "graphGeneration" : true,
        "graphPerceptrons" : false,
        "graphAge" : false,
        "graphChildren" : false,
        "graphMurders" : false,
        "graphEnergy" : false,
        "graphMushrooms" : false,
        "graphSpeed" : false,

    "bins" : 50,
    "showInterface" : true,
    "showWorldObjectBounds" : false,
    "showQuadtree" :  false,
    "showQuadtreeEntities" : false,
    "showVision" : true
  }
}
```