Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

A Framework for Algorithmic Learning: How Does Complexity Arise in Evolution?

Group Project

Aleksandar Terzic, Thomas Buob

terzica@student.ethz.ch, tbuob@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Ard Kastrati
Prof. Dr. Roger Wattenhofer

July 2, 2021

Acknowledgements

We would like to thank our thesis supervisor Ard Kastrati for his continuous support and insightful ideas throughout the semester, especially during the busy days leading up to the submission deadline.

We would like to thank the DISCO group for giving us the opportunity to try our hands out in an area we were both unfamiliar with by working on a fun and creative project idea.

Abstract

In this Group Project we have created an artificial life system in Python which serves for the evolutionary search for algorithms which are able to compute Boolean functions of two inputs. The ideas which drive the system are the same ideas believed to govern evolution in nature. Based on existing work in the form of the "Avida" open source software platform, we have written an easy-to-use system which simulates evolution in Python, have expanded upon the original work by adding new instructions, and have analyzed the performance of the system on from-scratch evolution of Boolean functions.

Contents

Chapter 1

Introduction

Artificial Life is a term coined by the American biologist Christopher Langton, and is a field of study of phenomena related to natural life and genome evolution through hardware, software or biochemical models. It primarily focuses on answering questions in the field of study of biology. Many interesting works exist on the topic, including the one that served as the point of reference to which we compared our system, [1], in which the evolutionary origin of complex organism features was examined by simulating evolution of "Digital Organisms" that execute instructions from a basic instruction set.

Our work turns away from trying to answer questions about natural evolution and rather focuses on exploiting the rules thereof with the goal of finding algorithms capable of calculating different Boolean functions. For this purpose we have created and analyzed a system which simulates a Petri dish full of microorganisms capable of self-replication and evolution through random mutations.

The report here presents the main ideas behind our system, the main design choices, obstacles we have faced and overcome and the lessons we have learned from them, and finally ends with experimental analysis of complex feature evolution with varying system parameters.

Chapter 2

Avida

The main inspiration for our work is an existing system called "Avida". It is an artificial life software system, written mainly in C++ and Objective-C, which is based on the ideas which are believed to govern evolution, such as survival of the fittest and the idea that complex organism features are the result of a long line of random mutations which happened to be beneficial, starting from a very simple ancestor a very long time ago. It is primarily used for research in evolution, which is obviously infeasible on most living organisms because it happens on a very large time scale, much longer than the average life span of an evolutionary biologist. Artificial life systems are therefore the natural choice for trying to answer questions about evolution, but one must be careful that the system is well defined, otherwise the answers such a system is able to provide have nothing to do with the reality of things.

To the best of our knowledge, Avida is one such well-defined system. The typical Avida experiment starts with a so-called digital organism which cannot compute anything. This organism is then left to self-replicate and through random mutations evolve to compute various functions of its inputs, and is then appropriately rewarded for its computation depending on the complexity of the computed function. We will now explain the main ideas of the system without going into too much detail; We will later explain more details when we go through our own implementation, and they can also be found in references [2] and [1].

2.1 Digital Organisms

The Digital Organisms can be thought of as small self-replicating computers, each with their own memory, three registers, two stacks, IO buffers, and several heads which point to locations in memory, such as the instruction head and flow control head, but also the read and write heads whose purpose is to control self-replication.

The memory of the organisms is strictly instruction memory, and all numerical values are stored in either the stacks or the registers. The instruction memory, often denoted as the **genome**, is composed of basic instructions such as "increment", which increments the value stored in a particular register, "push to stack", "pop from stack", and so on. It is a basic but very cleverly designed instruction set, in which nothing can go wrong in the following sense: Any possible combination of instructions composes a valid algorithm. Now, whether that algorithm does something of value or not is a different story, but no instruction can ever be put in any location such that it would "break" the system. This idea is extremely important for evolutionary search, as such a search is based on randomness, and so we can't allow for a random mutation to break anything at any point in the system.

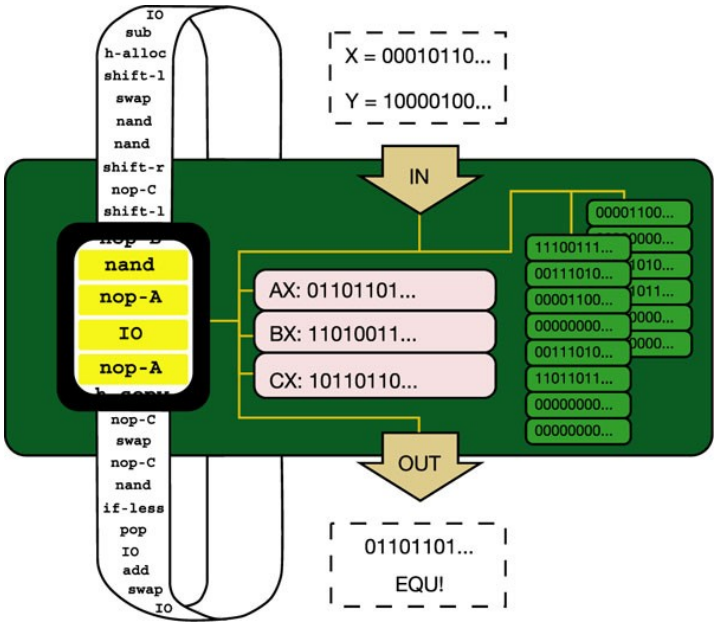

Figure 2.1: A Digital Organism in Avida

The default self-replicating organism, which is the typical starting point of the system, is not as naive as it sounds. For an organism to replicate itself, it needs to:

- Allocate new memory into which it can copy its own instructions
- Position three of the four available heads in the exact correct locations such that it can copy its own memory content into newly allocated memory
- Execute the "H-DIVIDE" instruction, which splits the organism in two pieces, the location of the split depending on the positioning of the heads from the previous point

We omit the details of everything that is happening here as it is not the focus of our work; it suffices to say that the process described above models **asexual reproduction** found in simple organisms in nature. Firstly the organism copies its own genome, such that at the moment right before it divides, the genetic material contains both the original genome and a copy thereof, which is the basis of the offspring's genetic material. It then divides to produce two equal organisms, ignoring mutations for now.

Self-replication is the only functionality of the default organism. It can not calculate anything, nor does it do any input/output. Its structure is such that in the beginning of the genome we have instructions necessary for self-replication, then we have a certain number (by default 35) "none" instructions, which do not do anything, and at the end of the organism we again have a certain number of instructions which are part of the self-replication mechanism.

Asexual reproduction is a quite delicate system, and it is interesting to observe how it necessarily survives through the generations and how it itself mutates, often becoming shorter and therefore more efficient than it is upon initialization.

2.2 Evolution

The main driving force of evolution is the following: each time an instruction is copied into the allocated memory as explained above, with a certain probability it is copied erroneously, such that a "none" instruction described above may turn into a "nand" instruction, available in the instruction set, and we suddenly have an organism which is capable of computing the NAND of two numbers.

Additionally, evolution happens upon division, where with a certain probability a random instruction from the instruction set is inserted into a random location in the organism, or else the instruction at a random location is deleted from the organism.

Through several generations of mutations, in [1], organisms which can compute all of the Boolean logic functions listed in the table below were evolved.

Function Name:	Logic Function:	Reward:
NOT	$\neg$ A; $\neg$ B	2
NAND	$\neg$ (A $\wedge$ B)	2
AND	A $\wedge$ B	4
OR_N	(A $\vee \neg$ B); ($\neg$ A $\vee$ B)	4
OR	A$\vee$ B	8
AND_N	(A $\wedge \neg$ B); ($\neg$ A $\wedge$ B)	8
NOR	$\neg$ A $\wedge \neg$ B	16
XOR	(A $\vee \neg$ B) $\vee$ ($\neg$ A $\vee$ B)	16
EQU	(A $\wedge$ B) $\vee$ ($\neg$ A $\wedge \neg$ B)	32

Table 2.1: Logic functions whose evolution was investigated, and the rewards in form of multiplicative factors applied to the metabolic rate for computing them. Symmetrical operations, shown separated by a semi-colon, are treated as the same function. No added benefit is obtained for performing any function multiple times ([1])

We can now imagine something akin to a Petri dish full of microorganisms which self-replicate and evolve in a fully random fashion. At a certain point, one of these organisms may discover fire, or in our case discover how to compute a Boolean NOT of an input. How, then, is such an organism rewarded for its discovery?

To explain **the reward system**, we need to take a step back away from the details of the organisms themselves and look at an entire collection of them. The standard Avida experiment contains a 30×30 up to 60×60 size pool of organisms, all independently self-replicating, evolving, and possibly also computing logic functions. The organisms execute their instructions in a quasi-parallel manner, with the help of a **scheduler** which lets each organism execute a certain number of instructions, the number being proportional to the organism's metabolic rate.

The metabolic rate, sometimes also denoted as the fitness level, is an attribute of digital organisms which indicates their computation capacity. The smallest metabolic rate in the pool defines the baseline rate, and the number of instructions that an organism is allowed to execute any time it is scheduled is equal to its own metabolic rate divided by the baseline rate. So, if we for example have a 1×3 pool with the rates [1,3,5], in each scheduler run over all the organisms, the first organism will execute one instruction, the second organism will execute three instructions and the last organism will execute five instructions.

Upon organism initialization, its metabolic rate is equal to its genome length. If it however so happens that an organism proves capable of computing a boolean function of its inputs, **it is rewarded in the following manner**: Upon division, the metabolic rate of the child and the metabolic rate of the parent itself are multiplied with the appropriate reward factor, as in Table 2.1. This means that the organism is not immediately rewarded for a computation as soon as it happens, but rather only upon division, to discourage dead-end organisms which can compute and output functions but are unable to self-replicate and so evolve further.

2.3 The Complexity of EQU

In [1], 30 experiments were run on a pool of size 60×60, each experiment starting with the default self-replicating organism, and in each case the most complex investigated logic function, bitwise equivalence, was eventually evolved.

Initially it may be perplexing why bitwise equivalence of two numbers would be a difficult function to compute, and why we would need evolutionary search in order to find it. We have to remember that we are working within the confines of a very basic instruction set, in which the complexity of the instructions does not go much further than "compute the NAND of values in register B and C and save the result to register B". To give you an idea of what an organism needs to do to compute bitwise equivalence, a custom-made program in the Avida instruction set which inputs two numbers and outputs their bitwise equivalence is presented:

1. Save input to register B
2. Save input to register C
3. Push both registers to stack, swap active stack
4. Compute NAND of registers B and C, store it to C
5. Pop empty stack to register B
6. Store register C to stack, swap active stack
7. Pop from stack to register A, pop from stack to register C
8. Calculate NOT(B), store it to register C
9. Push register C to stack
10. Push register A to stack
11. Pop stack to register C
12. Calculate NOT(A) to register C
13. Pop NOT(B) from stack to register B
14. Compute NAND of registers B and C, store it in register C
15. Swap stack, pop top of stack to register B
16. NAND registers B and C, store value in register A
17. Output value stored in register A

This results in a list of 45 perfectly coordinated instructions, changing any of which ruins the organism's ability to compute bitwise equivalence. It should also be noted that this is even without the self-replication mechanism, usually around 15 instructions long. If we imagine that we start with organisms which compute nothing at all, it is obvious that there is a long way to go from nothing at all to the EQU program presented above, but with the well-defined principles of evolution, it happened consistently over 30 experiments in [1].

Chapter 3

Our Implementation

Based on [2], we spent the semester recreating the Avida system with most of its basic functionalities in Python, in such a way that it can easily be used in quite the same manner as one would use a standard scikit-learn library for machine learning tasks. What we hoped to achieve by doing this is to create a much simpler system than Avida is originally, one which is easy to use and expand in the future, and one which would hopefully prove insightful for future works on the same topic.

We have succeeded in creating one such system which shows behavior very similar to that of Avida, and which can additionally be expanded with new instructions (which we did, and whose effects we have analyzed and presented in the "Experiments and Observations" chapter). The system could do with some optimization, however all of the main components are there and we do indeed observe well-behaved algorithm evolution. This part of the report concerns itself with the design decisions behind all of the main components of our system, how they interact, and the obstacles we have faced during development. We will go through the main components of our system and for each one note the most important lessons we have learned while working on it.

3.1 Digital Organism

Digital Organisms are the basic unit of our system, and their purpose is modeling hardware in which integer values are stored and in which all of the calculations take place, as well as keeping track of some attributes which are necessary for evolution. The functionality of the fundamental units of our system is split between the CPUEmulator and the CPU classes, CPU being the more basic of the two.

CPU models the basic hardware components of Avida organisms. It is the lowest level of abstraction we have in our system, and as such its complexity turned out to be very low. All of its components are modeled through separate classes and are listed on the following page:

- Three registers, denoted as register A, B and C
- Two stacks, only one of which is active at any time
- Two FIFO buffers, one of which serves for input values, one of which for output values

On its own, an instance of the CPU class is just a collection of integer values and is as such, in the context of algorithm evolution, useless. It only obtains its meaning when coupled with the class introduced next.

CPUEmulator models a fully functional digital organism. It contains in itself an instance of the CPU class, instruction memory, the four heads which serve as pointers to different locations in the memory, all of the functions and attributes needed for executing instructions which manipulate the virtual hardware, along with attributes which help us define and keep track of organism evolution.

Each CPUEmulator keeps track of three parameters which most heavily influence the evolution of organisms through the generations. These parameters are:

- Copy Mutation Probability – The probability that an instruction is copied erroneously during self-replication
- Insertion Probability – The probability of a random instruction being inserted at a random position into the offspring organism upon self-replication
- Deletion Probability – The probability that the instruction at a random position is deleted from the offspring organism upon self-replication.

The organism genome is contained in the memory of the CPUEmulator. This is perhaps also the most important part of the CPUEmulator, as it defines the algorithm it is capable of running.

Instructions can be thought of as letters in an alphabet, the alphabet in our case being simply a range of integers. As such, the memory of an organism is simply a list of integers in a certain range; Originally from 0 to 25, including both, as there are originally 26 instructions in the instruction set. The instructions only obtain their meaning when they are loaded into a CPUEmulator by using the load_program() function. Upon loading a program, the state of the CPUEmulator is fully reset and CPUEmulator.instruction_memory then contains objects which represent instructions that are linked to this particular emulator and can modify its state when executed via the CPUEmulator.execute_instruction() method.

One final thing worth mentioning about CPUEmulators is the “mediator” attribute. In the context of evolutionary search for algorithms, we abstract away from looking at a single CPUEmulator and instead initialize a whole bunch of them and let them self-replicate and mutate. As such, the organisms “live” in an environment with which they interact as they execute instructions and evolve. In order to reduce coupling between different components of our system, we turned to the Mediator design pattern, which defines a function which different components can use to communicate with each other without them necessarily knowing the details of the inner workings of each other. The mediator is then an interface between a CPUEmulator and the Environment in which it lives, which defines a simple “notify” function which an organism can use to communicate important events to the environment it is linked to. The exact workings of this mechanism are explained in one of the following sections.

3.2 Instructions

Next we turn our attention to the instruction set. Separate classes for all of the twenty-six instructions from the Avida default instruction set were defined, as well as two new ones. The instructions are all fully defined via only two functions:

- The constructor, which saves a reference to a CPUEmulator such that the instruction is able to directly manipulate it
- execute(), which when run executes the functionality of the instruction and modifies the CPUEmulator it is linked to

The entire instruction set is defined and explained in [2]; In our project we have a reconstruction of it in Python, in a way that is compatible with our definition of CPUEmulator. We omit further discussion of the instruction set as it is not the focus of our report.

As mentioned earlier, two new instructions which made sense in the context of artificial life and organism evolution were defined. They do not directly manipulate CPUEmulators; Their entire functionality is relegated to the environment the digital organisms live in. The newly defined instructions and their functionalities are:

- Move, which causes an organism to move in a random direction
- Sexual Reproduction, which causes two organisms to combine their genomes and together create an offspring

Sexual reproduction went through several iterations until a form which proved not to ruin evolution was found. It is defined in the following simple manner: Upon executing the "Sexual Reproduction" instruction, the organism is paired with the fittest organism in its neighborhood, with the probability of successful

sexual reproduction arbitrarily chosen as $\frac{1}{32}$. If sexual reproduction is successful, a new offspring organism is created by taking one half of the genome of one parent and one half of the genome of another parent. The resulting organism is then put into a free spot in the neighborhood of the parent which initialized sexual reproduction, if there is one. If there are no free spots in the neighborhood, the offspring is put into the place occupied by the weakest organism in the neighborhood, weakest being the one with the minimum metabolic rate.

Designing digital organisms and the instruction set was a great learning experience. Here we had our first contact with real-world object oriented programming and design patterns. The system we initially designed that was not following the principles of object-oriented programming had to be fully redefined. To give you an idea, what we started with was one single class, CPUEmulator, and this class contained all of the basic attributes as well as full definitions of every single instruction from the instruction set. It was a highly complex and non-modular piece of software. With our supervisor's guidance, we transformed it into many smaller components in such a way that they can be individually changed and expanded without worrying about a small modification breaking the entire system.

3.3 World

Next we turn our attention to the World class, a fairly complex system which regulates all interactions between organisms, is responsible for scheduling instructions, and most importantly of all, serves as a home for Digital Organisms.

An instance of the World class serves as a Petri dish in which organisms are left to live, replicate and combine, and at the same time is responsible for enforcing all of the rules which were implemented in order to model evolution.

The most important attributes and parameters any World instance contains are the following:

- Pool, an $N \times N$ array of Digital Organisms
- Rates, an $N \times N$ array of the metabolic rates of organisms
- Ages, an $N \times N$ array of the ages of organisms
- Inputs, an $N \times N$ array of tuples which keeps track of the two most recent inputs given to any organism
- Replacement strategy, which defines the position any offspring organism is placed in
- Insertion, deletion and copy mutation probabilities, which any organism in the pool inherits
- Instruction set, either "default" or "custom"

The most important functions which the World class implements are the following:

- The **schedule** fuction, which goes over all of the organisms in the pool and schedules the appropriate number of instruction executions on all of them
- A collection of **reaction** functions which define how the world reacts on various notifications from the organisms which live in it (for details see the "Mediator" section)
- A collection of **organism placement** functions, which place an organism at either a random or a specified location in the pool and correctly initialize all of the associated attributes.

The World is responsible for enforcing most of the principles which govern life and evolution, such as:

- Modeling and tracking organism fitness
- Modeling the natural death of organisms
- Modeling cosmic ray mutations

1: Modeling Organism Fitness

It is obvious that if we are going to try to emulate evolution, we need to ensure that there is a notion of fitness in play, such that there is a tangible reward for higher quality organisms. Organism fitness is the main principle behind well-behaved evolution. In our implementation, organism fitness is modeled as an integer value called the *metabolic rate*. As explained in Chapter 2, it determines how many virtual CPU cycles are awarded to an organism each time it is picked for execution by the scheduler.

The scheduler is simply a loop which goes over all of the organisms in the pool and executes on each organism the number of instructions which is equal to the organism's rate divided by the World's baseline rate, the latter one being the smallest non-zero rate in the World.

The World is responsible for keeping track of the rates of all of the organisms in the pool, and upon scheduling enforces a simple variation of the rule given above. Namely, in our rule, each time an organism is scheduled for execution, it is allowed to execute at most 32 instructions, if it so happens that the organism's rate is greater or equal to $32 \times Baseline\,Rate$.

For organisms whose rates are more than 32 times smaller than the largest rate in the World, upon each scheduler run they execute one instruction with the probability of their own metabolic rate divided by the largest metabolic rate in the pool.

This may seem unnecessary at first, but with our rule we believe we are ensuring more uniform evolution. Consider the following situation: The minimum rate in the World is 1, the maximum rate is 2000. The 2000 rate organism then gets to execute two thousand instructions every time it is picked for scheduling, while the organism with rate 1 only gets to execute one instruction. A likely result of the execution of two thousand instructions are dozens of self-replications of this organism in a single scheduler run, which may fully annihilate the organism's neighborhood, seeing as upon self-replication the offspring is placed in the neighborhood of the parent, which, unless there are free spaces, results in the death of the organism which was previously there.

Without the modified rule, the powerful organism may act as an overly efficient factory, potentially giving birth to and simultaneously killing its own offspring every single time it is picked for scheduling. With the modified rule, instead of a single run, the powerful organism executes two thousand instructions over a large number of scheduler runs. In each of these runs the less powerful organism now at least has the chance execute an instruction. This effect is increased with increasing world sizes.

2: Modeling Natural Death

The age of an organism is defined simply as the number of instructions the organism has executed in its lifetime. An important design choice in our system has been the introduction of natural death. As we are trying to, among other things, also model life, it follows common sense that we would introduce natural death. Surprisingly though, it has proven to be one of the most important principles necessary for *well-behaved* evolution.

Natural death is modeled by a very simple principle: If an organism's age gets larger than a certain maximum value, we assume that it has shown us all that it had to show us to and it is time for it to pass on. The exact definition of "a certain maximum value" here was chosen to be: $32 \times Genome\,Length$

A natural question now is, why did such a simple principle have a large impact on well-behaved organism evolution?

During the development of the system there was a large problem that haunted us throughout, and the introduction of natural death proved to be an efficient way of solving it. The problem was the following: **Organisms were able to "cheat" their way through to obtaining large metabolic rates while actually being completely incapable of any sensible computation.**

This is something that has become obvious to us during trial runs. One example of how this can happen is the following:

In any experiment, there is a large number of organisms (900 in our experiments) continually executing instructions. One of these instructions is the "IO Instruction", upon whose execution an organism outputs the value stored in one of its

registers. As we can imagine, if we start with default organisms which don't yet compute anything, many of these organisms will have empty registers and upon running the IO instruction will simply output 0.

The organisms work with 32-bit numbers, and as it happens the Boolean AND of two 32-bit numbers has a $(\frac{3}{4})^{32}$ probability of being 0, around 0.01%. It is also of course true that Boolean NOR of A and B can be rewritten as a Boolean AND of the negations of the two. With many generations of organisms in a pool which supports 900 of them, all of them continually getting random 32-bit inputs and outputing numbers stored in their registers, at some point an organism is not unlikely to simply output 0 and for this 0 to be the actual correct result of the Boolean NOR of the organism's last two inputs. When this happens, the organism is rewarded, and rightfully so, for it did correctly compute a Boolean function, but it is obvious that such organisms are undesirable.

Without natural death, such organisms were left to run for a long time, executing many more instructions than they should rightfully be entitled to because they are fully incapable of any computation. In a very young organism pool, if the above situation with NOR happens, they would execute sixteen instructions for every one instruction executed by the rest of the pool, and so would all of its potential children. We can imagine this being quite problematic as we would then have a cluster of powerful organisms producing lots of offspring, and it would be quite difficult for other organisms to eliminate this cluster through self-replication because if they are not particularly fit more likely than not it will be they who will be eliminated and not one of the organisms from the powerful cluster. Effectively, part of the pool is now biased towards algorithms close to the one which just outputs 0.

With natural death, even though such an organism will still be rewarded, in the worst case scenario it will hog the CPU cycles for the number of scheduler runs which is equal to its genome length, after which it meets the Grim Reaper and is forever forgotten. When this organism and its potential children die off, balance is again restored in the organism pool.

We can make no claims on whether natural death is the best way of solving this problem, but in our trial runs it was indeed observed that it solved it, and it is a solution which goes along very nicely with the general theme of simulating life.

3: Modeling Cosmic Ray Mutations

There has been a lot of research in evolutionary biology on the influence of cosmic ray mutations on organism evolution. These are gene mutations which result from irradiation of living cells by electromagnetic waves of a certain power and frequency. In a lot of cases cosmic ray mutations are detrimental, as they, well, may result in cancer. However, it being a source of mutations and mutations being the source of evolution, neglecting cosmic ray mutations may be equivalent to neglecting a powerful driving force behind evolution.

Cosmic ray mutations, also called point mutations, are defined in the following manner: Periodically (by default every 200 updates), each organism in the pool has a 10% chance of having one instruction at a random position in its genome be replaced by another randomly chosen instruction.

Just like in real life, these mutations are often likely to incapacitate the organism, but we cannot neglect the likelihood of a cosmic ray mutation being the source of complex feature evolution. Additionally, cosmic ray mutations serve as an outside push to the system, as it can often be experimentally observed that the organism pool stagnates in a certain configuration, with many organisms being able to compute different functions while seemingly not caring to bother to evolve further for long periods of time.

World's Other Duties:

Above we have listed the main principles which govern evolution and which have provided us with a system which reliably evolves complex organisms from default ones. Apart from enforcing these principles, the World has several other very important duties:

- It initializes organisms in its pool and constantly keeps track of many of their attributes
- It is responsible for initializing an offspring organism upon self-replication and choosing a spot in the pool where to put it in
- It keeps track of all of the IO operations happening in the pool and rewards the organisms which computed Boolean functions accordingly
- It manages all instructions which have to do with two organisms interacting with each other. This includes our custom instructions for organism movement and sexual reproduction

The second point mentions choosing a spot in the organism pool where offspring organisms are to be put. In the original Avida system, as per [2], newborn organisms are to be put in the neighborhood of the parent, in the spot occupied by the oldest neighbor. This effectively kills the neighbor and replaces it with a fresh organism. Using this replacement strategy in our system had an unexpected result, in that it effectively caused the opposite of evolution to happen.

Without fail it was observed that after a certain time each population using this replacement strategy resulted in a pool full of very short organisms which could only self-replicate. Our explanation of this anomaly would be the following: It did not pay off for the organisms to calculate Boolean functions. If they did, their metabolic rate would increase, and therefore in each scheduler run they would be allowed to execute a larger number of instructions, therefore also get old quicker, with each new instruction execution becoming more of a target to be killed off.

We have settled on a replacement strategy which balances age and fitness. The replacement strategy places the offspring organism in the spot in the one-hop neighborhood of the parent at which lies the neighbor with the highest age/rate ratio. This has provided us with stable well-behaved evolution, because now fit organisms are less likely to get killed off.

The last point is especially important for anyone looking to expand the system with new instructions. All complexity of instructions which involve several organisms interacting can and should be be relegated to the World. This can be done in a particularly easy manner allowed to us by the benefit of having the World implement the mediator design pattern. The following example on how we did that for movement may serve as a guide for future expansions:

1. Define a new instruction class InstructionMove
2. Modify the CPUEmulator's list of available instructions and the allowed integer range of instructions such that the new instruction is allowed to be a result of a random mutation.
3. In the execute() function of the InstructionMove class, simply notify the emulator's mediator of the "move" event
4. In the World class, define the function which reacts on this notification accordingly, in this case, by making the organism move in a random direction

Designing the World was admittedly a much more fun task than designing the Digital Organisms and the Instruction Set. Here we finally got to observe evolution, and through many iterations of changes to the system get it to perform well. The power of a well-chosen design pattern also made itself particularly clear here; With the world implementing the Mediator design pattern we were able to significantly reduce the strong coupling between the organisms and the world that they live in.

3.4 Experiment

Up to now we have concerned ourselves with the main details of the back-end of our system. To be able to use the system and manipulate the main parameters in an easy manner, the highest layer of abstraction, the class Experiment, was introduced.

Experiment came into existence from the natural need for the ease of use of the system. It works in the following manner: One defines the starting point of the experiment, the function one wants to evolve, and all of the important parameters of the system. These parameters include:

- The mutation probabilities
- The size of the organism pool
- Whether and how often cosmic ray mutations happen
- Which instruction set should be used ("default" or "custom")

One then lets the experiment run and either quickly grabs a coffee and comes back to observe the behavior, or goes for an ultramarathon run, because depending on the parameters the runtime of an experiment with the same start and endpoint can range from around half an hour to over ten hours.

The default starting point of the experiment is the default self-replicating organism from Avida, and the end function one wishes to evolve is per default bitwise equivalence. Both can however be changed, such that we may for example start with a self-replicating organism which can compute NAND and see which path evolution takes us along as we attempt to evolve XOR.

How an experiment exactly works is as follows:

1. All of the parameters of the experiment are defined and an Experiment object is instantiated. The experiment is then let to run.
2. Anytime an organism reports the wanted target function to the experiment, the organism is extensively tested to check whether it truly computes the function it reports it can compute, or whether this report was an anomaly associated with the aforementioned "cheating"
3. If such an organism which truly and reliably computes the target function is found, it is saved by the experiment for further analysis, and the experiment is stopped.

3.5 Mediator

The class Mediator is an abstract interface which defines how all of the different components of our system communicate with each other.

The Mediator is loosely based on the mediator design pattern, and defines a single "notify" function. The interface was introduced in order to reduce the initially very strong coupling between the different components of our system. All communication between instances of different classes happens using the "notify" function.

The problem we initially had, for which the mediator interface turned out to be an elegant solution, was the following:

Each time an organism divides, the world in which the parent organism lives in needs to instantiate a CPUEmulator in which the offspring genome is to be contained, and to place the resulting emulator into an appropriately chosen position in the pool. How we initially checked for division was: In the scheduler loop, upon every single executed instruction, we checked whether the instruction was "H-DIVIDE", the instruction which splits the parent organism's genome in two. If it was, the World runs the procedure for instantiating a new CPUEmulator. The procedure itself then involved the world accessing the CPUEmulator's memory and pulling out the resulting child genome.

Why this is problematic is that with this approach there obviously exists a large degree of coupling between the World and the CPUEmulator implementation. Any change to the structure of the memory of the emulators or to the name of the division instruction would also require changes to the implementation of the World.

This is just one scenario which requires communication between different components of the system. We can imagine that with a dozen or so different scenarios where communication is necessary, the entire system becomes quite messy, with a strong degree of codependence between different components.

With the introduction of the communication interface, the above procedure is transformed into the following: When a CPUEmulator executes the "H-DIVIDE" instruction, it notifies its associated mediator of this by executing the function:

notify(sender=self, event="division", result=offspring_genome)

The World in which the CPUEmulator lives in implements the Mediator interface, and as such defines a concrete implementation of the *notify* function. When notified of any event by any of the many CPUEmulator living in its pool, it runs the appropriate reaction procedure. In case of a division event, the World runs the *react_on_division* procedure, which instantiates a new CPUEmulator in the neighborhood of the emulator which just underwent division, with the genome as passed to it in the result argument of the *notify* function.

With all communication happening through such a communication interface, we are now free to change the implementation of the H-DIVIDE instruction and the structure of the CPUEmulator's memory without worrying about having to change anything in the World's implementation. This is just one of eleven such communication examples. It's safe to say that the introduction of the mediator interface saved us many headaches throughout the semester.

Here we show simplified sketches of two different communication scenarios in the system, along with the relevant class attributes and methods:

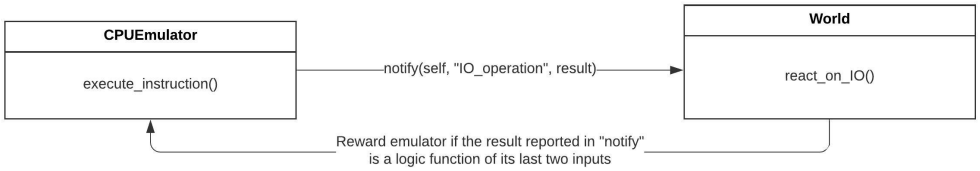

Figure 3.1: A world's reaction to a reported IO operation from an organism in its pool.

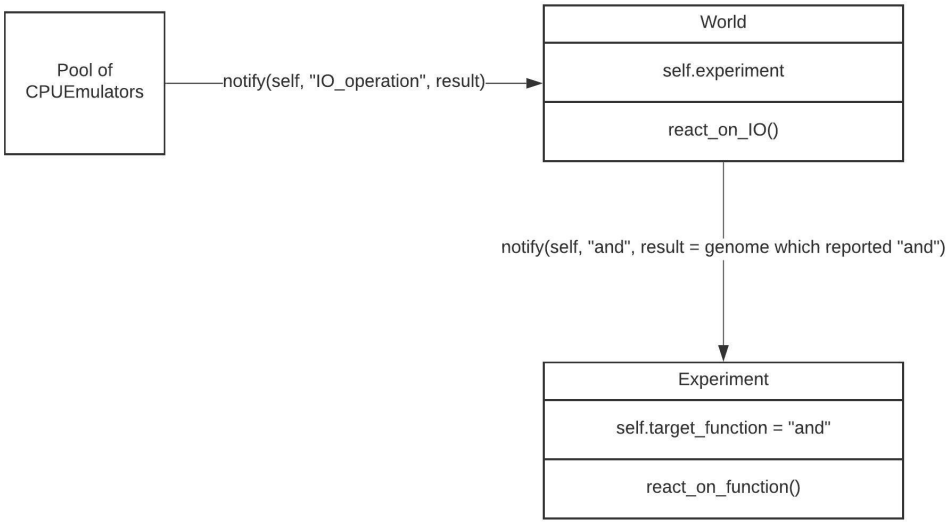

Figure 3.2: An example of an experiment with a target function "and". If an "and" is reported by an emulator in the pool, a chain of notifications is started which goes all the way up to the Experiment. The genome which reported "and" is then extensively tested by having it run a large number of cycles in five separate one-slot test worlds. The "and" is accepted if it reported a sufficient amount of times across the five test worlds.

Chapter 4

Experiments and Observations

Taking a similar approach to one taken in [1], we have tested our system on the evolution of the Boolean function NOR, starting from the default self-replicating organism which contains the self-replication skeleton and 35 "none" instructions. We have restricted our attention to evolving NOR due to time and hardware limitations.

All experiments were run on a 30×30 size pool. One update corresponds to each organism on average executing 30 instructions. By focusing on updates rather than run-times we can analyze performance without needing to take into account differences between different computer systems that a particular experiment was run on. Insertion and deletion probabilities were held fixed at 0.05 each.

Five experiments were run for each parameter configuration. The experiments are automatically stopped as soon as an organism capable of computing NOR is found, this organism being denoted as the "resulting organism".

Since experiments can potentially take up to ten hours, we were forced to restrict the focus of our attention to what we believed to be the main driving force of evolution, copy mutations. Apart from the effects of varying copy mutation probabilities, we have also investigated what kind of an effect the introduction of sexual reproduction and organism movement have. Further effects were investigated and presented, but not in the same methodical manner as the aforementioned two, due to time restrictions.

Some promising results were found, but we have to take the data with a grain of salt. Even though some trends are to be observed, as already explained the system is subject to a lot of randomness, which will immediately become clear from the variances of below results.

4.1 Experimental Results

4.1.1 The Default Instruction Set

Outlined here are experimental results with the default instruction set and several different copy mutation probabilities:

Copy Mutation Probability:	0.01	0.007	0.005
Mean Number of Updates:	**1001.4**	3788.5	4125
STD of the Number of Updates:	325.9	4829.1	2021.5
Minimum Number of Updates:	**499**	1075	2064
Maximum Number of Updates:	1470	**14525**	7312
Mean Resulting Organism Length:	**47.6**	65.0	64.4
STD of the Resulting Organism Length:	1.01	26.8	13.5

Table 4.1: Default Instruction Set, Varying Copy Mutation Probabilities. The focus of the table lies on the number of updates needed to evolve to NOR along with the length of the genome which computes it.

4.1.2 The Custom Instruction Set

Outlined here are experimental results with the custom instruction set, which is equivalent to the default one with the addition of movement and sexual reproduction, and variable copy mutation probabilities:

Copy Mutation Probability:	0.01	0.005
Mean Number of Updates:	4047.0	4293.0
STD of the Number of Updates:	1103.5	3342.3
Minimum Number of Updates:	2455	1376
Maximum Number of Updates:	5421	10840
Mean Resulting Organism Length:	90.8	73.2
STD of the Resulting Organism Length:	16.1	39.1

Table 4.2: Custom Instruction Set, Varying Copy Mutation Probabilities. The focus of the table lies on the number of updates needed to evolve to NOR along with the length of the genome which computes it.

4.1.3 Statistics of Evolved Populations

Here we showcase some statistics of different evolved populations in different experiments. All experiments were stopped as soon as one NOR organism was evolved. No XORs or EQUs were observed.

Logic Functions	Copy-mutation probability 0.005:	Copy-mutation probability 0.007:	Copy-mutation probability 0.009:
NOT:	489.2±169.34	402±270.32	479.4±245.92
NAND:	262.2±214.75	298.4±222.31	117±226.5
AND:	0±0	68.6±136.7	19.2±1325.76
OR_N:	385.6±175.03	325.8±203.27	163.8±153.59
OR:	340.4±205.8	216.8±206.95	339.8±163.65
AND_N:	3.4±6.8	71.4±137	38.4±42.25

Figure 4.1: Statistics over different evolved populations in various experiments. The numbers indicate the number of organisms capable of computing these functions in a world which supports 900 of them. Note how rarely AND appears in comparison to other functions; Something also observed in the original Avida system.

NOT:	528±121.8
NAND:	323±141.23
AND:	0±0
OR_N:	505.3±88.6
OR:	128.6±181.49
AND_N:	**351.66±251**

Figure 4.2: Statistics of a population evolved with a 0.15 insertion mutation probability, 0.005 copy mutation probability. Note that the population tends to evolve AND_N, something not often observed with a lower insertion mutation probability.

4.1.4 Examples of Evolutionary Paths

Here we showcase two examples of different evolutionary paths organisms evolved along in order to in the end achieve NOR.

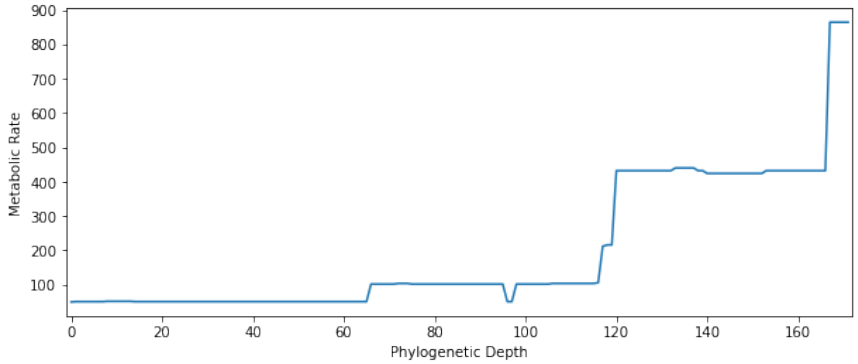

Figure 4.3: The lineage of an organism which reliably computed NOR after 172 generations. The path evolution took us along is the following one:
NOT → OR_N → NOT and OR_N → NOT and OR → NOR.
Particularly interesting here is the fact that the genome temporarily lost the ability to compute NOT, but then quickly regained it, after which it evolved smoothly towards NOR.

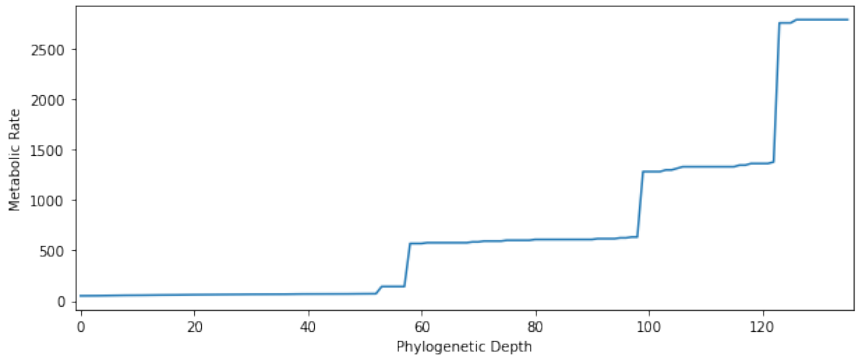

Figure 4.4: An example evolutionary path of an organism with high insertion probability (0.3) The very high rate is the result of a combination of complex instructions and a long resulting genome. The path taken here was:
NOT → OR → NOT and OR → OR_N and OR → NOT and NOR.

As a reminder, all experiments are stopped as soon as an organism which computes NOR is found. As such, we don't see the last step in the figures above.

It is interesting to observe how the functions combine together in different ways. In Figure 4.3 NOT and OR immediately combined into NOR. In Figure 4.4 NOT and OR evolved into OR_N and OR rather than directly into NOR.

4.2 Observations on the Experimental Results:

Before we go into discussing the results, we want to draw attention to the following:

- The mean number of updates is the best indicator of the expected run-time
- The mean resulting organism length is an indicator of expected algorithm complexity

We consider the best result to be the one which minimizes both. Our goal was never explicitly mentioned to be algorithm simplicity, but we believe that for obvious reasons it is a goal worth striving toward.

Such a result is immediately clear from Table 4.1. **Using a copy mutation probability of 0.01, with the default instruction set, we were able to consistently evolve to NOR in on average 1000 updates, with the minimum achieved number of updates of 499**. As a point of reference, 1000 updates take around 30 min on a 6th generation desktop Intel i5 processor. Evolution reliably happened with a very low variability of the number of updates.

With this particular choice of copy mutation probability, another interesting effect is to be observed: Even though we start with an organism length of 50, the average resulting organism length is shorter, at around 47.6.

At this point we would like to mention an interesting phenomenon we have observed: Even though insertion and deletion probabilities were kept at 0.05 each, after a large number of updates the organisms start **naturally growing**, surely incentivized by the fact that their metabolic rate is proportional to their genome length. It is therefore likely that the resulting average organism length of 47.6 is directly connected with the small number of updates needed to evolve to NOR from scratch. This hypothesis is further supported by the larger average resulting organism length in experiments with longer run-times, all other parameters having been kept equal.

Perhaps unsurprisingly, **introducing new custom-made instructions did not prove beneficial for system performance.** These new instructions highly increased the variability of results and had a tremendous negative impact on both of the two important points mentioned earlier, run-time and algorithm complexity. We have omitted experimental results with a 0.7% copy mutation probability, because with such high variability of results, even the differences between the two presented copy mutation probabilities are unclear. One possible reasoning for the negative impact of new instructions may be the following:

Without sexual reproduction it was observed that organisms often take specific paths over functions through long lines of evolution, with the end organism being the result of the combination of such functions. As an example, one path of evolution which was often observed on the path to *NOR* was: $NOT \rightarrow NOT$ and $OR\_N \rightarrow NOT$ and $OR \rightarrow NOR$. With the introduction of sexual reproduction we are disrupting this balance and are very likely producing offspring organisms which are incapable of any computation, even though both of its parents may well be viable well-behaved organisms capable of computing functions.

4.3 General Observations

Copy mutation probability can have a tremendous impact on runtime. This result seems natural as with a larger copy mutation probability we are introducing a larger amount of randomness into the system and as such are effectively exploring the possible space of algorithms more quickly. The positive effect of increasing the copy mutation probability is immediately obvious from Table 4.1. With a well-chosen copy mutation probability of 0.01, we were able to evolve to NOR within only 499 updates, and over the course of five experiments we have evolved to NOR after an average of 1001.4 updates.

Increasing insertion probability showed no clear positive impact on runtime, yet a clear negative impact on algorithm complexity. Five experiments were run with the same parameters as the ones that proved most promising (copy mutation probability 0.01), with the exemption of an increased insertion probability of 0.15 as opposed to the previous one of 0.05. The results are not substantial enough to merit their own table, and are briefly summarized here. The average number of updates was not substantially changed, being at around **1200**. The average resulting organism length was however, as expected, significantly increased, being at around **72**.

Cosmic Ray Mutations Rarely Drive Evolution. In ten experiments of default to NOR evolution with varying parameters, in which the presence of cosmic ray mutations in an organism's lineage was documented, only one organism capable of computing NOR had a cosmic ray mutation in its ancestry.

There can be large variability in the run-times of equivalently posed experiments. With one particular parameter choice we have tested evolving to NOR could take from 40 min to 8 h 30 min. Some trends can be observed, but it can be difficult to say which run-time is to be expected for a particular experiment.

There are several typical populations which can often be observed. Even though evolution is fully random, there are indicators that there are desirable paths which populations often take in search of complex functions. Boolean

AND does not necessarily often show up in a large group of organisms, however if it does, the population often ends up being one capable of computing all of the first four listed Boolean functions. Another fairly popular population path is one that goes over NOT, NAND, over to OR_N and OR, and finally combines these four in a way such that we obtain a stable population which can compute NOR and NOT.

XOR is very difficult to evolve to. This is the reason why we have restricted ourselves to experiments that only go up to NOR. This effect is also to be observed in the official Avida system. A stable population of organisms computing all functions except for XOR and EQU is easily achievable, however the step over to XOR seems to be an extremely difficult one. As a small exercise in computer engineering we have constructed custom organisms for computing all of the relevant Boolean functions, and indeed, the combination of instructions necessary to compute XOR is greatly more complex than any of the other ones, excluding EQU. This is no solid proof, but is an indicator of why we observe such behavior both in our system and in the official Avida system. If one however seeds the pool with custom-made organisms which can self-replicate and compute XOR, we have evolved to EQU from it multiple times in around 30 min, leading us to the conclusion that XOR is the main obstacle on the path towards the evolution towards EQU.

Final Remark:
The following analysis provides us with a **very rough estimate** of the benefit of using evolutionary search rather than randomly sampling algorithms from the space of all possible combinations of instructions. The organism evolved in Figure 4.3 was around fifty instructions long. If we exclude the self-replication skeleton, usually around twenty instructions long, we are left with around thirty coordinated instructions which when combined compute the Boolean NOR of two inputs. If we were to sample algorithms from the space of all possible algorithms over the default instruction set with twenty-six instructions, the probability of achieving this exact algorithm is around $(\frac{1}{26})^{30}$. Using evolutionary search, such an algorithm was evolved to through a lineage that is 172 generations long, a relatively short evolutionary path.

Of course, as noted, this serves as an extremely rough estimate. There are many possible combinations of instructions capable of computing Boolean NOR. Analyzing the probability of randomly sampling such an algorithm is beyond the scope of this paper.

Bibliography

[1] R. T. P. Richard E. Lenski, Charles Ofria and C. Adami, "The evolutionary origin of complex features," in *Nature 423*, May 2003.

[2] C. Ofria and C. O. Wilke, "Avida: A software platform for research in computational evolutionary biology," in *Artificial Life Models in Software*, Jul. 2009.