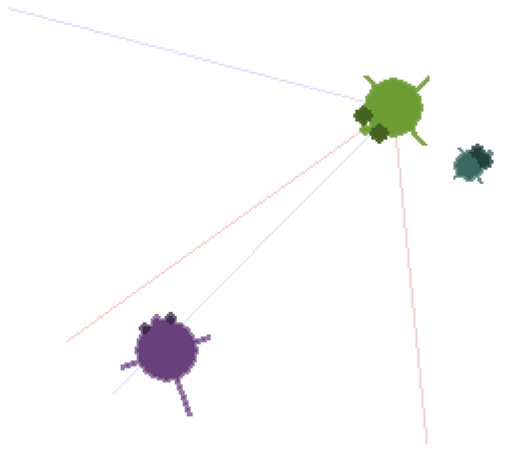
Development and Analysis of “Cambrians”

Creating computer-generated ecosystems using AI



Author: Charles Maddock

2019

Abstract

This project stems from the desire to create a complete software that generates an ecosystem through the use of artificial intelligence, by creating artificial primitive animals and allowing them to evolve. The paper describes the software's structure as well as discussing topics such as *creation of artificial life* and *optimisation of evolution for computer processing*. Finally, it is discussed if the software's generated environments could be considered realistic digital ecosystems. The conclusion is that, whilst many of the software's generated environments share many aspects with real ecosystems, it would be a stretch to call them realistic digital ecosystems since many important factors are missing. Instead, they should be viewed as simplified versions of ecosystems.

1. Introduction

This work documents my attempt to create an AI-based computer program that creates and simulates ecosystems containing primitive animals through the use of artificial intelligence. In any given iteration of the program, animals' behaviours are dictated by their neural network that consists of connected, computer-generated neurons. Animals with behaviours that benefit their survival reproduce more and consequently, their neural networks and behaviours are represented in later generations' gene pools. The animals have a level of complexity similar to the animals of the early geological era Pre-Cambria, hence the program's name "Cambrians".

1.1 Artificial Intelligence

Artificial intelligence, or AI, is a broad term for intelligence demonstrated by computer systems. Most AIs purposes are to resemble the brain's ability to draw conclusions, solve problems and learn.[1] This section will describe the two main 'AI components' utilized in the software: Neural networks and genetic algorithms.

Neural Networks

Keep in mind:
Neural networks are hard to understand at first. For this reason, and since they are such fundamental components of this paper, heavy emphasis will be put on the description of them. This results in an introduction that is both longer and more informal than it would be otherwise.

Artificial neural networks are a form of artificial intelligence that are vaguely inspired by the network of neurons that compose the animal brain.[2] Artificial neural networks consist of neurons and connections between the neurons. The connections each have a *weight* that is

[1] *Nationalencyklopedin,* artificiell intelligens, https://www.ne.se/uppslagsverk/encyklopedi/l%C3%A5ng/artificiell-intelligens, Retrieved 2019-01-22

[2] *Frontiers*, van Gerven, Marcel, Bohte, Sander (2017), https://www.frontiersin.org/articles/10.3389/fncom.2017.00114/full , Published on 19 December 2017, Retrieved 2018-02-20.

often times a number between -1 to 1. Similar to the animal brain, a well-adapted neural network can process inputs to produce outputs that confer success. The easiest way to understand neural networks is through examples. A simple example of a neural network will therefore be presented, explaining the very basics.

The Figure below is without doubt the most simple version of a neural network. It consists of three components: One *input neuron*, a *connection* and one *output neuron*.

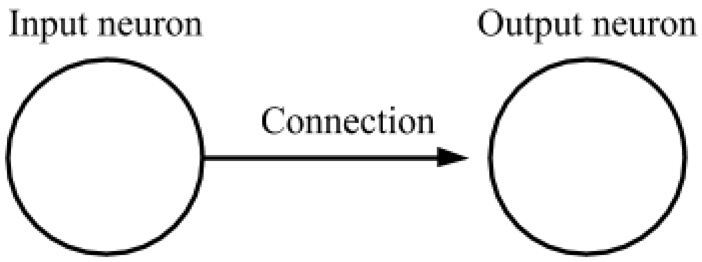


Fig 1, simple neural network without values.

The point of these three components are to take in a number and output a different one. An example of what this can look like can be seen in the Figure below.

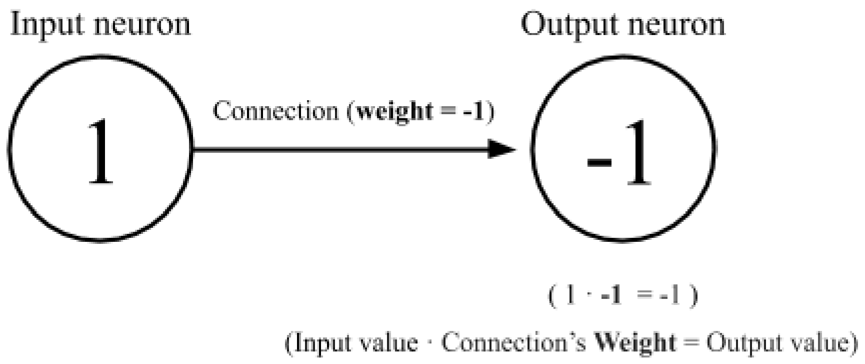

Input neuron
Output neuron
Connection (**weight = -1**)
1
-1
(1 · **-1** = -1)
(Input value · Connection's **Weight** = Output value)

Fig 2, simple neural network with values.

In the example above, the number '1' has been entered into the input neuron. The number then 'passes' through the connected to the output neuron and changes to -1. The number changes since it is multiplied by the *weight* of the connection it passes through. In this case, since the input value is 1 and the connection's weight is -1, the output value is -1 because
1 · -1 = 1.

Now, from the short explanation above the basis for neural networks should be clear. Simply looking at numbers does not say much however, therefore, to clarify how neural networks can be used to solve a task, another example will be given.

Say for instance that one would want to create a very simple digital animal. The animal has a tail and one eye (that is limited to seeing plants), so it's only abilities are to see plants in front of it and to move straight forward. If we wanted to give this animal a brain, we could use a neural network that looks like this:

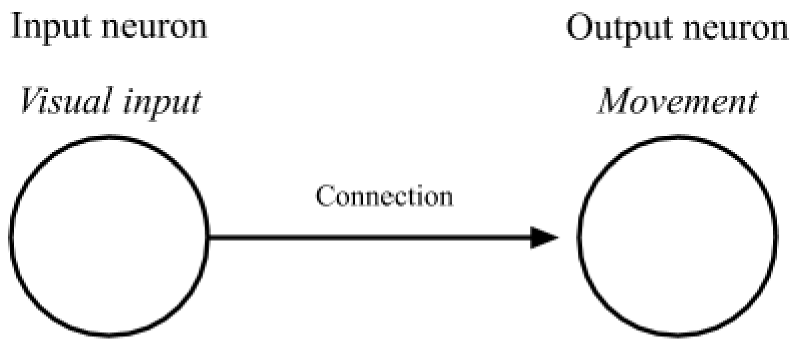


Fig 3, neural network, or brain, of a simple digital animal.

This animal's brain, or neural network, has one input neuron and one output neuron. The input neuron is coupled to the animal's eye, so if the animal for example sees a plant in front of it, the input neurons value will change. The connection's weight will then decide the output neuron's value, by multiplying the input value with the weight. The output neurons value will then decide the animal's movement, or in other words, at which speed to move either forward or backwards. Let's look at an example.

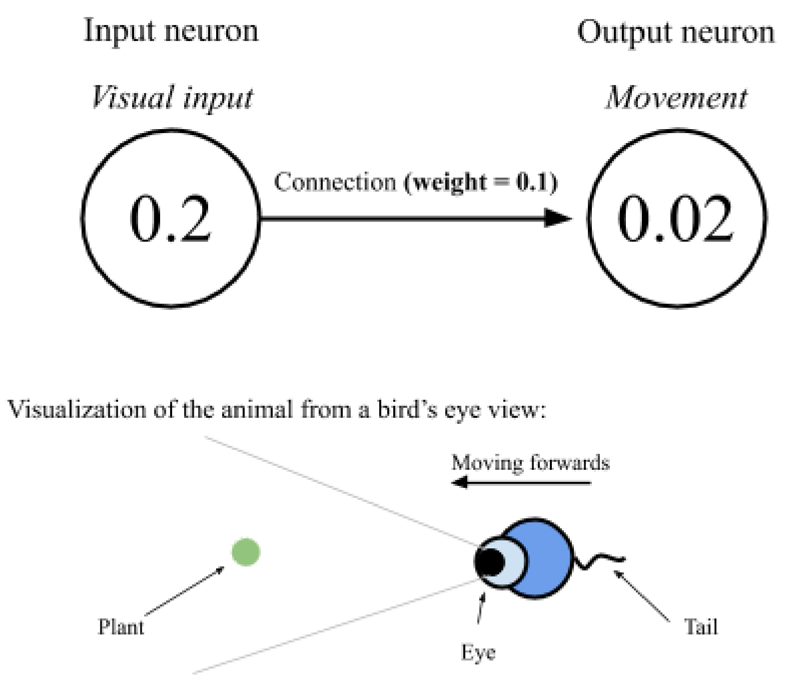


Fig 4, neural network of a simple digital animal with a visualization of the animal.

In the example above, the animal's eye has spotted a plant in front of it. If the plant had been closer to the animal the input value would have been a larger number such as 0.9 or 1, but in this case it is further away so the value is 0.2. Since the connections weight is 0.1 the output neuron's value is 0.02. This means that the animal is moving at two percent of its max speed. It will therefore eventually reach the plant and eat it, avoiding starvation.

This animal's neural network gives it a simple food-seeking behavior that promotes its survival. It is important to remember that it is the connection's weight is the deciding factor in its behaviour.

To further demonstrate the importance of the weights being 'suitable', let's imagine that the animal from the example gives birth to two mutated copies of itself. The offsprings' neural networks are mutated so that one child's connection has a weight of -0.1 and the other a weight of 0.5. The child with the negative weight will actually move backwards when seeing a plant, which is a 'bad' trait, whilst the child with the weight of 0.5 will move towards plants faster than its parent, since its weight is larger and therefore also its output value. See the Figure below.

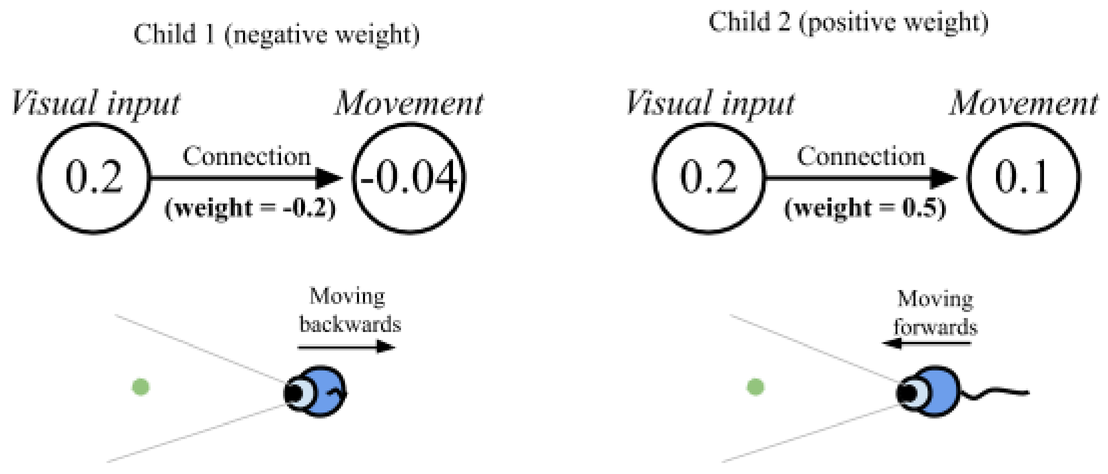


Fig 5, the offspring's neural networks.

Even though Cambrians's animal's neural networks are a lot more complex with many more intertwining connections with different weights, they work and evolve in a very similar fashion to the given example. Any 'smart' animal in Cambrians is the result of many generations' mutations to said species neural network's weights. Cambrians's neural network structure is explained in greater detail in section 2.3.

The given examples were very simple. Usually a neural network consists of many more input and output neurons. If there are several input neurons, all input neurons in a neural network are called the input layer. The output neurons are referred to as the output layer. Sometimes there are so called hidden layers. Hidden layers are simply neurons in between the input and output levels that allow for more complex 'thinking' in neural networks. With more neurons there will be several connections leading to the same neuron. The values from the different connections are added together inside the joining neuron and are then often times compressed to a desirable range with the help of an activation function. A common activation function is the standard sigmoid function, which returns values between -1 and 1.

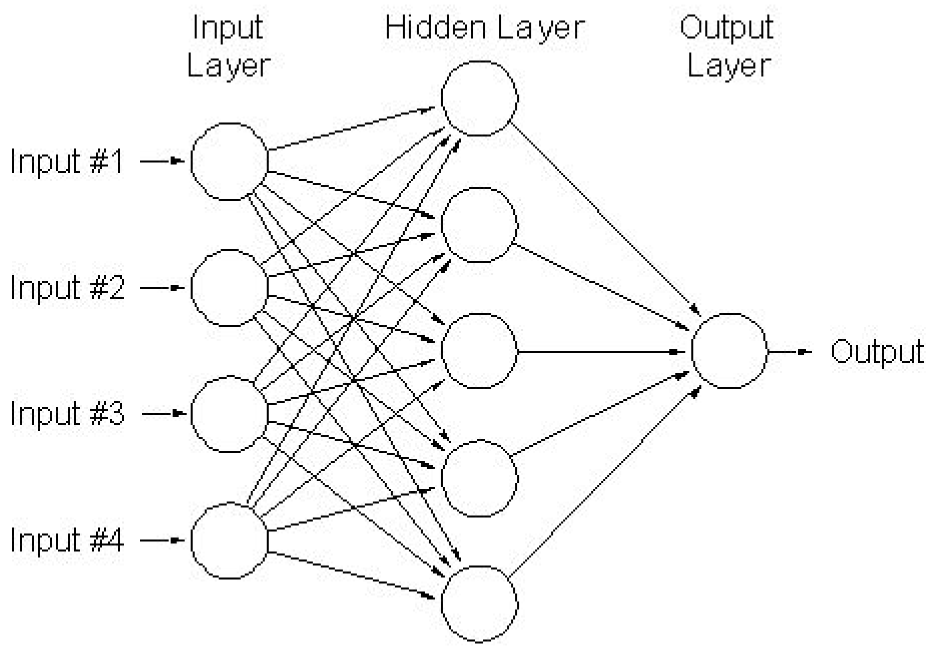


Fig 6, a neural network with several layers. Every input neuron is connected to every hidden neuron. When deciding the value of a given hidden neuron, the connections leading to its different values are added together inside the hidden neurons and then passed on to the output neuron.

Genetic Algorithms

A genetic algorithm is a higher-level mathematical procedure inspired by the process of natural selection.[3] Genetic algorithms can differ a lot in complexity and efficiency, however, the majority work by evaluating the fitness of individual neural networks in a population of neural networks and then selecting the neural networks with the greatest fitness. The fitness of a neural network describes how well it performs its given task. How fitness is measured for animals in Cambrians and how fittest animals are selected is explained in greater detail in the following section.

1.2 Biological Terminology

This section describes relevant ecological terms as well as vocabulary used in Cambrians for certain biological phenomenon.

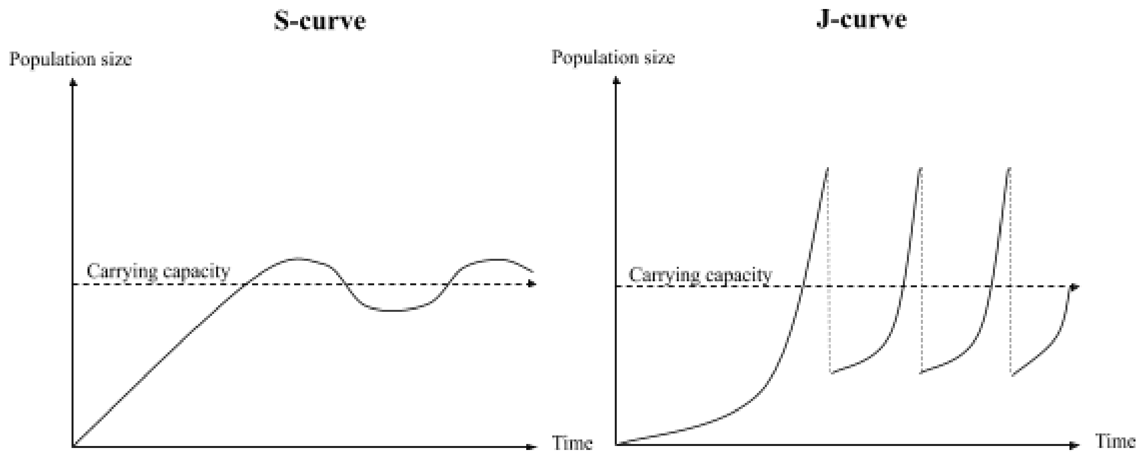
[3] Mitchell, Melanie (1996). *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT Press. ISBN 9780585030944, Retrieved 2018-02-20.

Ecosystem

An ecosystem is per definition a biological community interacting with each other and their environment.[4] In other words, biotic (living) and abiotic (non-living) components interacting.

Carrying Capacity and R/K Selected Species

Within ecology, carrying capacity is the maximal amount of individuals that can live of the resources in any given environment.[5] When analysing the carrying capacity and population growth there are generally two main growth curves: The j-curve and the s-curve. They are both centred around the carrying capacity, however, differ largely in appearance. See the diagrams below.



Diagrams 1 and 2, simple visualization of the two different population growth curves

The curves look the way they do depending on the species whose population is being measured. Animals whose birth cycles are slow, such as for instance humans, otherwise known as K-selected species, generally grow in accordance to the s-curve. On the other hand, animals such as rabbits, otherwise known as R-selected species, grow rapidly in population and then die out as resources diminish according to the J-curve.

[4] *Nationalencyklopedin,* ekosystem - Uppslagsverk - NE.se." https://www.ne.se/uppslagsverk/encyklopedi/l%C3%A5ng/ekosystem. Retrieved 2019-01-19

[5] *Nationalencyklopedin,* bärförmåga, http://www.ne.se/uppslagsverk/encyklopedi/lång/bärförmåga, Retrieved 2019-01-22

Predator-Prey Cycles

When an ecosystem contains predators and prey animals an interesting statistical phenomenon sometimes appears: When the prey animal population increases, the predator population increases as well. However, when the predator population increases, hunting of the prey animals increases as well, causing the prey population to sink. This eventually leads to a decrease in the predator population, as there is less food for the predators, which brings us back to the start were the prey species is free to grow in population since there are fewer predators. This statistical phenomenon is sometimes referred to as predator-prey cycles.

1.3 Software Structure

Cambrians is written in the object oriented programming language Python (version 3.7.0). The software is not completely written in default Python, large parts of the program were made with libraries. Libraries are Python extensions that ease the development of certain areas. The following section will explain Python and the Python libraries used in Cambrians.

Python

Python is a high-level programming language for general-purpose programming. It is widely used in the scientific community since it is powerful, yet easy to learn. Python's main design philosophy is code readability, for this reason, Python feels intuitive when first studying it in contrast to the majority of other program languages that use traditional programming syntax.[6]

Pygame

Pygame is a library that eases the visual and keyboard-input aspect of programming with Python. Pygame's draw and key functions simplify the process of implementing graphics and keyboard input to a given Python program.

[6] Kuhlman, Dave, "A Python Book: Beginning Python, Advanced Python, and Python Exercises". Archived from the original on 23 June 2012, Retrieved 2018-02-20.

NumPy

Numpy adds support for large, multi-dimensional matrices, along with high-level mathematical functions to operate on these matrices.[7] Therefore, NumPy allows for the implementation of neural networks with only a few lines of code.

2. Method

The following chapter describes the process of creating the software. It begins by describing inspiration and original ideas for the software and then continues to explain all components of the finished program in greater detail, discussing the different choices that were made.

2.1 Planning and Ideas

Original Idea

I was always sure that I wanted to create a computer program for my thesis paper. The original idea was to create a software that simulated a functioning ecosystem with manually programmed animals. However, after some thought, I decided that a more interesting approach would be to create animals with neural networks that, in their struggle for survival, created ecosystems by themselves.

Inspiration

My main inspiration when it comes to the software's general function is Andrej Karpathy, Computer Science Ph.D at Stanford. He owns a youtube channel from which he year 2009 posted a video titled "Evolution of animals using neural networks".[8] In his video, Karpathy explains the basics of his program. The first version Cambrians is largely inspired by the ideas and methods from his video.

[7] *NumPy*, https://www.numpy.org, Retrieved 2018-03-10

[8] *Youtube*, Andrej Karpathy, Evolution of animals using neural networks, https://www.youtube.com/watch?v=2kupe2ZKK58&t=4s

2.3 Cambrians - The Program

This section will showcase all of the different components of the final version of Cambrians except for the animal, which is so complex a component it deserves its own section. The development process is also documented and available in appendix, section 5.5, which describes encountered bugs and other issues, as well as my thoughts surrounding certain implementations. If this interests you or if you plan on creating a similar program, I would recommend reading it.

Starting the Program

The program is started by entering the command `python3 Main.py` into the command prompt in the directory that contains the software's files. This initializes the program, setting all variables, creating the world and most importantly creating 30 random animals. These random animals have random weights in their neural networks, causing them to have random behaviours. Generally, the majority of these animals die quickly since their behaviours do not include the ability to find food. Sometimes however, a few lucky animals have a behaviour that increases their chance of survival, such as for instance turning left when seeing a plant in their left eye. These animals can give birth to offspring, which might mutate and become even better at finding food. Several generations later, normally after a few hours, animals that are born are very good at completing several different tasks that increase their chances of survival.

Main Loop

One of the software's essential functions is the main loop. The main loop is quite simply a loop that does everything the program has to do a certain amount of times per second. This includes drawing the animals, calculating the neural network's outputs, checking if there are any inputs from the user and so on. This task is easily done with Pygame's clock functions. With the clock function *tick()* the main loop can be limited to a reasonable amount of loops per second. Through trial and error, I concluded that 40 loops per second is the best amount for Cambrians; not too many processor-heavy calculations whilst not giving the animals a 'laggy' appearance. See the flowchart below.

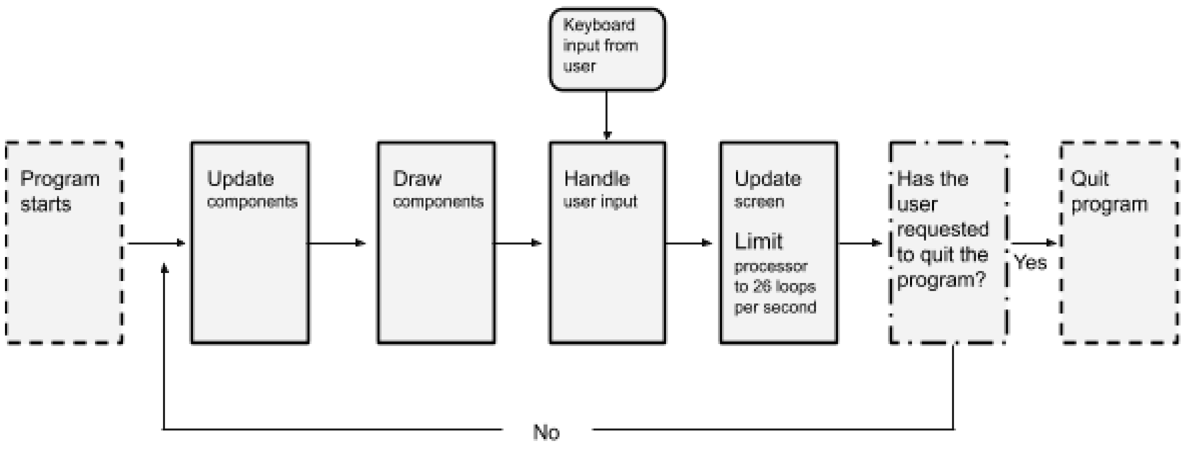


Fig 7, a flowchart that describes the main loop.

Genetic Algorithm

The first version of the genetic algorithm reintroduced mutated clones of successful dead animals. In other words, animals that lived long and gave birth to many offspring. It also introduced new random animals. However, it was quickly discovered that this genetic algorithm was flawed, as unsuccessful animals seemed to be promoted by the algorithm. Animals I referred to as "dumb spinners" would always gain the highest fitness. Dumb spinners spun around in circles and hoped to collide with large corpses and with the gained energy give birth to many offspring that also spun, but without energy. The offspring would die and not compete against the parent, allowing it to give birth to more offspring and hopefully collide with another corpse. To solve this, a new fitness definition was created that essentially increased animals' fitness for giving birth to successful offspring, instead of rewarding fitness for simply giving birth to many offspring.

As mentioned, the genetic algorithm caused many problems when it came to the evolution of animals in Cambrians. So why is one needed at all? There are two main reasons:

1. It reduces computer processing cost. The most preferable scenario would be to have a huge world with millions of animals. This way, natural selection would function more like in real life. However, this is impossible from a computer processing aspect, especially considering the fact that my own computer's processing power is low. Instead, the impression of an environment containing many successful animals is

created by storing successful animals with the genetic algorithm. If one successful species happens to go extinct due to bad luck it still exists in the program. One could imagine that the unlucky 'extinct species' went extinct locally but then 'migrated back' from another area when reintroduced with the genetic algorithm. However, if a species proves to be unsuccessful several times the genetic algorithm will remove it, causing it to 'extinct' permanently.

2. To store evolutionary progress, so that it is not lost whenever execution stops. By having a genetic algorithm that saves successful animals to the computer's memory one does not have to worry about losing evolutionary progress if the program is for instance quit by accident. If the program is quit, a new iteration of the program can be started in which stored animals can be 'resurrected' to recreate the quit iteration.

The final version of the genetic algorithm does three things: Defines animals' fitnesses, stores fit animals and reintroduces fit animals. A detailed description of how fitness is defined in Cambrians is available in appendix, section 5.4. As for storing and reintroducing: Clones of the animals with the highest calculated fitnesses are periodically reintroduced to the simulation at random locations, at regular intervals of circa 10 minutes. The clone is treated as an independent ancestral animal, thus it does not give fitness to the animal it was cloned from, its generation is also set to zero and it is treated as a new species. If there are no fit animals to clone, random animals are created. At the ten-minute mark all the dead animals' fitnesses are evaluated and if a dead animal has gained a fitness higher than that of another dead animal in the list where fit animals are stored, the two swap places, so that the list of fit animals only contains the fittest animals.

Finally, it is important to remember the fact that many of the fittest animals are 'selected' without the help of the genetic algorithm through basic natural selection. Animals that are adapted to their environment survive better than those who do not, leading to the program producing fit animals without the genetic algorithm. However, as mentioned, the genetic algorithm serves several useful purposes and therefore the program would be worse of without it.

Neural Network

In the early versions of the software, the software's neural networks had an input layer, one hidden layer with seven hidden neurons and an output layer. This structure was used as a placeholder whilst other features were developed for the animals. It was not until much later that experiments on different neural networks began. After many tests, it was discovered that the number and size of hidden layers was arbitrary to a large extent, except for the fact that the evolution of the neural networks went slower when more layers were introduced, or in other words, it took longer time to produce successful animals when they were given larger brains. The final neural network was therefore very similar to its first version with only one hidden layer.

The activation function was altered slightly throughout development. In the beginning, the default sigmoid function was used. This is due to the fact that animals should be able to 'react to nothing' such as when animals are searching for food and all input values are zero. The default "tanh" activation function was not used because it would, when all inputs are zero, result in the animal doing nothing. Furthermore, it would sometimes return zero resulting in division-with-zero errors. The sigmoid function did not have these drawbacks, which made it the best choice. Since all input values are positive numbers the sigmoid function was later altered to account for this. See the Figure below.

$$\frac{1}{1+e^{(1.5-2x)}}$$

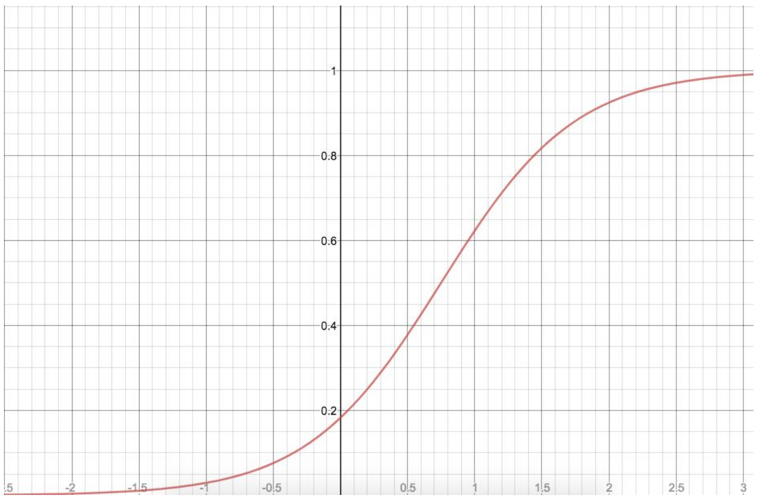


Fig 8, the sigmoid function used in the software's neural networks.

In summary, the final structure for the neural network used in Cambrians consists of an input layer, one hidden layer and an output layer. The hidden layer has nine neurons. Connection weights lie between -1 to 1. The activation function in Cambrians is displayed in Figure 8.

The World

The world is a plane in which animals live and interact with each other. It has a bright blue colour and it is resizable with the keyboard controls. If an animal reaches one of the world's edges it appears on the opposite edge to the one it passed through. Plants also grow randomly in the world at a rate that is adjustable with the keyboard controls.

Plants

Plants are static objects that grow randomly in the world and which are eaten by herbivores, providing them with energy. When a plant is created its initial radius is 1 pixel. This increases with time and after circa thirty seconds it reaches its maximum radius of eight to ten pixels. A plant disappears three minutes after its creation if not consumed, resembling decomposition.

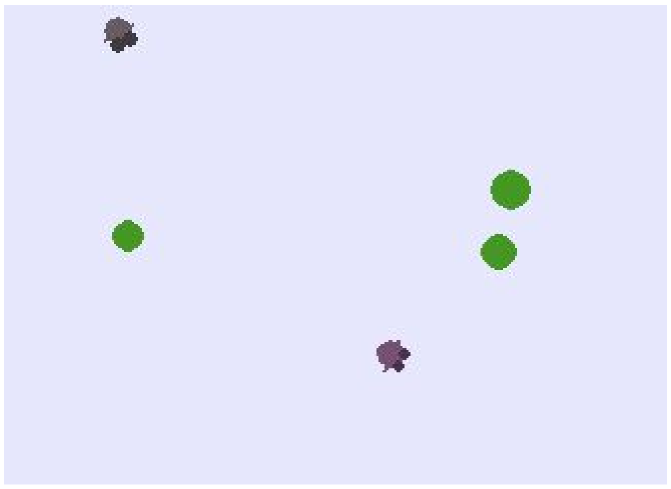


Fig 9, three fully grown plants and two small animals.

The Control Panel

Animals can be selected by clicking on them with the mouse. The control panel shows the selected animal's energy, health, age, metabolism type (herbivore or carnivore), number of birthed offspring, number of mutations and mutation rate. It also displays a visualisation of the animal's neural network.

2.4 The Animal

Vision

When it came to programming the eyes two options were considered: Either to give the animals the ability to detect colour, or to give the animals the ability to recognise certain objects. It was decided to give animals the ability to recognise certain objects instead of seeing colour. For efficiency reasons this method is preferable as herbivores would not, for instance, have to first evolve a fear for all colours that predators in its environment have, and then evolve a fleeing behaviour from them. Instead, with the chosen method, it could recognize all predatorial species from the beginning and then be able to directly evolve the fleeing behaviour. Furthermore, this allows for the dynamic opportunity to store and relocate a herbivore from one instance of the software to another and still allow the herbivore to recognise carnivores in the new instance, even if the carnivores in the new environment have different colours than the ones in the herbivore's original environment.

Animals' can with their eyes recognize related animals, unrelated herbivores, unrelated carnivores, plants and corpses. Viewing distance and angle of field of view are genetic factors that are passed on from parent to offspring. Each animal has two eyes encoded by two input neurons for each recognisable object in the animal's neural network. The main loop updates the animals' view a certain amount of times per second. This is done by looping through the `living_animals_array` and selecting an animal whose view is to be updated. When an animal is selected by the main loop, the main loop loops through the `plants_array` and `living_animals_array` to check if any plant or animal is within the selected animal's viewing distance and field of view. If, for instance, a plant is visible in the left eye of the selected animal its distance from the selected animal is calculated. This distance is proportional to the strength of the left eye's plant-recognizing input neuron. See a simplified version of the code for this procedure is presented in section 5.2.

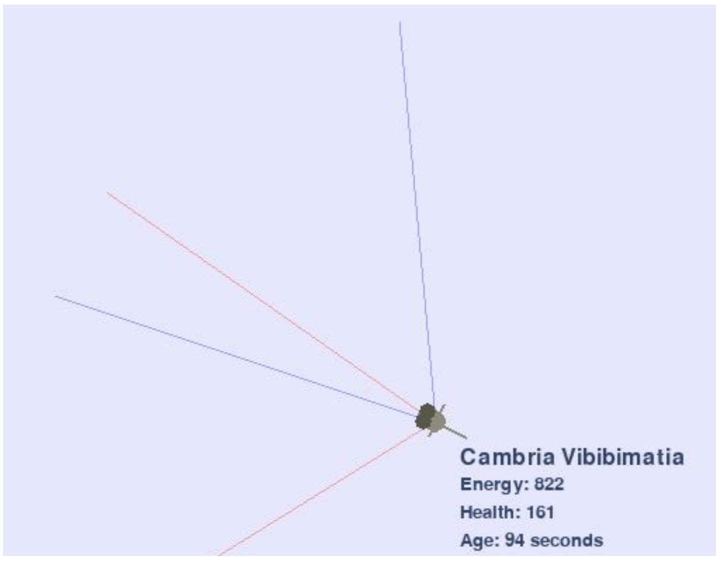


Fig 10, a selected animal with its field of views visible.

Hearing

An animal can hear in all directions. The 'hearing-range' is the same as for it's vision. There are therefore two input neurons, one for each side of the animal's body, and the strength of the input is decided by the speed of nearby animals within hearing range.

Pain, Energy and Health

When animals lose health they feel pain. The amount of pain an animal feels is equal to the fraction of its health lost. Furthermore, pain decreases with time, unless it suffers yet another injury. Health in Cambrians works in a similar fashion to health points in video games. If a given animal in Cambrians has low health it is close to death. Animals can lose health from either being attacked by another animal or starving to death. Finally, animals know how much energy they have at any given time, in other words how hungry they are.

Brain

As mentioned, the brains of animals are encoded by neural networks. Whilst the program's neural networks have been described, there are still a few unmentioned steps on how the actual neural networks are used, or in other words, how inputs from senses can lead to outputs or 'actions'. Below, the order in which animal brains' in Cambrians 'think' is listed:

1. The main loop calls the update function of every living animal. The update function handles all physical changes the animal is prone to. This includes everything from losing energy over time to adding time to the animal's age. When this is done, the main loop calls the hearing and seeing functions for each living animal. When this is finished all information is gathered and ready to be processed by the animals' neural networks.

2. When all the information is gathered, the main loop calls every animal's `think()` function. Inside the function, all sensory information is put into a list called `inputs`. To see the `inputs` list written in Python, see the appendix, section 5.3. The `think()` function then proceeds to feed the input values into the neural network: The `inputs` list becomes the input layer in the neural network. Next, with NumPy's dot multiplication method `dot()`, the input values are used to generate the hidden layer's values by multiplying the input values with the neural network's weights. From the hidden layer new connections, with new weights, leading to the output layer.

3. Once the output layer is determined, it's values are translated to actions that the animal will complete. Say for instance that an animal's output value for acceleration is 1. This means that said animal will accelerate at 100 percent of its capacity, whilst a 0 means that it will stand still.

Metabolism

Animals need energy to survive, if they lose all energy they begin to lose health until the point where they either gain more energy or die when their health reaches zero. An animal can replenish energy through eating either plants or corpses, depending on if the animal is a carnivore or herbivore. Herbivores can only consume plants and gain no energy from eating corpses, the energy gained from eating a plant is equal to the area of the plant in pixels multiplied by two. Carnivores gain energy from eating corpses of dead animals. When a carnivore manages to kill an animal and then proceeds to eat the corpse left by the dead animal it will gain energy equivalent to the area of the dead body plus the amount of energy stored in the killed animal before its death. All animals lose energy gradually over time and from movement. Faster movement and having larger bodies drains energy faster. Health is also slowly regained if an animal has energy.

Reproduction

Animals reproduce asexually. If an animal has as much energy as its total area in pixels and its `birth_timer` has reached 200 seconds since its last birthing cycle it will give birth. Ninety percent of its child's brain will mutate slightly whilst another smaller percent (defined by the parent's `mutations_rate` variable) with change completely. A lower `mutations_rate` means that the child is susceptible to larger changes in its neural network, vice versa. The child's size, eye variables and mutation rate can also mutate at a probability of circa ten percent for each attribute. When the child is born, the parent transfers the energy equivalent to half the size of its body in pixels to its child.

3. Results and Discussion

This chapter will analyse the results of Cambrians and discuss if the choices made during development were sufficient to create an environment prone to the creation of evolutionary successful animals, as well as a functioning artificial ecosystem.

3.1 Analysis and Discussion of Software

Observed Behaviours

All successful animals develop behaviours such as the ability to locate food. For herbivores, finding food often means locating plants and then, depending on their energy, either eating it directly or savouring it. Sometimes savouring food includes waiting in front of the plant until said animal's energy begins to get too low or until another animal threatens to eat the plant, other times herbivores would dodge plants to let others eat them. Generally, carnivores evolve the behaviour to either chase herbivores or to pounce at them if they come too close when it comes to locating food.

In population-dense environments, it is common for many species to develop a *scavenger behaviour*. Generally, these scavenger species tend to move backwards slowly, lunging towards any spotted dead corpses without ever attacking living animals. The slow movement is probably to conserve energy, as these species generally go long times without food.

All successful species developed basic *parenting behaviours.* Often times, these included quickly turning away from an organism related to themselves, as to not cause them any injury.

In one iteration of the program, some successful animals developed a basic *herd behaviour*. After approximately 30 mutations this herbivorous species developed a behaviour in which an individual of the species retracted its mouth when a related animal was directly in front of it, whilst moving towards that related animal. This resulted in the species grouping together, instead of dispersing and becoming vulnerable to carnivores. When an animal of the species did not have a related animal in front of it the animal extended its mouth. Furthermore, if a carnivore was spotted, animals of the species would quickly aim their mouths towards the carnivore. This created a protective barrier around the herd, as seen in Figures 11 and 12, which proved very successful for the survival of the species as a whole. This species was so successful that it outcompeted all other species in its iteration of the program, even though it is generally only carnivores that manage to do this. This behaviour showcases, at least partially, why so many of the herbivorous species in real life have developed this defensive behaviour.

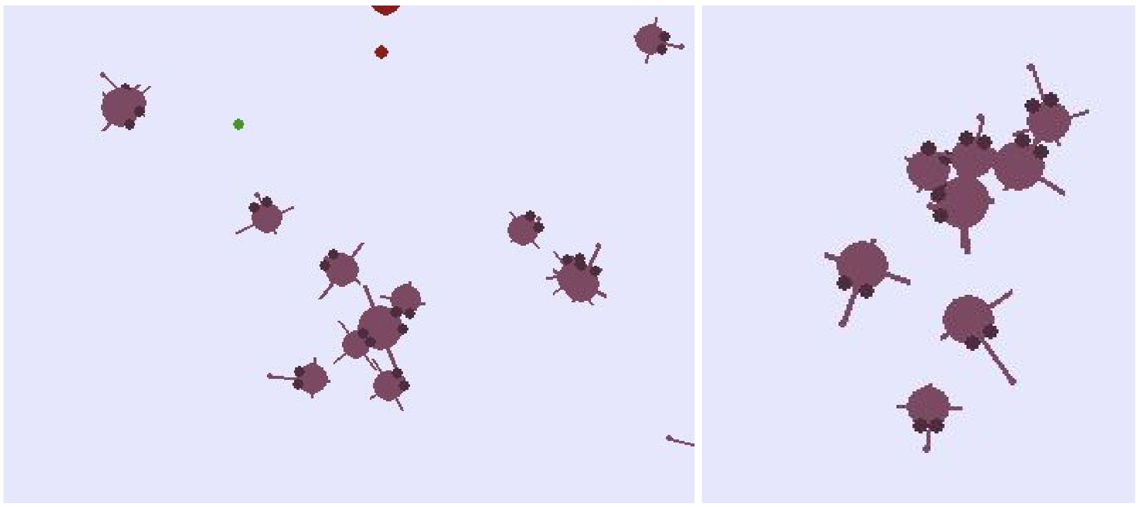


Fig 11 and 12, the species "Cambria Viloutus" forming different types of defensive herd formations.

Simulated Population Curves

When analysing population size over time an interesting observation was made. As for real life ecosystems there clearly seemed to be a carrying capacity in the software's generated worlds. Animals' populations would rarely grow or sink underneath a certain amount of animals drastically. To test if the populations' growth curves would change in appearance around the carrying capacity depending birth cycle length, one world was created with animals that had a five-second birth cycle and one world was created with animals that had a 180-second birth cycle. See the diagrams below.

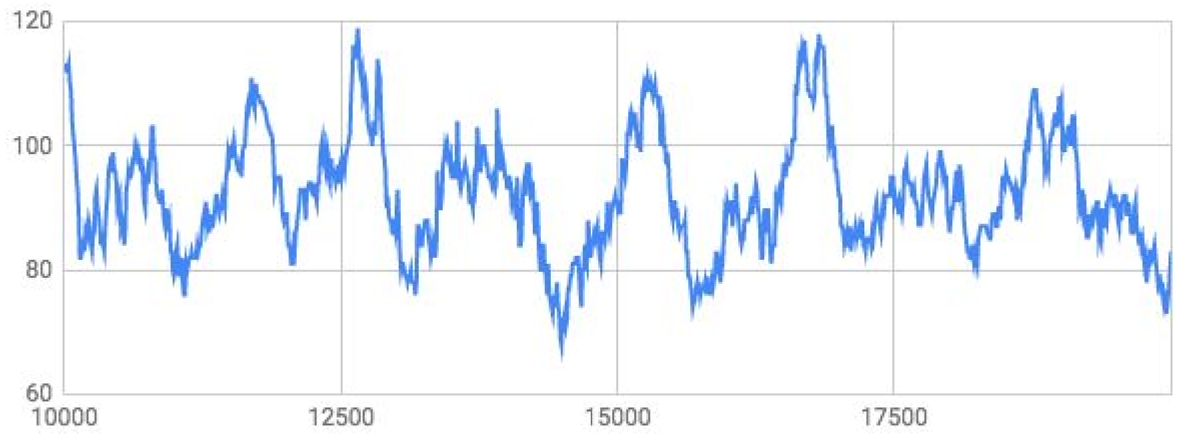


Diagram 3, population curve for species with a birth cycle of 180 seconds. The x-axis is the time in seconds and the Y-axis is population size.

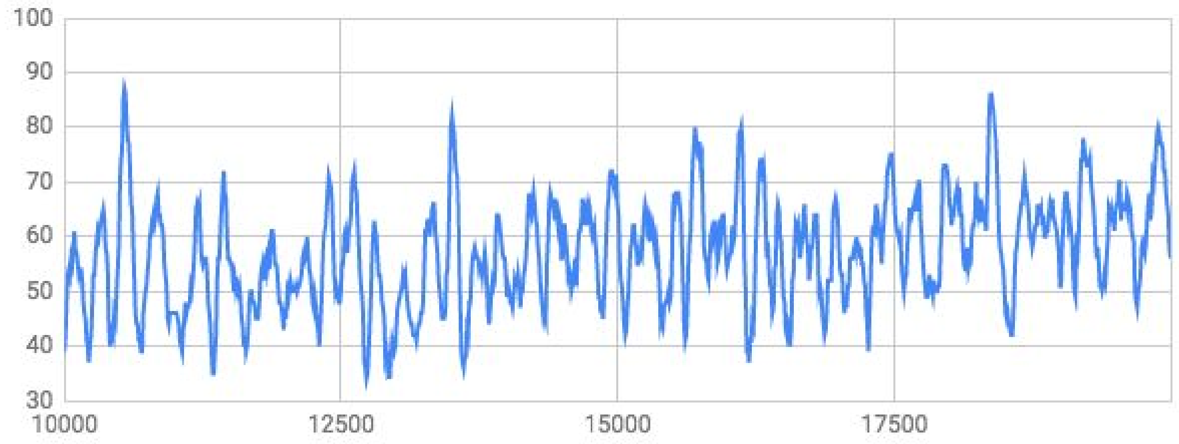


Diagram 4, population curve for species with a birth cycle of 5 seconds. The x-axis is the time in seconds and the Y-axis is population size.

It is clearly visible from Diagrams 3 and 4 that animals, in identical environments, differ largely in population growth depending on birth cycle length, similar to real life ecosystems.

Furthermore, these population curves demonstrate that the created environment does indeed have a carrying capacity, further drawing similarities between Cambrians's environments and real ecosystems.

Predator-Prey Cycles in Cambrians

After analysing the populations of carnivores and predators in Cambrians, similarities between Cambrians's populations and real ecosystems' populations were discovered. As seen in diagram 5, the carnivore's population size rarely grows larger than the herbivore's population size.

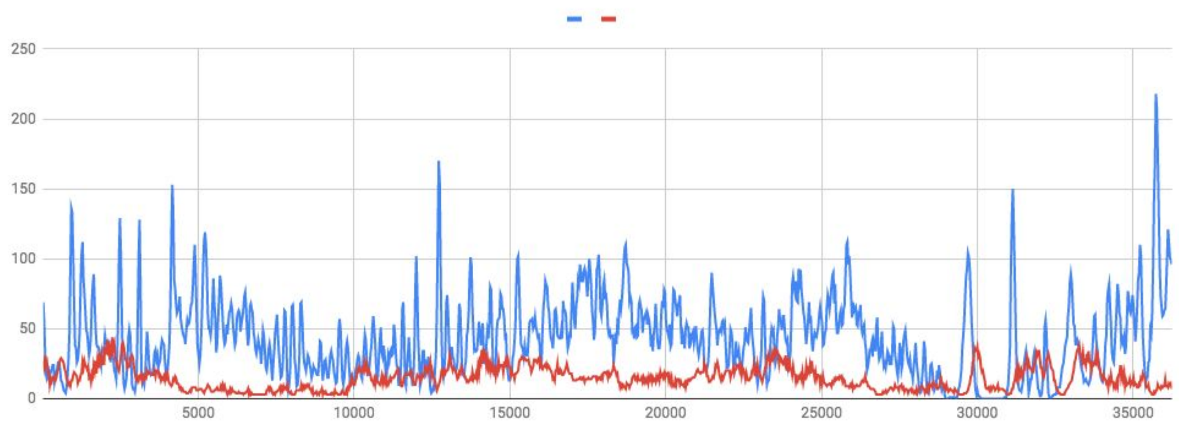


Diagram 5, Population of carnivores (red) and herbivores (blue) over the course of circa 10 hours. The x-axis is the time in seconds and the Y-axis is population size.

At certain points, Cambrians's generated predator-prey cycle follows real life predator-prey cycle's signature patterns such as for instance: Increased prey population leading to an increased predator (carnivore) population, which in turn leads to a decreased prey population with eventually leads to a decreased predator population, etc. See Diagram 6.

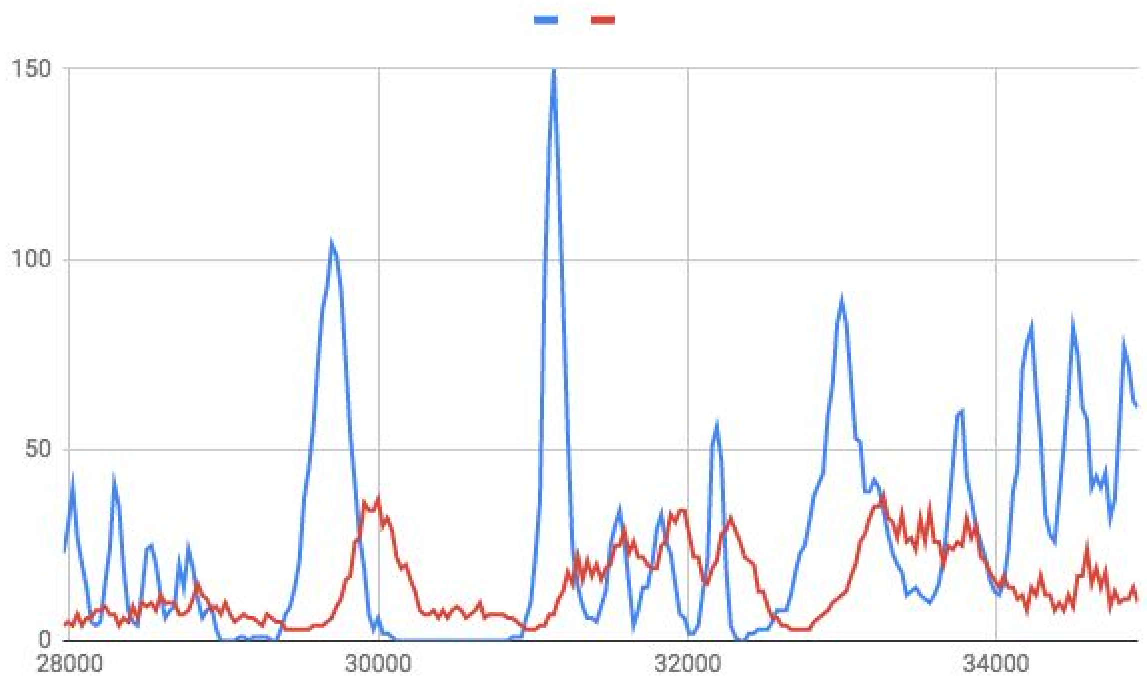


Diagram 6, Populations of carnivores (red) and herbivores (blue), zoomed in section of diagram 5.

3.2 *Final Discussion*

It is hard to say whether Cambrians completed its goal of creating an artificial ecosystem. Whilst Cambrians biotic components definitely interact, one could argue that there are no abiotic factors in the program at all. In real ecosystems, abiotic components include the sun, winds, mountains, etc. The only considerable abiotic factor in Cambrians is the world itself since it is cannot under any circumstances be recognised as living.

Animals life criterion in Cambrians	
Consists of cells	Whilst Cambrians's animals do not consist of biological cells, their bodies remind of real animals' organs that do consist of cells.
Can reproduce	Yes.
Metabolism	Yes.
Energy metabolism	Yes. Animals can manage energy, although not through chemicals such as ATP.
Information transfer	Yes. Animals transfer information to their offspring when reproducing.
Evolution	Yes.

Table 1, a showcase of which life criterion Cambrians's animals fulfil according to NE.[9]

Even though animals fulfill the majority of the life criteria seen in Table 1, they do it on an extremely simple level. Furthermore, the fact that plants and animals are not linked together through for instance nutrients cycles shows that complete ecosystems are not being created. The software does on the other hand share statistical phenomena with real life ecosystems. Additionally, the animals in Cambrians developed some similar behaviours to animals in real life ecosystems. Both these arguments point towards the fact that Cambrians has indeed created an ecosystem, or at least a simplified version of one. To conclude, it would be inaccurate to call Cambrians's generated worlds complete artificial ecosystems since too many deciding factors are missing such as nutrition chains, diverse abiotic and biotic components, energy cycles etc. However, since the software's worlds show key similarities to real life ecosystems, it would be safe to say that Cambrians can produce simplified simulations of ecosystems.

3.3 Final Reflections

As discussed, Cambrians's generated environments can not be considered realistic digital ecosystems and therefore it would be drastic to draw any real biological conclusions from the

[9] " *Nationalencyklopedin,* liv, https://www.ne.se/uppslagsverk/encyklopedi/l%C3%A5ng/liv. Retrieved 2019-01-19

program. However it begs the question: What if a more dynamic and well-structured program was written, running on a supercomputer? I believe that such a program could have a gigantic impact on the field of evolutionary biology.

Keep in mind:
The following text consists of personal thoughts and ideas collected during the development of Cambrians. Therefore, the text contains personal speculations on the future use of Cambrian-like software, rather than carefully considered scientific forecasting.

Evolutionary research is widely accepted in the scientific community, however, as of now it is hard to study evolution empirically as evolution generally takes place over long periods of time. Therefore, most evolutionary research is based upon historical evidence such as fossils and uncovered genetic material. However, if a powerful evolution simulator existed, which proved to be able to create biological systems similar to the real worlds', evolution could be tested in a way previously impossible. For instance, a digital single celled organism could be created and added to a digital environment, similar to environments that we believe life first developed in, and then have the program's 'clock' run thousands of times faster than real life, efficiently speeding up evolution for convenience sake. This way, hundreds of years of evolution could be 'done' in a matter of seconds, making it easier to draw conclusions on life's evolution. From a computer-processing aspect this may seem impossible considering the fact that such a program could contain millions (if not billions) of different variables and functions running many times per second. However, with general- and quantum computing constantly improving it may become possible. Furthermore, if optimization-functions are added such as genetic algorithms a great reduction of the processing power could be achieved, as seen in Cambrians. To take things a step further, imagine creating a simple digital organism in the program, allowing it to evolve into several different species in a digital environment similar to that of planet earth for millions of 'in-program years' and then trying to find self conscious and intelligent species such as ourselves. Whilst very controversial and science-fiction-like, this is a theoretical way to create super intelligent and benevolent AI, as an intelligent animal from the program could have developed herd behaviors, similar to ourselves, which for instance can include sympathy for other beings and compassion. The possibilities seem endless.

Finally, to mention my personal thoughts, it has been a great experience creating the program and writing about it. I have not only taught myself Python but I have also gained a basic understanding for artificial intelligence and machine learning. I hope you learned a thing or two about AI as well, or at least that you have been inspired to learn more! As for the program itself, I plan on creating a newer, improved version from the knowledge I have obtained from the development of Cambrians. My intention is to write the program in a more powerful programming language such as C++, and that the newer version might actually serve a purpose in real biological research.

If you are wondering anything else or have any questions, please feel free to contact me at *charlesalexander.maddock@gmail.com!*

4. Sources

Nationalencyklopedin, artificiell intelligens, https://www.ne.se/uppslagsverk/encyklopedi/l%C3%A5ng/artificiell-intelligens, Retrieved 2019-01-22

Frontiers, van Gerven, Marcel, Bohte, Sander (2017), https://www.frontiersin.org/articles/10.3389/fncom.2017.00114/full , Published on 19 December 2017, Retrieved 2018-02-20.

Mitchell, Melanie (1996). *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT Press. ISBN 9780585030944, Retrieved 2018-02-20.

Kuhlman, Dave, "A Python Book: Beginning Python, Advanced Python, and Python Exercises". Archived from the original on 23 June 2012, Retrieved 2018-02-20.

NumPy, https://www.numpy.org, Retrieved 2018-03-10

Nationalencyklopedin, ekosystem - Uppslagsverk - NE.se." https://www.ne.se/uppslagsverk/encyklopedi/l%C3%A5ng/ekosystem. Retrieved 2019-01-19

Nationalencyklopedin, bärförmåga, http://www.ne.se/uppslagsverk/encyklopedi/lång/bärförmåga, Retrieved 2019-01-22

Nationalencyklopedin, liv, https://www.ne.se/uppslagsverk/encyklopedi/l%C3%A5ng/liv. Retrieved 2019-01-19

5. *Appendix*

5.1 Simplified Code for Drawing an Animal's Flagellum

Using Pygame's line function, two points, a colour and a width are specified to create the flagellum, for instance. The first point for the line is the animal's centre, or in other words its position. The second point is proportional to the animal's speed. To find this point, Python's native trigonometric functions are used. The trigonometric calculations are done in the animal's update function and then, afterwards, the line is drawn in the draw function. The line's colour is the same as the animal's colour. See a simplified version of the code below.

```
import math
import pygame

def update_animal(self):

   self.flagellum_len = self.sizeRadius + self.speed

   self.flagellum_x = self.x + math.cos(self.rot * self.flagellum_len)

   self.flagellum_y = self.y + math.sin((self.rot * self.flagellum_len)

def draw_animal(self):

   pygame.draw.line(
      self.color,                            #color
      (self.x, self.y),                      #point 1
      (self.flagellum_x, self.flagellum_y),  #point 2
      self.flagellum_width )                 #width
```

This general structure of calculating and then drawing applies to all visual aspects of the animal.

5.2 Simplified Code for Animals’ Vision

```
#Inside the main loop
for animal in living_animals_array:   #Checking each living animal

      for plant in plantsArray:       #Checking if any plant is within
                                      #viewing distance
          dx = animal.pos.x - plant.x

          dy = animal.pos.y - plant.y

          distance = math.sqrt(dx * dx + dy * dy)

          if distance < animal.view_distance:

             animal.check_eye_collision( distance, plant, 'l', 'p' )

             animal.check_eye_collision( distance, plant, 'r', 'p' )


#Inside the animal’s check_eye_collision function
def check_eye_collision( distance, obj, eye, obj_type): #obj = plant

   if eye == 'l': #Plant is in left eye

      if obj within animal’s FOV:    #Extremely simplified if statement

          if obj_type == 'p':        #Check if viewed object is plant

                self.left_eye_plant_intensity += ((self.view_distance -
distance) / self.view_distance)     #Input value to neural network

          [...] #The function checks if viewed obj is something else
                # ex. obj_type == ‘ak’ (animal_kin, or related animal)
```

5.3 List of Input Values for Animals' Neural networks

```
def think(self):
      inputs = [
      self.left_eye_plant_intensity,
      self.right_eye_plant_intensity,
      self.left_eye_corpse_intensity,
      self.right_eye_corpse_intensity,
      self.left_eye_herbivore_intensity,
      self.right_eye_herbivore_intensity,
      self.left_eye_predator_intensity,
      self.right_eye_predator_intensity,
      self.left_eye_kin_intensity,
      self.right_eye_kin_intensity,
      self.left_ear,
      self.right_ear,
      self.pain,
      self.energy / (pow(self.sizeRadius, 2) * 3.14) * 2,
          #a fraction for how much energy the animal has

      self.health / (pow(self.sizeRadius, 2) * 3.14),
          #a fraction for how much health the animal has

      (self.age) / 500, #500 seconds is the max age
          #a fraction for how old the animal is
      ]
```

5.4 Fitness Calculation

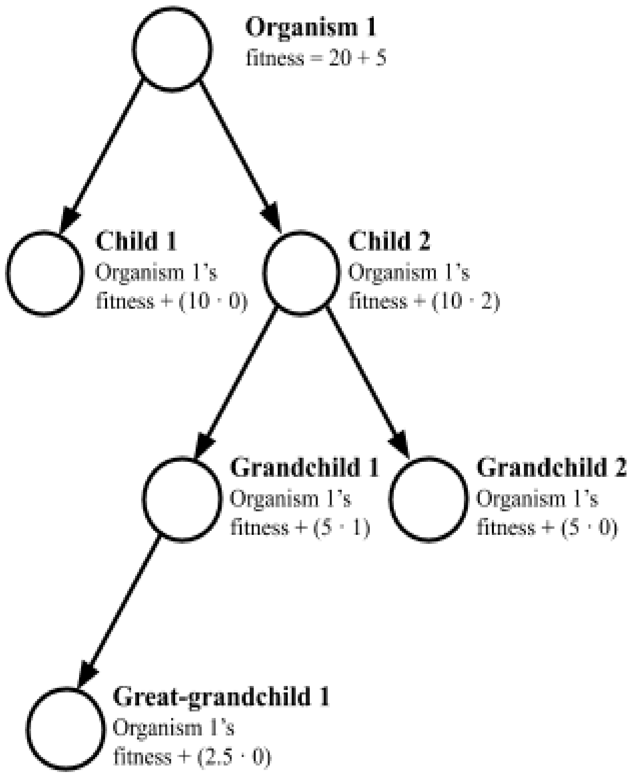


Fig 12, example how the fitness for "animal 1" is calculated.

In this case, "Organism 1" gives birth to two offspring. Since one of its offspring, "Child 1", did not give birth to any offspring, it gives zero "fitness points" to its parent, "Organism 1". Organism 1's second child, "Child 2", has given birth to two offspring. This results in Organism 1 gaining 20 fitness, since it receives ten points multiplied with the amount of offspring its child had, which is two. One of Child 2's offspring, "Grandchild 1", has a child of its own, resulting in Organism 1 gaining five fitness points.

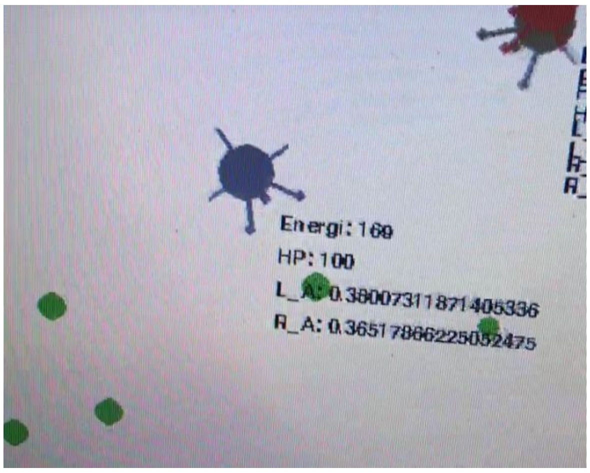
5.5 Development of Cambrians

Initial Implementations

The project's first challenge was deciding how to structure both visually and technically. Considerations to use Python's built-in visual library "Turtle graphics" for displaying and running Cambrians were made. However, it felt limiting, so Pygame was used instead. After installing Pygame the display, a bright blue rectangle (the world) and the main loop were created. With the foundation working, the `Plant` class was added as well as the `create_plant` function. The `create_plant` function simply creates a new plant object at a random point in the world and then appends it to the `plantArray`. The `plantArray` is essentially a list containing all existing plants. If a plant is inside the `plantArray` it will be updated and drawn several times per second from the main loop, if it is removed from the list however it is deleted permanently from the program. In the main loop, a timer is added that calls the create_plant function at an interval of time decided by a variable called `plant_timer_max`. The plant class is divided into three functions: The `__init__` function which creates new plant objects, the update function and the draw function.

First Version of the Animal

With the world and some plants implemented the next goal was to add animals. The `Animal` class is created. Similarly to the `Plant` class, the `Animal` class initially consists of an `__init__` function, an update function and a draw function. The `animalArray` is also created to store animal objects. Before any AI is implemented, manual control of the animals is added for testing purposes. In the animal's update function, code allowing for movement is added, essentially updating the animal's x and y coordinates in accordance with the user's keyboard input. The draw function takes in the animal's position and rotation and draws a circle at the said position, with the circle's radius being equal to the animal's radius and the circle's colour is equal to the animal's colour. This is was the first version of the animal. The next step was to add antennae and limbs to the animal. This was done by creating different lines: Two short lines facing forward representing the antennae, two short lines along the sides of the animal representing fins and one long line at its back representing the animal's flagellum. Further details on how limbs can be drawn with pygame are available in the appendix, section 5.1.



```
Energi: 169
HP: 100
L_A: 0.380073118714053336
R_A: 0.365178662250552475
```

Fig 13, testing antennae on an early version of the animal.“L_A” and “R_A” stand for left antenna intensity and right antenna intensity. The intensity is proportional to the distance between plant and antenna.

Development of Senses

The first added senses were antennae that could sense nearby plants. However, after some thought, it was decided that antennae were limiting for the animals’ evolution. For instance, antennae restrict animals from being able to evolve directed vision. This restraint would, for example, make it impossible for carnivores to develop vision pointing straight forward, towards prey. This argument, amongst others, was the reason why eyes where implemented. In the beginning, eyes were limited to recognizing plants, related animals and unrelated animals. Later on the ability for recognising unrelated carnivores and herbivores was implemented. This update drastically increased the dynamics of animals, as certain herbivores would, for instance, flee when seeing carnivores whilst acting aggressively towards other herbivores (presumably due to competition for resources). It was therefore decided to add more senses such as hearing, hunger and pain.

Bugs

Ninety per cent of the time the errors were typos or simple careless mistakes. A few bugs were, on the other hand, Python-related bugs. One of the program’s most devastating bugs had to do with copying in Python: When animals’ give birth they essentially create an independent, mutated copy of themselves. Before the bug-fix independent copies were not created at reproduction, instead, all animals and their variables referenced themselves. This led to occasional randomization of all the brains in an entire related species, effectively erasing all behavioural progress said species had made instantly. When this was discovered, Pythons `copy` library was implemented. The `copy` library, with its function `copy.deepcopy()` allows for the creation of independent copies in Python. This solved the bug.