UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Investigation of the truly two-dimensional artificial life evolution

Author:
Oleksandr SYZONOV

Supervisor:
Oleg FARENYUK

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences
Faculty of Applied Sciences

Lviv 2021

Declaration of Authorship

I, Oleksandr SYZONOV, declare that this thesis titled, "Investigation of the truly two-dimensional artificial life evolution" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ______________________________

Date: ______________________________

"The purpose of life is finding the largest burden that you can bear and bearing it. "

Jordan Peterson

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Investigation of the truly two-dimensional artificial life evolution

by Oleksandr SYZONOV

Abstract

Artificial life is a field, where researchers try to create digital models of real, biological, life in order to understand evolution, life, and processes in life by reverse engineering those processes. Before this study, there were several successful attempts to simulate artificial life: Tierra (one-dimensional simulation), Avida (pseudo-two-dimensional circular space with ancestor), Amoeba (pseudo-two-dimensional space without ancestor), and Fungera (truly two-dimensional with ancestor). This study is focused on two main topics: investigating the behavior of Fungera and inventing new instruction sets for Fungera in order to make this simulation more stable and sustainable. In order to conduct an in-depth investigation, new metrics and analysis methods were introduced. For a more sustainable system, three additional instruction sets were introduced, inspired by DNA repair mechanisms and template-based control structures.

Acknowledgements

I am extremely grateful to Oleg Farenyuk for the best thesis I could choose to write, for being patient enough, for validating and extending my ideas and for providing directions to learn, write and think.

Also, my friends and family should be mentioned, for support, patience and fun time.

Contents

List of Figures

List of Abbreviations

CPU	**C**entral **P**rocessing **U**nit
DNA	**D**eoxyribo**n**ucleic **a**cid
NOP	**N**o **O**peration
RAM	**R**andom **A**ccess **M**emory

Dedicated to: my father and grandfather for being good examples, to my mother and another grandpa

Chapter 1

Introduction

1.1 Motivation

Understanding the evolutionary processes is very important to us as a humanity. Evolution is a spontaneous process that creates extremely complex, but orderly and antifragile systems. Firstly, we need to know what exactly caused nature to be the way it is and how complex organism systems interact with each other, evolve, and go forward. Why do some species go extinct and how does it influence other species? Why do species look the way they look and could they be as fit as current, have the same evolutionary roles, but look differently? Secondly, we have different gene interactions, genetic plasticity, and epigenetics, which makes our DNA code, which can be flexible enough to make us resilient and adaptable to a lot of environments and external conditions that we can encounter. It's also interesting to note that our DNA can change without breaking, given our external circumstances, for example, while producing antibodies. Thirdly, we need to understand evolution in order to predict the future of our biosphere. This could give us a hint on a better understanding of the species extinction and whether it's normal that some species go extinct and emergence in the process of speciation. Moreover even our struggle against viruses, pathogenic bacteria, and even cancer cells.

The most accurate way of understanding evolution in the real world is to actually observe it. This way, we have access to all possible data with the best accuracy we can get. The first problem with trying to understand evolution using observation is that it could potentially take many millions of years for new species to evolve and appear. For example, it took about 5.7 million years for people to evolve from the most recent common ancestor Hominini (dated 5 to 7 million years ago) of chimpanzees and humans to Homo sapiens dated about 300 000 years ago (Callaway, 2017). Of course, not every species evolved as slowly as humans, but the experiment time is still at least in thousands of years. We don't have thousands or millions of years to run an experiment, because of obvious reasons. Secondly, in the case of observation of evolution, we have only one experiment, but for a good understanding of some phenomena, we need to observe it at least multiple times across multiple contexts and situations, because many factors that contribute to this phenomenon are random. Evolution can produce organisms that are equally fit for equal conditions and ecosystem niches but are vastly different. Therefore, we need some comparative study of evolution. Both problems are addressed to some extent by direct experiments: twelve populations of Escherichia coli are studied for more than 30 years – over 60000 generations, revealing ecological diversification and showing interesting evolutionary dynamics (Good et al., 2017); large or even whole-genome sequencing allowed to study evolution process comparing related species (Soria-Carrasco et al., 2014, Tusso et al., 2020, Kapheim et al., 2015, Baker, Hanson-Smith, and Johnson, 2013), or populations of the same specie (Matute et al., 2010, Moyle and Nakazato, 2010); molecular biology experimental techniques advance provided insights of exact evolution ways (Tan et al., 2009, Werner et al., 2010, Christodoulou et al., 2010, Diss et al., 2017). But though those experiments being highly interesting and insightful, they are costly, time-consuming, and have inherent limitations. Given all this, a deep understanding of evolutionary processes would benefit from using additional approaches – parallels with other fields (Lieberman et al., 2007, Pagel, Atkinson, and Meade, 2007) and some kind of simulations (Adami, 1999).

Every simulation could not be 1-to-1, because of a variety of factors. Firstly, living things are overly complicated. For example, in addition to translation and transcription systems, we have a complicated DNA repair system, which consists of many mechanisms: base excision repair, nucleotide excision repair, mismatch repair, double-strand break repair, and many others (Chatterjee, 2017). These repair methods act on many levels using many different mechanisms, from simple duplication mechanisms to complete removal of damaged DNA parts and DNA regeneration. Secondly, we have many DNA damaging factors, ranging from simple replication errors in DNA to bulky lesions that cause a lot of damage, like UV radiation, Ionizing radiation, etc. These kinds of lesions can cause cancer if damaged cells are replicating (eg. stem cells) and aging if they are non-replicating (eg. brain cells) if repair mechanisms are not working properly. And this is just the only example of the complexity of the biological systems – they are extremely complicated on every abstraction level – molecular, cellular, organismic, population and ecosystem levels, and so on. Moreover, beyond those complexities, which are studied and understood to some extent, we have other, most random influential factor, which is nature by itself: we have Earth temperature falling and rising, asteroids falling, and ecosystems constantly changing because of many random factors that cannot be fully accounted for. But these factors are among the main drivers for speciation (Wagner, Kosnik, and Lidgard, 2006) – only the sympatric speciation is an exception (Wikipedia, 2021b). Therefore, simulation cannot fully reflect the whole variety of factors and better not event try – good simulation should capture important features of the system using just a minimal set of tools. It is interesting that some part of the biological complexity is unnecessary[1] (Fernandez and Lynch, 2011). With these ideas in mind, we need to formulate some criteria to choose factors that simulation should have.

1.2 Requirements for a good simulation

In order for something to be alive, we need to clearly define what life is. One of the simplest definitions of life is as follows (McKay, 2004): "*The simplest, but not the only, proof of life is to find something that is alive. There are only two properties that can determine if an object is alive: metabolism and motion. ... All living things require some level of metabolism to remain viable against entropy.*". Therefore, something is considered to be alive if it holds (at least partially) homeostasis, replicates, and somehow interacts with the environment.

We also need our system to be not only "alive", but also constantly evolving. Therefore, we need to define features that are inherent to evolution (Forbes, 2010): "*Evolution is defined as the change in the inherited traits of a population of organisms through successive generations. When living organisms reproduce, they pass on to their progeny a*

[1]Using some vaguely defined notion of adaptive changes which are needed.

collection of traits. ... When particular genetic sequences change in a population (e.g., via mutation) and these changes are inherited across successive generations, this is the stuff of evolution."

From this definition, we can extract several defining features of evolution: heritability, mutation, and natural selection, which drive evolutionary forces.

Therefore, a minimal list of requirements for a good simulation looks like this: it should have some mechanisms that do selection and organisms that live and reproduce with heritable traits, but are still mutable.

1.3 Conclusion

Understanding evolution is crucial for us because this way we can understand how complex systems evolve and future trends in the number of species. Observing evolution in nature is not enough, even though it's the most accurate way of researching evolution because it's too slow and hardly comparative. Therefore, we need some simulation of the real-life processes. They are very complex, so we cannot simulate them 1-to-1, because it's impossible to do this accurately and because it's very computation-heavy. So, we need a simplified simulation that has all traits of life and evolution: reproduction, mutations, natural selection, homeostasis, and action.

1.4 Requirements for a good simulation

Chapter 2

Related works

In this work, we focus on simulations capable of creating De novo features as contrasted to the evolution of systems with predefined phenotypic traits.

2.1 Tierra

The first successful simulation was created by Tom Ray and described in the paper "An Approach to Synthesis of life" (Ray, 1993a).

Every living organism utilizes energy in order to survive, reproduce and hold homeostasis. In the case of the Tierra simulation, the energy was simulated as a CPU time, memory was used as space. Instead of the DNA accompanied by transcription and translation system, Tierra organisms execute specially created machine instructions to replicate themselves directly, in a way similar to the hypothetical RNA self-replicators (Wikipedia, 2021a). RAM block is called a "soup" in parallel with the primordial soup idea (Wikipedia, 2021), which is populated by the machine (assembly) instructions.

The ancestor has 3 main parts: self-examination, replication loop, and copy operations. In order to find its size, it firstly finds its beginning coordinate, marked by a special pattern, and puts it to the AX register, then it finds the ancestor end and puts it to the BX register. After that, it finds organism size by subtracting the beginning and the end. Then, it allocates memory for a child and copies all of his commands by iterating through itself in a replication loop.

Then, we have a self-replicating assembly code that can quickly populate the soup. But in order for evolution to occur, there need to exist some mechanisms to mutate genomes.

In Tierra, there are several ways of mutation (Ray, 1993a, Ray, 1991):

1. Some bits are randomly chosen from the soup and flipped
2. While copying bits, they are flipped with some probabilities
3. During command execution some commands have erroneous execution.

This way, similarly to real evolution, everything can go wrong, mutate and change: external factors can cause mutations, replication errors sometimes occur and even gene expression is sometimes flawed.

To mimic competition for the resources, every organism is put into an execution queue, which determines the order in which commands will be executed. Each creature has some amount of CPU time, depending on its genome size. There is a parameter of slicer power. If it's 1, the CPU time is distributed with equal probability, if it's less than 1, small creatures get more time. Also, new organisms are placed at the end of this queue, so child organisms get CPU time only after the parent organisms.

Because all the creatures are constantly reproducing, at some point in time they fill the finite memory entirely, stagnating further evolution process. In order to solve this problem, the reaper is introduced. The reaper is a priority queue. When an organism is created, it's placed at the end of this queue. When a time comes to kill, the killer always chooses a creature at the top of it for killing. During the execution of the instructions, when errors occur, each execution error moves the organism higher in the queue in order to ensure the extinction of flawed creatures. This way flawed organisms are dying, but also older organisms die faster than young ones because new ones are placed at the end of this queue.

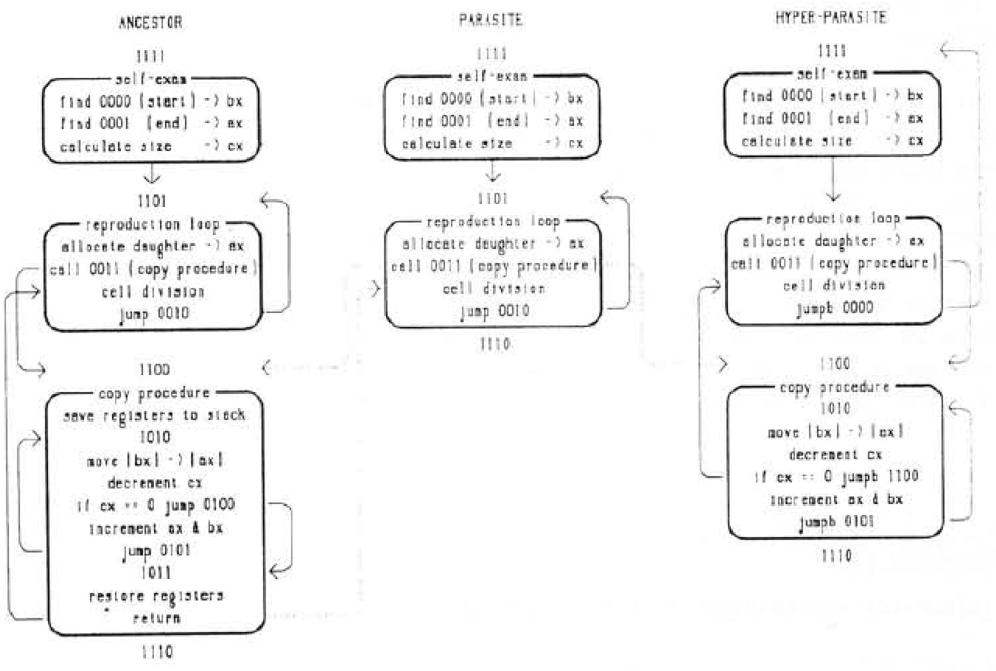


FIGURE 2.1: Examples of organisms, (Ray, 1993a)

When this simulation was run, the soup became quickly populated with creatures. Then, the reaper mechanism kicked in. Organisms became very short-lived. Many of them were killed quickly. But then, new more resilient genotypes appeared. With this, the diversity of the whole population also increased. Through some time, the evolutionary processes created some interesting kinds of organisms that were very different from the original self-reproducing ancestor (Ray, 1993a, Ray, 1993b):

1. Parasites

   Because of mutation in one command, that was needed to denote the end of the organism during the self-examination, the organism calculates its size wrongly. This leads to the execution of the code of other organisms.

2. Hyper-parasites

   Hyper-parasites are organisms that exploit parasites for reproduction. When the instruction pointer of the parasite passes through hyperparasite code, it sets registers with organism location and size with hyperparasite's location and size so that parasite copies another organism's instructions. After copy it jumps and not returns, thereby rendering the parasite unable to copy itself.

3. And many others (creatures immune to parasites, social hyper-parasites, hyper-hyper parasites, and so on).

Those behaviors were not coded into initial organisms but occurred spontaneously due to the mutations. It is also interesting that after some time, normal ancestor organisms evolved to become immune to the parasites and parasites evolved to bypass protection mechanisms. So, the "arms race" continued.

2.2 Avida

Avida is a simulation, partially inspired by the Tierra, but with other goals in mind and many significant differences.

Firstly, instead of a one-dimensional environment, it has a pseudo-two-dimensional with torus topology (Adami and Brown, 1994): *"In Avida, the physical position of a string is determined by its coordinates in a $N \times M$ grid with the topology of a torus."*.

Secondly, its instruction set has some similarities to the x86 instruction set in comparison with the Tierra (Adami, 1999, Adami and al., 2015). For example, it contains: no-operation (nop-a, nop-b, nop-c, and nop-x, which is the only pure nop), control instruction (if-not-0, in-n-equ), jumps (jump forward or backward to the complementary pattern), subroutine calls, mathematical instructions (bit shifts, addition, subtraction, not-and, and not-or), allocation and division of a child, writes, reads, I/O (read from the input buffer and write to output buffer), stack switching, code injection, and many others. Also, organism genomes are circular. This way, they start from the beginning when reached the end.

Just like in the Tierra, we have an environment seeded with a self-replicating ancestor. Its replication process has these steps:

1. Allocate new child memory.
2. Execute self-copying loop to write into the child.
3. Then, when copying is fully done, call the division command.

The place, where to put the offspring is determined by the simulation. Each cell can be filled with one creature at a time. Each parent can produce offspring only into an adjacent cell. If adjacent cells are filled with other organisms, one of them is replaced with the newborn during child division. There are several mechanisms for choosing, which cell to replace with the new organism: choosing the empty cell, choosing one completely randomly, choosing to "kill" the oldest individual in the neighborhood, and choosing the least fit organism in the neighborhood.

Fitness is determined by the merit mechanism. In Tierra, there's no fitness measure and every organism that executes its code with no error is equally fit, but in Avida metric called merit was introduced. Merit is determined by what the program actually does. For example, at the first stage, all programs that do at least some I/O, are rewarded with greater merit. At the second stage, all programs that do correct I/O are rewarded. And then, programs that take one number from the input buffer and output them into the output buffer are rewarded. Merit is purely based on phenotype (program behavior), rather than on genotype (exact command sequences). Fitness is calculated as follows:

$$fitness = \frac{merit}{gestation\_time}$$

Gestation time is a time needed in order to reproduce. Additionally, fitness can also influence time slice allocation order.

With fitness-based selection mechanisms, simulation can breed organisms by organizing the selection of programs that can do what is demanded by external (to the simulated world) problems.

Given that all interactions are purely local (the organism can interact only with the adjacent cells), the simulation can be run asynchronously, because information propagation is very limited.

Lastly, the Avida CPU model is a bit more complicated. When a genome has its CPU time, the initial CPU state is set. Genomes are loaded into 1D memory, where each memory cell is filled with some instruction and has several flags that indicate whether this cell was executed before, mutated or edited. The CPU has three registers: AX, BX, and CX that store some 32-bit value, which is some random bit combination unless set to some other value. A CPU also has input-output buffers that can be read from and written to using get and put instructions. CPU also has 2 stacks where data can be stored and switched between using switch_stack instruction.

Authors found that using local propagation mechanisms makes extinction events less severe and deadly and new information had more time to be used in new genotypes and produce more diverse populations (Adami and Brown, 1994).

On the other hand, because of the Avida memory protection, substantially novel organism, such as parasites are impossible in the Avida.

2.3 Amoeba

Another important artificial life simulation is Amoeba (Pargellis, 2001).

Basically, it does not use human-created ancestors, so all organisms are required to be born from "soup". This is one of the main differences between Amoeba, Avida, and Tierra. To enable such behavior, after each global reaping cycle, a generator of random sequences introduces about 50 random sequences that are basically 5-30 randomly chosen instructions and codon labels.

Then cells try to reproduce and compete for the resources. Each cell is allowed to execute about 30 operations at a time by the time slicer. Very rarely a cell is a self-replicating one – the probability of this is very low: $P = N^{-l}$, where $N = 32$ is a total number of instructions in the instruction set and l is a length of the sequence.

In Amoeba, there's a reaper mechanism that combines features of reapers from the Tierra and the Avida. There exist both local and global reapers. Global reaper reaps about 25% of all organisms when there is no virtual CPUs left. Local reaper kicks in when a parent organism replicates and there is no adjacent place for a child. Then, it randomly kills an adjacent organism in order to free a cell. Also, reaper does not favor any phenotype and the only goal is to reproduce.

Even though random organisms are introduced every generation, there are mutation mechanisms in this simulation too. Its use is limited, compared to other simulations, because it occurs only on copy to a child. With some probability, a child organism will have at least one mutation. There are 3 types of mutations: command deletion, command insertion, and command substitution for a randomly chosen one.

Another difference is that Amoeba has a different addressing system: each command has a randomly chosen label from about 64 possible ones. This assignment is stored in an assignment matrix. These codons can be used for conditional jumps or loading address registers.

As a result (Greenbaum and Pargellis, 2016) in the simulation which started from the prebiotic soup, initially, some pre-replicators are born. They are inefficient and usually reproduce slowly, sometimes copying only partially. Because successful and robust replicators write only to adjacent cells, colonies of organisms are formed. Sometimes, though – very rarely (Pargellis, 2001, parasites, called "viruses" by the authors, are emerging too.

On the other side, Amoeba mostly fails to support diverse ecosystems – most of the time the only one species dominates.

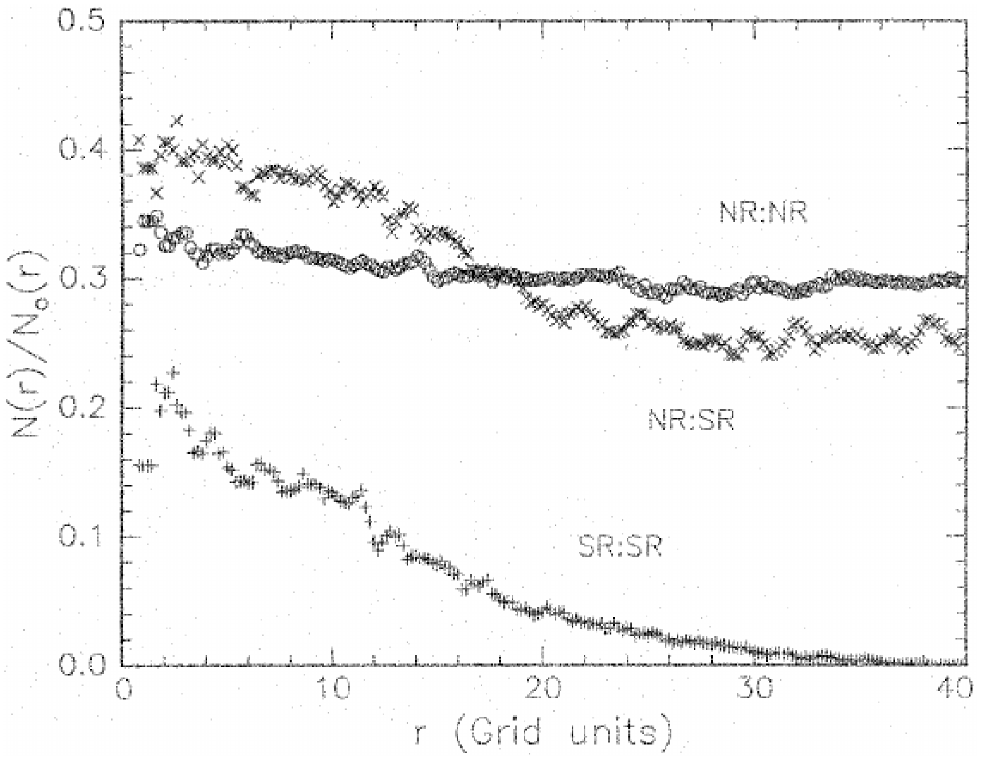


FIGURE 2.2: Self-Replicators/Non-Replicators in the Amoeba simulation (Pargellis, 2001).

2.4 Fungera

The main idea of the Fungera simulation (Poliakov, 2020) is to attempt to improve the complexity of the artificial life evolution by providing a truly two-dimensional environment. In all of the mentioned above works organisms were one-dimensional even if the memory had two dimensions. It limits the complexity of the possible artificial metabolism patterns and organism interactions. The Tierra is one-dimensional, the Avida and the Amoeba can be named pseudo-two-dimensional. There were limited experiments on two-dimensional artificial organisms evolution which showed that they are too fragile and always go extinct (De Dinechin, 1997). But this conclusion looks like excessive generalization (Poliakov, 2020, p. 25).

The initial Fungera instruction set is based on a programming language named Befunge. Befunge is a two-dimensional stack-based language, rather low-level and similar to an assembly language. Two-dimensional means that it can execute code from not only down (with jumps) but also in the up, left, or right direction and can change direction during the execution. It can be classified as an esoteric language (Wikipedia, 2021. Befunge has only basic math operations: addition, subtraction, and multiplication; stack manipulation operations: pushing, popping, and swapping stack values; direction modifiers; memory modification instructions. It's also worth mentioning that Befunge programs can be self-modifying, therefore it is rather suitable to use Befunge for mutating self-replicating organisms.

example of program written in Befunge can be seen on the Fig. 2.3.

```
2>:3g" "-!v\  g30          <
|!`"0":+1_:.:03p>03g+:"0"`|
@               ^  p3\" ":<
2 234567890123456789012345678901234567890123456789012345678901234567890123456789
```

FIGURE 2.3: Example of Befunge program(Sieve of Eratorsphenes in Befunge)

In Tierra, Avida, and Amoeba instruction sets of the virtual CPUs are rather similar to the classical real-world CPUs. But in order to properly implement two-dimensional memory and organisms, some modifications to the virtual CPU architecture were necessary: stack items, general-purpose registers (RA, RB, RC, RD), and instruction pointer are two-dimensional and have two components, corresponding to x and y components.

The initial instruction set of the Fungera is presented in Table 2.1.

Befunge instructions were extended with some useful commands for the artificial organisms. One of them was template matching instruction. An example of its usage can look like this:

```
>\&d:..
```

This command seeks the complementary pattern to ":.." which would be ".::" and puts in into the register RD. Here ":" and "." are NOP-codes intended to be used as a pattern. Code ":" is considered complementary to "." and vice versa. This command can be used in the organism code for detecting its size as a part of replication.

Code	Sym	Ops	Description	Type
[0, 0]	.	0	Template constructor	Template
[0, 1]	:	0	Template constructor	Template
[1, 0]	a	0	Register modifier	Register
[1, 1]	b	0	Register modifier	Register
[1, 2]	c	0	Register modifier	Register
[1, 3]	d	0	Register modifier	Register
[2, 0]	^	0	Direction modifier (up)	Direction
[2, 1]	v	0	Direction modifier (down)	Direction
[2, 2]	>	0	Direction modifier (right)	Direction
[2, 3]	<	0	Direction modifier (left)	Direction
[3, 0]	x	0	Operation modifier	Operation
[3, 1]	y	0	Operation modifier	Operation
[4, 0]	&	2+	Find template, put its address in register	Matching
[5, 0]	?	4	If not zero	Conditional
[6, 0]	0	1	Put [0, 0] vector into the register	Arithmetic
[6, 1]	1	1	Put [1, 1] vector into the register	Arithmetic
[6, 2]	-	2	Decrements value in register	Arithmetic
[6, 3]	+	2	Increment value in register	Arithmetic
[6, 4]	~	3	Subtract registers and store result in register	Arithmetic
[6, 5]	*	3	Add registers and store result in register	Arithmetic
[7, 0]	W	2	Write instruction from register to address	Replication
[7, 1]	L	2	Load instruction from address to register	Replication
[7, 2]	@	2	Allocate child memory of size	Replication
[7, 3]	$	0	Split child organism	Replication
[8, 0]	S	1	Push value from register into the stack	Stack
[8, 1]	P	1	Pop value of register into the stack	Stack

TABLE 2.1: Initial Fungera instruction set, (Poliakov, 2020, p. 12)

Other important two important but unusual for the ordinary instruction sets are: ″allocate child″ (″@″) and ″split child″ (″$″). They are used for reproduction. Instruction ″allocate child″ takes child size from the specified register and then finds a chunk of memory of this size in a direction of execution and puts its coordinates into another specified register. At a moment of time, an organism can write only into one allocated child and when it has finished, it calls the ″split child″ command, which creates a new child organism that has its own CPU with its own instruction pointer, registers, and stack.

Like all of the simulations described above, Fungera has reaper mechanisms. Like in Tierra, there is a reaper queue. Each organism on producing error moves up that queue and newborn organisms are appended to the end of it. Each time memory is filled for about 90%, the reaper kills 50% of the creatures on top of the queue. Also, organisms are killed when their number of errors is bigger than some threshold and when they are not reproducing for some period of time.

Mutation mechanisms for Fungera are fairly simple. It has a parameter, say n, that determines that each nth iterations some cell will be chosen and instruction in this cell will be randomly replaced with some other instruction from the instruction set. This way, mutations can occur both in some unallocated memory cells and in live organisms.

Ancestor in Fungera is similar in principle to the Tierra one but is much more complex in terms of the actual implementation. Firstly, by seeking patterns in its own code and putting their coordinates into registers, it determines its size. When it has found its own size, it determines the direction for child allocation by using the stack to "remember" the previous direction of allocation. Then, it allocates a child of the same size the organism is. After that it starts a double replication loop – for each column and row, using increments and decrements, which copies by loading instruction code from a specified memory cell into one of the registers and then writing this command into a specified child cell. At last new organism is created by the "split child" instruction.

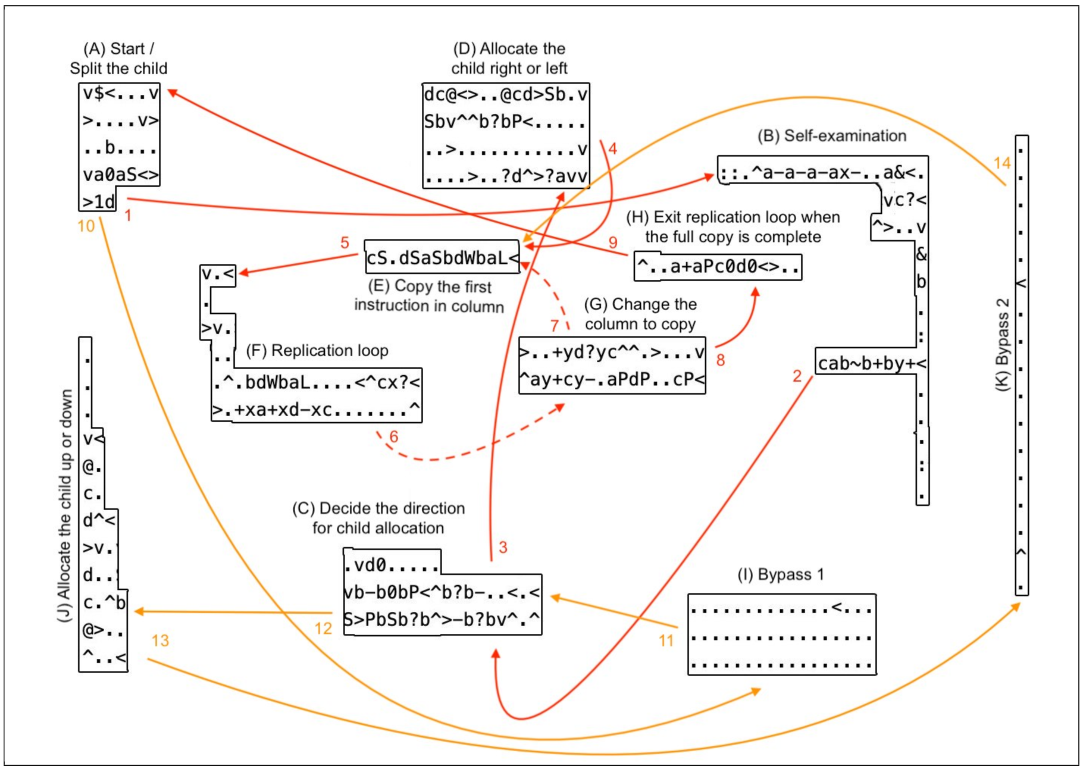
As a result of running this simulation, it was determined that because of mutations, some aberrant organisms occur that are small and non-functional. They were called microvesicles. They occurred because of the reaper setup, which wasn't killing them fast enough. Then, microvesicles even produced some other aberrant microvesicles.

Code of the initial Fungera ancestor and it's layout are presented in Figures 2.4 and 2.5.

```
v$<...vdc@<>..@cd>Sb.v.
>....v>Sbv^^b?bP<......
..b......>...........v.
va0aS<>....>..?d^>?avv.
>1d::.^a-a-a-ax-..a&<..
.v.<cS.dSaSbdWbaL<vc?<<
..^..a+aPc0d0<>..^>..v.
.>v.>..+yd?yc^^.>...v&.
v<..^ay+cy-.aPdP..cP<b.
@..^.bdWbaL....<^cx?<..
c.>.+xa+xd-xc.......^:.
d^<.vd0.....cab~b+bc+<.
>v.vb-b0bP<^b?b-..<.<..
d..S>PbSb?b^>-b?bv^.^..
c.^b.............<.....
@>...................:^
^..<...................
```

FIGURE 2.4: Initial Fungera ancestor, (Poliakov, 2020).

This study had a really interesting idea, but a lot of space for different improvements. Simulation behavior was only studied with one parameter set for reaper, mutations, and much other stuff. There was only one ancestor tried and only one command set. Also, the simulation could be made using a much faster implementation in other languages than Python.



```
(A)
v$<...v
>....v>
..b....
va0aS<>
>1d
```

```
(D)
dc@<>..@cd>Sb.v
Sbv^^b?bP<.....
..>...........v
....>..?d^>?avv
```

```
(B)
::.^a-a-a-ax-..a&<.
              vc?<
             ^>..v
                 &
                 b
                 .
                 :
         cab~b+by+<
                 .
                 .
                 .
                 :
                 .
```

```
(H)
^..a+aPc0d0<>..
```

```
(E)
cS.dSaSbdWbaL<
```

```
(F)
v.<
.
>v.
 ..
 .^.bdWbaL....<^cx?<
 >.+xa+xd-xc.......^
```

```
(G)
>..+yd?yc^^.>...v
^ay+cy-.aPdP..cP<
```

```
(C)
.vd0.....
vb-b0bP<^b?b-..<.<
S>PbSb?b^>-b?bv^.^
```

```
(I)
.............<...
.................
.................
```

```
(J)
.
.
.
v<
@.
c.
d^<
>v.
d..
c.^b
@>..
^..<
```

```
(K)
.
.
.
.
.
<
.
.
.
.
.
.
.
.
.
^
.
```

FIGURE 2.5: Algorithm of the initial Fungera ancestor, (Poliakov, 2020).

Chapter 3

Extending Fungera instruction set

To counteract 2D organisms fragility several instruction sets were developed in this work addressing possible fragility by using two approaches – error correction (inspired by DNA reparation and embryonic development equifinality) and more active usage of pattern-based addressing.

3.1 Instruction set with error correction – Fungera-2/Cellgera

Given that the initial ancestor organism was pretty fragile, partially because of the rigidity and fragility of the instruction set, it was decided to make the ancestor organism more resilient to the environment. The main cause of the malfunction of commands was a mutation. Almost any change of instructions can break the organism.

In the DNA, mutations, errors, and fractions occur all the time. But in the real world DNA has many repair mechanisms. For example, DNA has two complementary strands, so if one of them is broken or mutated, it uses a complementary one. Also, genetic code has some redundancy. So, the following principle of making redundant information for error correction was implemented.

Each command has its own ″block″, that consists of the main instruction repeated 4 times in the corners and error-repair entry commands.

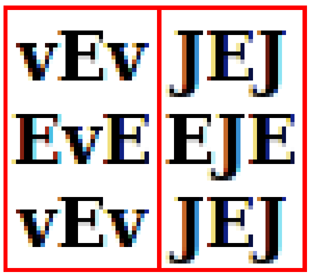


FIGURE 3.1: Block examples (blocks are outlined in red).

This way, the total area of the same organism increases 9 times, because each command is now in an error-correction block. When we enter a block, the voting mechanism kicks in, and among 5 commands the most numerous is chosen and then each corner and center of this block is set to the most numerous command.

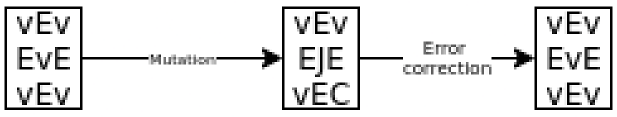


FIGURE 3.2: Error correction example.

This way, to change executed commands, there needs to change the majority of the commands to some other command. Therefore, each cell is much more mutation-resilient than a standalone cell.

After the execution of one cell, the instruction pointer jumps to the next cell, and error correction kicks in. Therefore, the number of allocated memory cells triples along each axis. Also, the copying mechanism was changed, because now we operate in terms of blocks, and blocks are copied by copying their center 5 times and inserting entry E(error correction) instructions by the sides.

Any ancestor from the basic Fungera can be compiled into a new instruction set because each cell is trivially constructed from the base instruction.

When creating an organism, the instruction pointer is placed not in the upper left corner, but in the center of the upper left block to avoid wrongful execution.

3.2 Instruction set with direction-dependent jumps and calls – Fungera-3

Another instruction set that was implemented is an instruction set with a direction-dependent jump and call commands – more Tierra-like.

The jump instruction works as follows. We have ”J” instruction, after which we have some patterns. The pattern consists from the sequence of the ”:” and ”.” NOP instructions which is finished by any not-”.” and non-”:” instruction. When executed, ”J” instruction seeks a complementary pattern (for ”:” complementary is ”.” and vice versa) in the direction of the instruction pointer. When the CPU finds it, it unconditionally jumps to the beginning of this pattern, bypassing everything in between.

Example of such jump:

> J :. abababababbababba .:

Direction-dependent call works in a similar fashion: CPU finds a complementary pattern and jumps to the beginning of it, but before the jumping previous position is saved to the stack. The opposite command, ”R”, ”return”, takes value from the top of the stack and jumps to it.

Example of code with directed call:

> C :. >C:.v.v.<... .: PaSa-ya-ya-ya R

3.3 Instruction set with direction-independent jumps and calls – Fungera-4

This instruction set is fairly similar to the previous one but the exact mechanism of finding patterns is different.

Firstly, in this set patterns are fixed-length and can consist of any instructions. For example, in this case, the pattern is "C:."

```
> C C:.
```

Secondly, we search for the pattern not only in the instruction pointer direction but also in every other direction within some pre-configured radius. Then, when patterns are found, we choose ones that are inside the organism first and find the closest one. In case there are no patterns inside the calling organism (mainly as a result of some mutation), we choose the closest pattern outside the organism.

3.4 Ancestor organisms for instruction sets Fungera-2, 3, 4

3.4.1 Instruction set with error correction (Fungera-2)

The only differences between this instruction set and the base instruction set are block structure and organism size. Therefore, a simple compiler from Fungera to this instruction set was implemented. The compiled Fungera ancestor is shown in Figure 3.3.

3.4.2 Instruction set with direction-dependent jumps and calls (Fungera-3)

Source code of the Fungera-3 ancestor is presented in Figure 3.4.

Its algorithm is simple:

1. In the first line, we make a call to get coordinates of the organism origin (top left corner coordinates) into the stack.
2. We pop them from the stack into register AX and decrement it 3 times.
3. After that, we have coordinates of the beginning of the organism in the register RA.
4. Then, we go to the opposite side of the organism and get coordinates of the end (right bottom corner) into RB.
5. Then, we subtract them and put the result – the size of the ancestor, into the RC.
6. After that, we allocate child of size from RC and put its beginning into the RD.
7. Then, we enter the reproduction loop for size and copy instructions by loading them into RD using coordinates from the RB register and writing them to the coordinates in the RA register.
8. When the reproduction loop is over and the ancestor is fully copied, we execute split child instruction and start all over.

This flow of the execution is visualized on the Figure 3.5, parts executing different stages of the algorithm are colored on the Figure 3.6.

3.4.3 Instruction set with direction-independent jumps and calls

Ancestor for this instruction set is similar to the previous one:

1. Firstly, we call to the right bottom corner.
2. Then, we call from there and pop the left top corner and right bottom corner into registers.
3. After that, we find organism size by subtracting them.
4. Then, we repeat steps 6-8 from the previous ancestor.

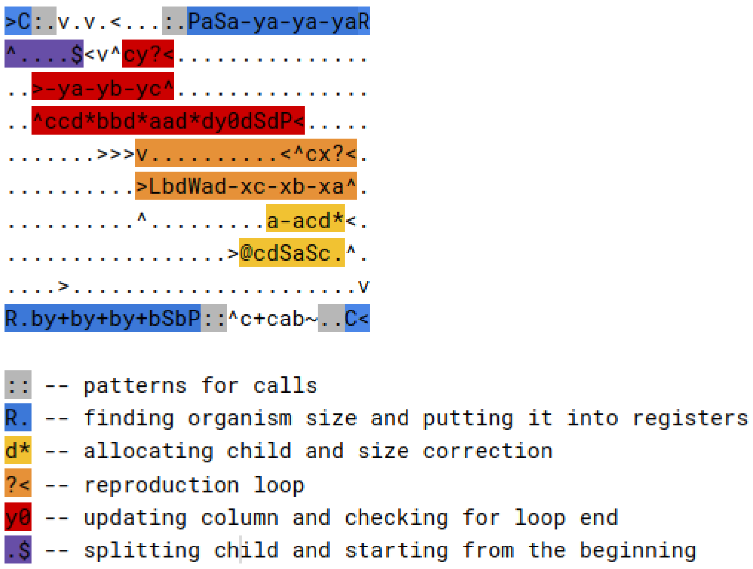
Source code of the Fungera-4 ancestor is presented on the Figure 3.7, it's flow of execution – on the Figure 3.9 and blocks are marked on the Figure 3.8.

```
vEv$E$<E<.E..E..E.vEvdEdcEc@E@<E<>E>.E..E.@E@cEcdEd>E>SESbEb.E.vEv.E.
EvEE$EE<EE.EE.EE.EEvEEdEEcEE@EE<EE>EE.EE.EE@EEcEEdEE>EESEEbEE.EEvEE.E
vEv$E$<E<.E..E..E.vEvdEdcEc@E@<E<>E>.E..E.@E@cEcdEd>E>SESbEb.E.vEv.E.
>E>.E..E..E..E.vEv>E>SESbEbvEv^E^^E^bEb?E?bEbPEP<E<.E..E..E..E..E..E.
E>EE.EE.EE.EE.EEvEE>EESEEbEEvEE^EE^EEbEE?EEbEEPEE<EE.EE.EE.EE.EE.EE.E
>E>.E..E..E..E.vEv>E>SESbEbvEv^E^^E^bEb?E?bEbPEP<E<.E..E..E..E..E..E.
.E..E.bEb.E..E..E..E..E..E.>E>.E..E..E..E..E..E..E..E..E..E..E.vEv.E.
E.EE.EEbEE.EE.EE.EE.EE.EE.EE>EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EEvEE.E
.E..E.bEb.E..E..E..E..E..E.>E>.E..E..E..E..E..E..E..E..E..E..E.vEv.E.
vEvaEa0E0aEaSES<E<>E>.E..E..E..E.>E>.E..E.?E?dEd^E^>E>?E?aEavEvvEv.E.
EvEEaEE0EEaEESEE<EE>EE.EE.EE.EE.EE>EE.EE.EE?EEdEE^EE>EE?EEaEEvEEvEE.E
vEvaEa0E0aEaSES<E<>E>.E..E..E..E.>E>.E..E.?E?dEd^E^>E>?E?aEavEvvEv.E.
>E>1E1dEd:E::E:.E.^E^aEa-E-aEa-E-aEa-E-aEaxEx-E-.E..E.aEa&E&<E<.E..E.
E>EE1EEdEE:EE:EE.EE^EEaEE-EEaEE-EEaEE-EEaEExEE-EE.EE.EEaEE&EE<EE.EE.E
>E>1E1dEd:E::E:.E.^E^aEa-E-aEa-E-aEa-E-aEaxEx-E-.E..E.aEa&E&<E<.E..E.
.E.vEv.E.<E<cEcSES.E.dEdSESaEaSESbEbdEdWEWbEbaEaLEL<E<vEvcEc?E?<E<<E<
E.EEvEE.EE<EEcEESEE.EEdEESEEaEESEEbEEdEEWEEbEEaEELEE<EEvEEcEE?EE<EE<E
.E.vEv.E.<E<cEcSES.E.dEdSESaEaSESbEbdEdWEWbEbaEaLEL<E<vEvcEc?E?<E<<E<
.E..E.^E^.E..E.aEa+E+aEaPEPcEc0E0dEd0E0<E<>E>.E..E.^E^>E>.E..E.vEv.E.
E.EE.EE^EE.EE.EEaEE+EEaEEPEEcEE0EEdEE0EE<EE>EE.EE.EE^EE>EE.EE.EEvEE.E
.E..E.^E^.E..E.aEa+E+aEaPEPcEc0E0dEd0E0<E<>E>.E..E.^E^>E>.E..E.vEv.E.
.E.>E>vEv.E.>E>.E..E.+E+yEydEd?E?yEycEc^E^^E^.E.>E>.E..E..E.vEv&E&.E.
E.EE>EEvEE.EE>EE.EE.EE+EEyEEdEE?EEyEEcEE^EE^EE.EE>EE.EE.EE.EEvEE&EE.E
.E.>E>vEv.E.>E>.E..E.+E+yEydEd?E?yEycEc^E^^E^.E.>E>.E..E..E.vEv&E&.E.
vEv<E<.E..E.^E^aEayEy+E+cEcyEy-E-.E.aEaPEPdEdPEP.E..E.cEcPEP<E<bEb.E.
EvEE<EE.EE.EE^EEaEEyEE+EEcEEyEE-EE.EEaEEPEEdEEPEE.EE.EEcEEPEE<EEbEE.E
vEv<E<.E..E.^E^aEayEy+E+cEcyEy-E-.E.aEaPEPdEdPEP.E..E.cEcPEP<E<bEb.E.
@E@.E..E.^E^.E.bEbdEdWEWbEbaEaLEL.E..E..E..E.<E<^E^cEcxEx?E?<E<.E..E.
E@EE.EE.EE^EE.EEbEEdEEWEEbEEaEELEE.EE.EE.EE.EE<EE^EEcEExEE?EE<EE.EE.E
@E@.E..E.^E^.E.bEbdEdWEWbEbaEaLEL.E..E..E..E.<E<^E^cEcxEx?E?<E<.E..E.
cEc.E.>E>.E.+E+xExaEa+E+xExdEd-E-xExcEc.E..E..E..E..E..E..E.^E^:E:.E.
EcEE.EE>EE.EE+EExEEaEE+EExEEdEE-EExEEcEE.EE.EE.EE.EE.EE.EE.EE^EE:EE.E
cEc.E.>E>.E.+E+xExaEa+E+xExdEd-E-xExcEc.E..E..E..E..E..E..E.^E^:E:.E.
dEd^E^<E<.E.vEvdEd0E0.E..E..E..E..E.cEcaEabEb~E~bEb+E+bEbyEy+E+<E<.E.
EdEE^EE<EE.EEvEEdEE0EE.EE.EE.EE.EE.EEcEEaEEbEE~EEbEE+EEbEEyEE+EE<EE.E
dEd^E^<E<.E.vEvdEd0E0.E..E..E..E..E.cEcaEabEb~E~bEb+E+bEbyEy+E+<E<.E.
>E>vEv.E.vEvbEb-E-bEb0E0bEbPEP<E<^E^bEb?E?bEb-E-.E..E.<E<.E.<E<.E..E.
E>EEvEE.EEvEEbEE-EEbEE0EEbEEPEE<EE^EEbEE?EEbEE-EE.EE.EE<EE.EE<EE.EE.E
>E>vEv.E.vEvbEb-E-bEb0E0bEbPEP<E<^E^bEb?E?bEb-E-.E..E.<E<.E.<E<.E..E.
dEd.E..E.SES>E>PEPbEbSESbEb?E?bEb^E^>E>-E-bEb?E?bEbvEv^E^.E.^E^.E..E.
EdEE.EE.EESEE>EEPEEbEESEEbEE?EEbEE^EE>EE-EEbEE?EEbEEvEE^EE.EE^EE.EE.E
dEd.E..E.SES>E>PEPbEbSESbEb?E?bEb^E^>E>-E-bEb?E?bEbvEv^E^.E.^E^.E..E.
cEc.E.^E^bEb.E..E..E..E..E..E..E..E..E..E..E..E..E.<E<.E..E..E..E..E.
EcEE.EE^EEbEE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE<EE.EE.EE.EE.EE.E
cEc.E.^E^bEb.E..E..E..E..E..E..E..E..E..E..E..E..E.<E<.E..E..E..E..E.
@E@>E>.E..E..E..E..E..E..E..E..E..E..E..E..E..E..E..E..E..E..E.:E:^E^
E@EE>EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE:EE^E
@E@>E>.E..E..E..E..E..E..E..E..E..E..E..E..E..E..E..E..E..E..E.:E:^E^
^E^.E..E.<E<.E..E..E..E..E..E..E..E..E..E..E..E..E..E..E..E..E..E..E.
E^EE.EE.EE<EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.EE.E
^E^.E..E.<E<.E..E..E..E..E..E..E..E..E..E..E..E..E..E..E..E..E..E..E.
```

FIGURE 3.3: Fungera-2/Cellgera ancestor

```
>C:.v.v.<...:.PaSa-ya-ya-yaR
^....$<v^cy?<..............
..>-ya-yb-yc^..............
..^ccd*bbd*aad*dy0dSdP<.....
.......>>>v.........<^cx?<.
..........>LbdWad-xc-xb-xa^.
..........^.........a-acd*<.
.................>@cdSaSc.^.
....>......................v
R.by+by+by+bSbP::^c+cab~..C<
```

FIGURE 3.4: Fungera-3 – directed jump ancestor



```
>C:.v.v.<...:.PaSa-ya-ya-yaR
^....$<v^cy?<..............
..>-ya-yb-yc^..............
..^ccd*bbd*aad*dy0dSdP<.....
.......>>>v.........<^cx?<.
..........>LbdWad-xc-xb-xa^.
..........^.........a-acd*<.
.................>@cdSaSc.^.
....>......................v
R.by+by+by+bSbP::^c+cab~..C<

:: -- patterns for calls
R. -- finding organism size and putting it into registers
d* -- allocating child and size correction
?< -- reproduction loop
y0 -- updating column and checking for loop end
.$ -- splitting child and starting from the beginning
```

FIGURE 3.5: Fungera-3 ancestor code parts marked by their functions.

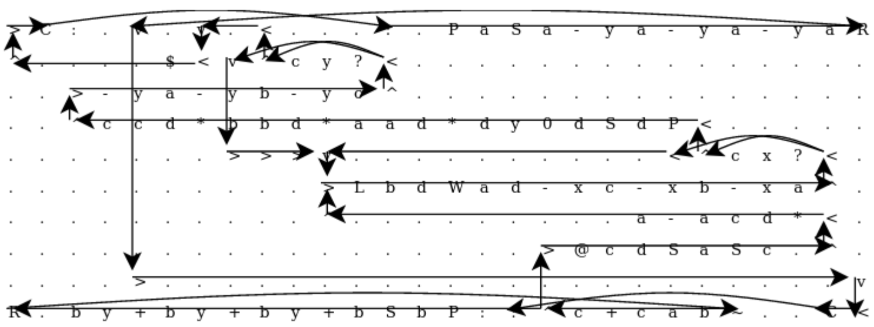
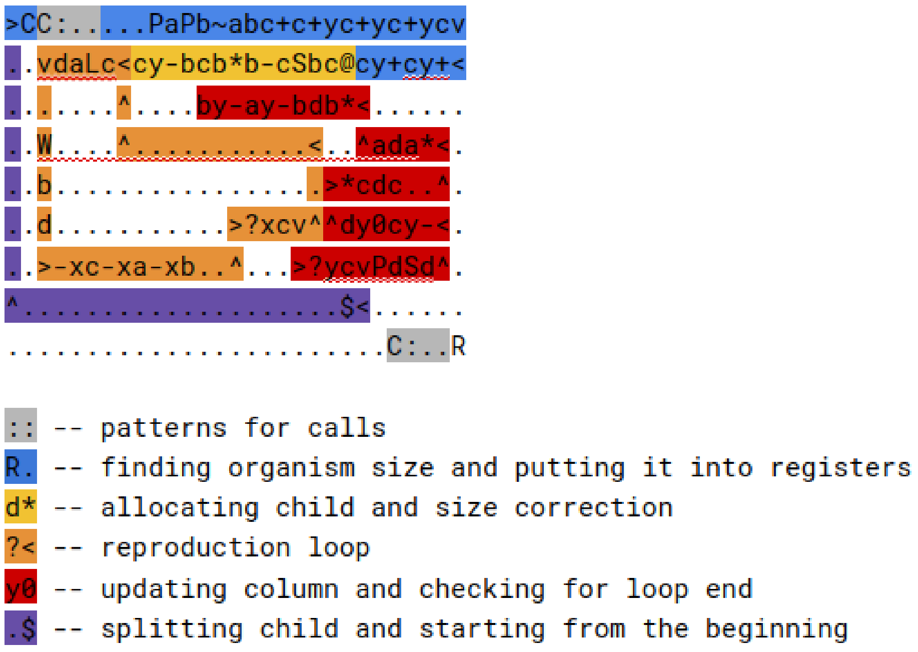


FIGURE 3.6: Fungera-3 ancestor execution flow.

```
>CC:.....PaPb~abc+c+yc+yc+ycv
..vdaLc<cy-bcb*b-cSbc@cy+cy+<
......^....by-ay-bdb*<......
..W....^..........<..^ada*<.
..b...............>*cdc..^.
..d..........>?xcv^^dy0cy-<.
..>-xc-xa-xb..^...>?ycvPdSd^.
^..................$<......
......................C:..R
```

FIGURE 3.7: Fungera-4 – direction independent jump ancestor.



```
>CC:.....PaPb~abc+c+yc+yc+ycv
..vdaLc<cy-bcb*b-cSbc@cy+cy+<
......^....by-ay-bdb*<......
..W....^..........<..^ada*<.
..b...............>*cdc..^.
..d..........>?xcv^^dy0cy-<.
..>-xc-xa-xb..^...>?ycvPdSd^.
^..................$<......
......................C:..R

:: -- patterns for calls
R. -- finding organism size and putting it into registers
d* -- allocating child and size correction
?< -- reproduction loop
y0 -- updating column and checking for loop end
.$ -- splitting child and starting from the beginning
```

FIGURE 3.8: Fungera-4 ancestor code parts marked by their functions.

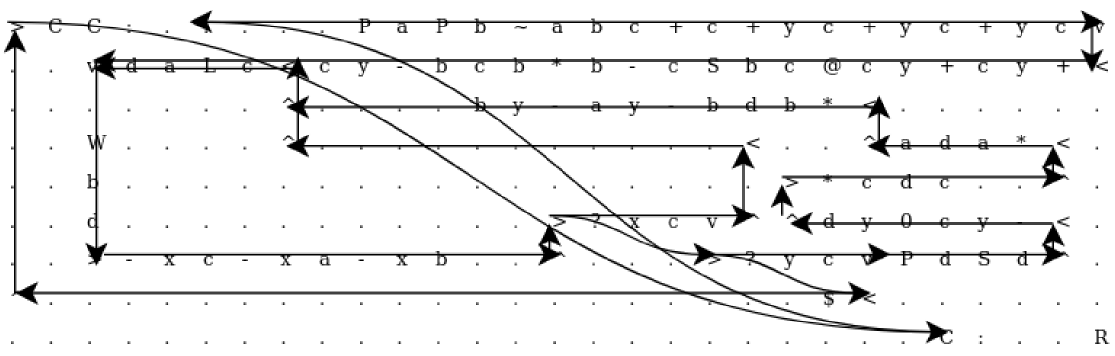


FIGURE 3.9: Fungera-4 ancestor execution flow.

Chapter 4

Investigation of the Fungera behavior

4.1 Additional metrics for investigation

Because Fungera is a living system with many complex interactions between organisms and environment and organisms with each other, the only way to truly understand its behavior is to observe organisms' actions and interactions using a visual debugger. But there's a problem with this approach: we cannot observe raw simulations on each and every run and each and every moment, because, obviously, it takes too much time. Also, because we cannot see all registers of all organisms at the same time. Therefore, we need some additional metrics for a better understanding of simulation behavior and to know where to look and see the general trend behind this evolution.

4.1.1 Total organisms entropy and per-site entropy

There are many definitions of entropy from different fields: from information theory to physics. But on the intuitive level entropy means this: how much potential information can be extracted from this system. How much do genomes of organisms have in common? This helps us measure the convergence of evolutionary processes to some specific genotypes or sets of genotypes.

On a mathematical level, the formula for it looks like this:

$$H = \sum H(i,j),$$

where $H(i,j)$ is a per-site entropy for points of the 2D genome.

A per site entropy is calculated according to this formula:

$$H(i,j) = \sum p_k \log p_k$$

Where p_k is estimated by:

$$p_k \approx \frac{n_m}{N},$$

where n_m is a number organisms with some command in this site and N is a total number of organisms.

Basically, total entropy shows us how diverse genomes are (zero entropy means they're all the same and big entropy means that we have a huge difference between them). Per-site entropy allows us to understand, what areas of organisms are most different between living organisms at this moment and then to see later, which gene variations are associated with which behaviors. This way, we can see the heatmap and see if our mutations are functional and influence something or they are non-critical (for example, one NOP type replaced with the other, like ":" and "." NOP-instructions).

4.1.2 Expected lifetime

Because we have no fitness function for the organism, the only way to estimate how well-adjusted our organism is, we need to know its life duration. But, because simulation is pretty random, the life duration of one organism in simulation is simply not representative of its general fitness and survivability. So, to make this more accurate, we need to observe organisms across many runs and make conclusions about it. Also, it's helpful to understand whether an organism can survive on its own or not.

In order to do this, we introduce a new metric called expected life. Basically, an organism is put into an empty simulation with frequent mutations and its lifespan is measured across multiple simulations with different random seeds and averaged.

4.1.3 Replication/Non-replication

Another criterion of organism fitness is whether it replicates itself. One of the many criteria to determine if something is alive is whether it reproduces and reproduction is frequently included in life definitions. Also, in our simulation, an organism is killed if it doesn't replicate, so it's a good measure of the livability of the whole genome/species, etc.

In order to measure these metrics, an organism is placed into an empty environment with absolutely no mutation, so it's fully deterministic and nothing can break its genome or replication loop. This way, we can see whether it is able to create a child organism (by itself).

4.1.4 Snapshots analysis

We could try to analyze every snapshot, but there is a problem with this approach. Estimating expected lifespan, finding if a genome is a replicator or a non-replicator, and doing other analysis can take up to several minutes, while each snapshot is taken every 20 seconds. Therefore, for 20 seconds of running one(!) simulation, we would need several minutes of calculating some metrics. Therefore, we need some method of selecting snapshots that are most interesting to us – some method of summarizing.

The first step of this analysis is using a simple moving average, which is also called rolling mean with a sliding window for smoothing. This helps us to determine global trends that are not dependent on some short-term anomalies. Also, we see the general tendencies behind short-term fluctuations of values. In our case, as a function of interest, we take a moving average of entropy, because it shows us how diversity is developing in the whole ecosystem.

Then, when we see general trends behind the moving average, phenomenons that are interesting for us, occur in local extremes, when the situation is changing, and when the first derivative is the biggest which is when we can have diverse organisms that replicate the most.

Example of summarizing of entropy plot. Before smoothing:

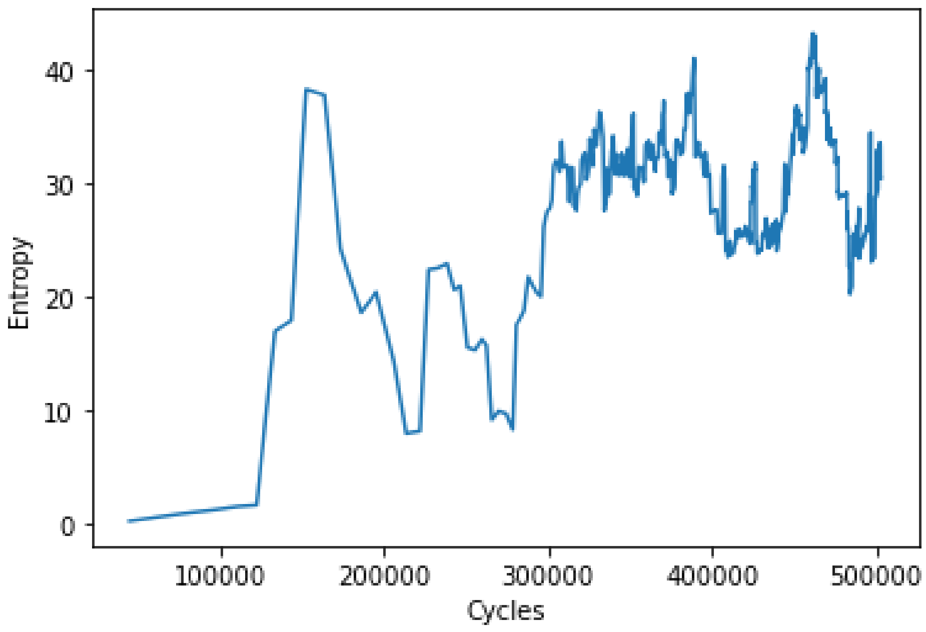


FIGURE 4.1: Before smoothing

After smoothing:

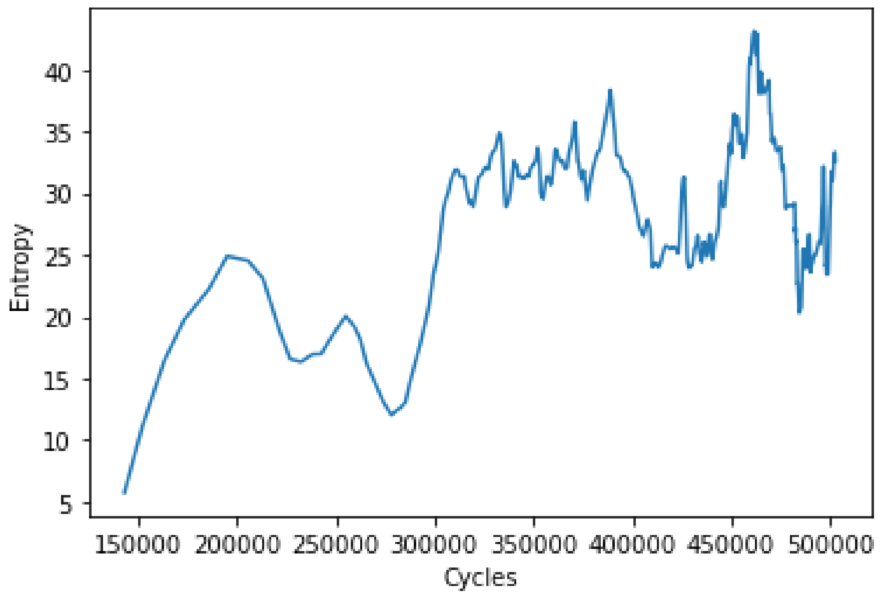


FIGURE 4.2: After smoothing.

After summarizing: By using summary snapshots, in this case, we get the most interesting information with only 14% of snapshots to analyze.

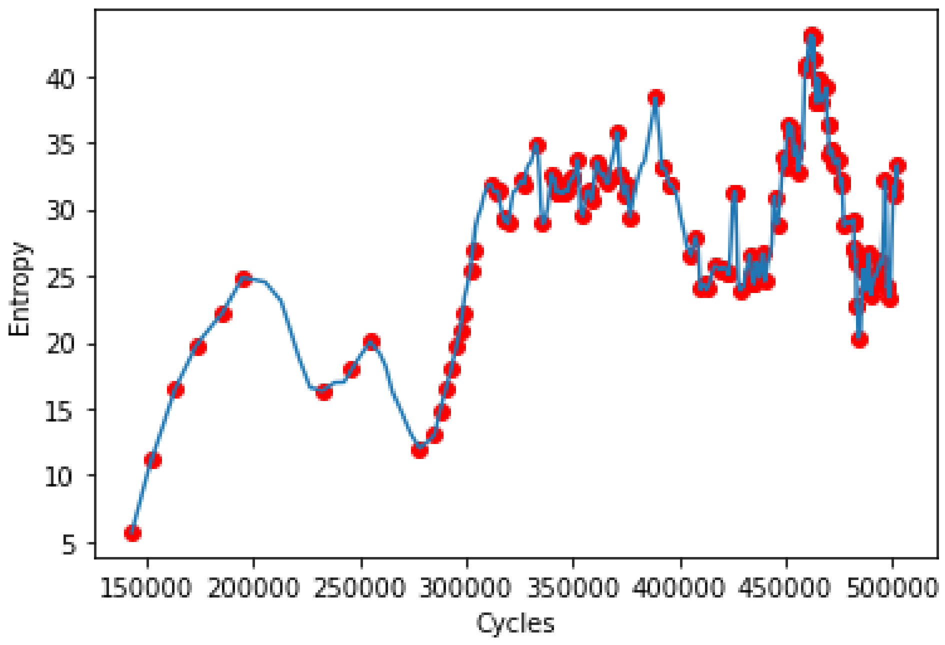


FIGURE 4.3: After summarizing (summary points marked red).

4.1.5 Caching genome metrics

Let's say that we have a snapshot, we have extracted all unique genotypes from it and we need to do further analysis of each genome. For each genotype we need to get the expected lifespan and if it is a replicator or not in order to find interesting species that can survive on their own.

But the problem is that in order to find if it's a replicator and the expected lifespan for one genotype, we need to run at least 5-10 simulations to get the expected lifespan, which can take up to 2 or 3 minutes. Given that we can have up to 4000 organisms in a relatively small simulation (500 to 500), it would take extremely long to calculate these metrics even for one snapshot.

In order to solve this problem, we introduced genome caching: we store all unique genomes with their life expectancies and replication flags in a separate cache file and write into it when we encounter some new genome. This way, we can drastically speed up snapshot analysis.

4.2 Analysis pipeline

So, if we put all the additional analysis metrics and approaches together, we have this analysis pipeline:

1. Per site entropy and simulation entropy is calculated during simulation run and dumped into separate metrics snapshots.
2. All simulation entropies from metrics files are copied into one time series and points, that are interesting and critical are found (by searching for local extrema and points with biggest first derivatives).

3. Then, we analyze genomes in chosen snapshots for expected life and replication/nonreplication
4. Then, we output all this into a file for future analysis

Console utility was developed to automate it.

Chapter 5

Results

5.1 Base instruction set

With basic Fungera, simulation ran until at least 4.5 million cycles. As we can see from the entropy plot, the entropy was continuously increasing, while the number of organisms peaked at about 550 organisms. This shows us that differences between organisms were increasing, while the number of organisms remained almost the same.

Figure 5.3 shows results of the replication/nonreplication analysis.

We can see that the number of self-replicating organisms decreased with time and sometimes became zero even after about 400 thousand cycles. This shows that the starting ancestor was relatively fragile and that even with mutation rate once every 400 cycles it became disruptive for all ancestors.

Another interesting thing is that somehow even after death of the last self-replicator, organisms continued to appear. This can be caused by some aberrant organisms – "microvesicles", allocating random memory chunks and splitting them as a child. Also, the reaper was probably not strong enough to stop aberrant organisms from appearing.

5.2 Instruction set with direction-dependent jumps and calls

The simulation with directed jumps and calls died after about 140 000, but the reaper was harsher this time and killed after only 10 execution errors.

As we can see on this plot, at iteration 125 thousand the number of replicators rapidly decreased, while the total number of organisms increased, which means that they replicated for the last time and then almost all of the child organisms were non-replicating because of some kind of mutation.

If we look at the code of all replicators, we can see that each one of them differs from the starting ancestor insignificantly, by only one command, that is not executing.

5.3 Instruction set with direction-independent jumps and calls

With this instruction set, the situation was different: it was running for 600 thousand of cycles with relatively stable entropy and a slowly decreasing number of organisms. We can say that this instruction set was more robust than previous ones and at least it had a relatively stable population.

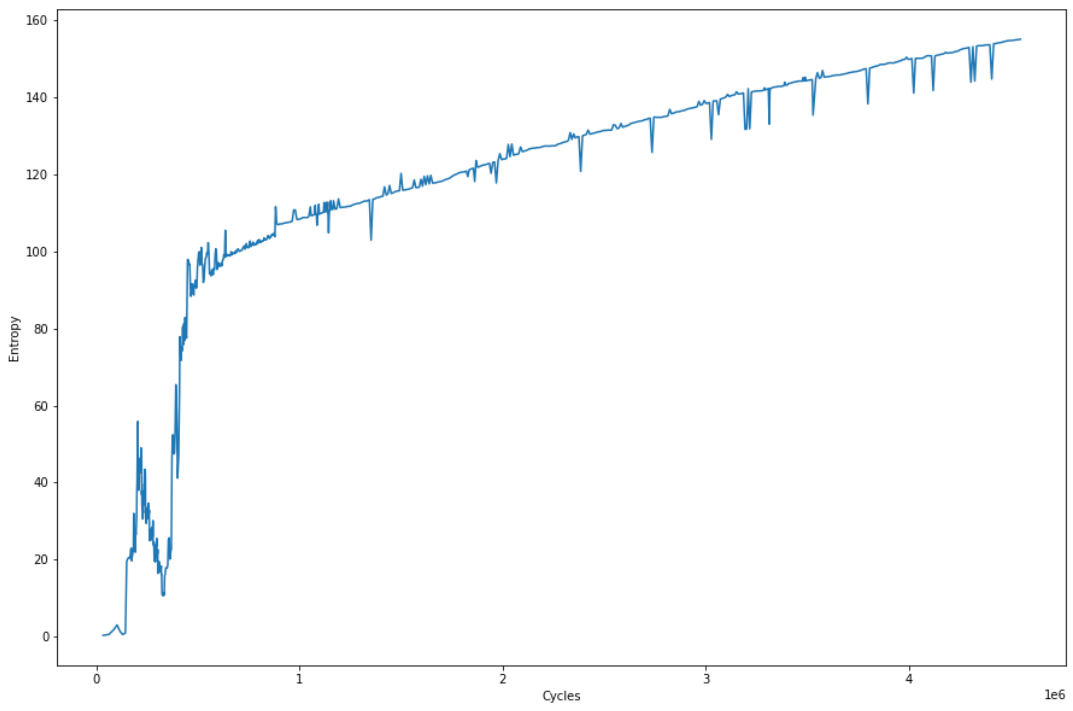


FIGURE 5.1: Entropy – Fungera-1

5.4 Instruction set with error correction (Cellgera)

This one was the most interesting, because it had relatively low entropy and population, compared to previous ones, but it was more robust and resilient to mutations because of the error correction mechanisms.

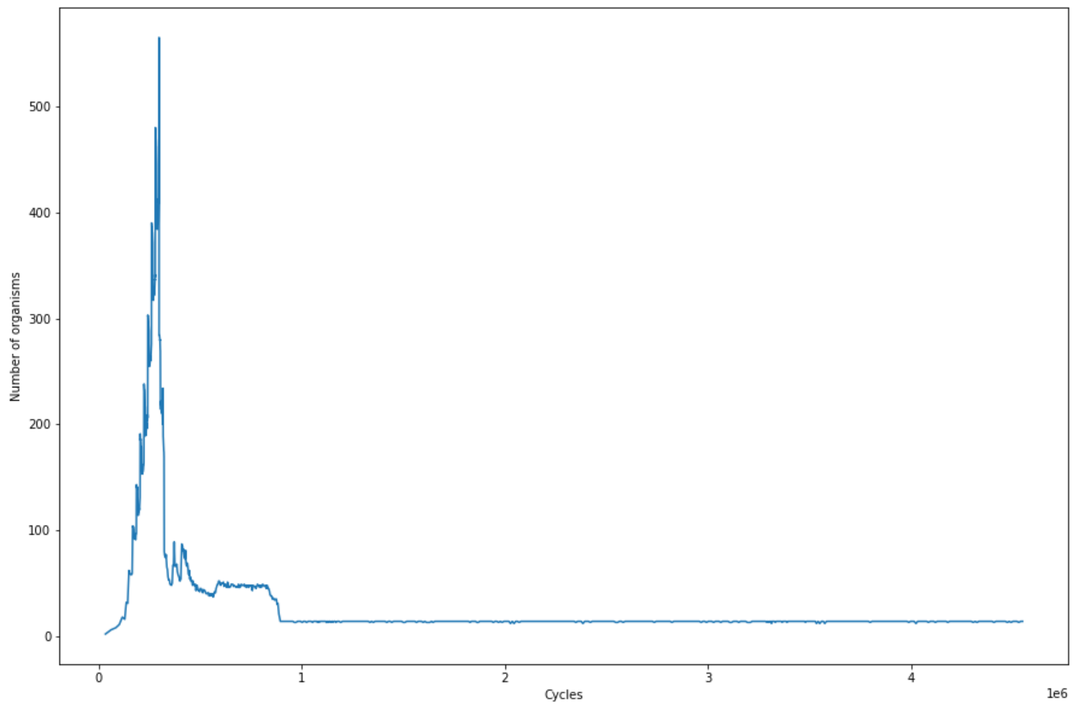


FIGURE 5.2: Number of replicating vs non-replicating organisms – Fungera-1

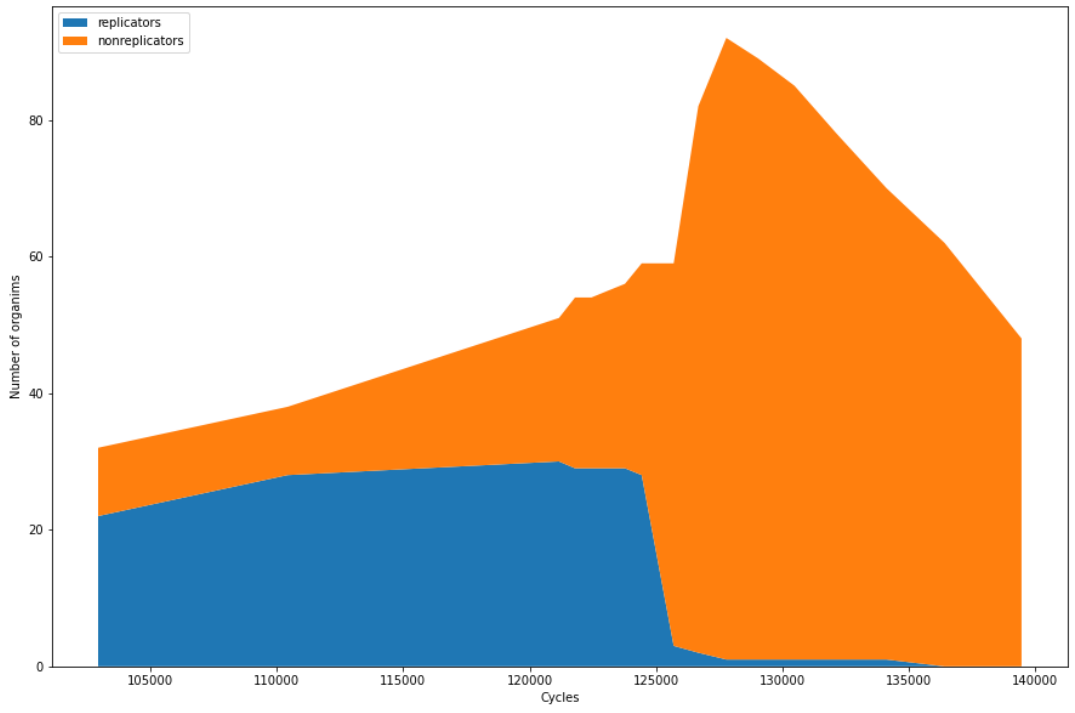


FIGURE 5.3: Replicators/nonreplicators – Fungera-3

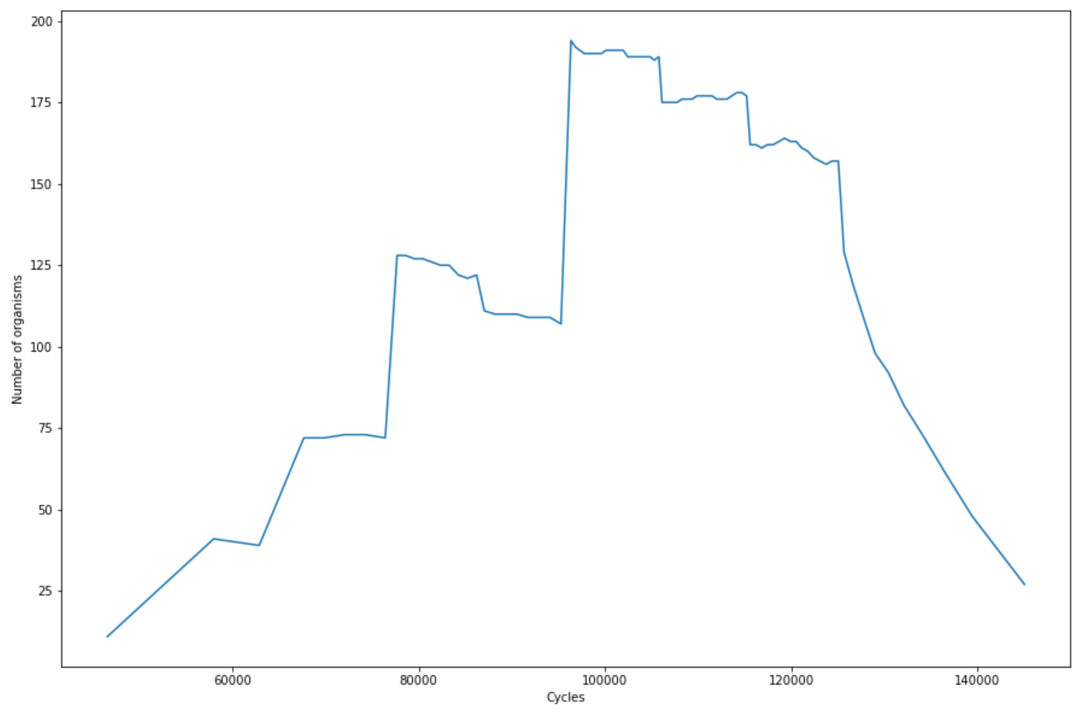


FIGURE 5.4: Number of organisms – Fungera-3

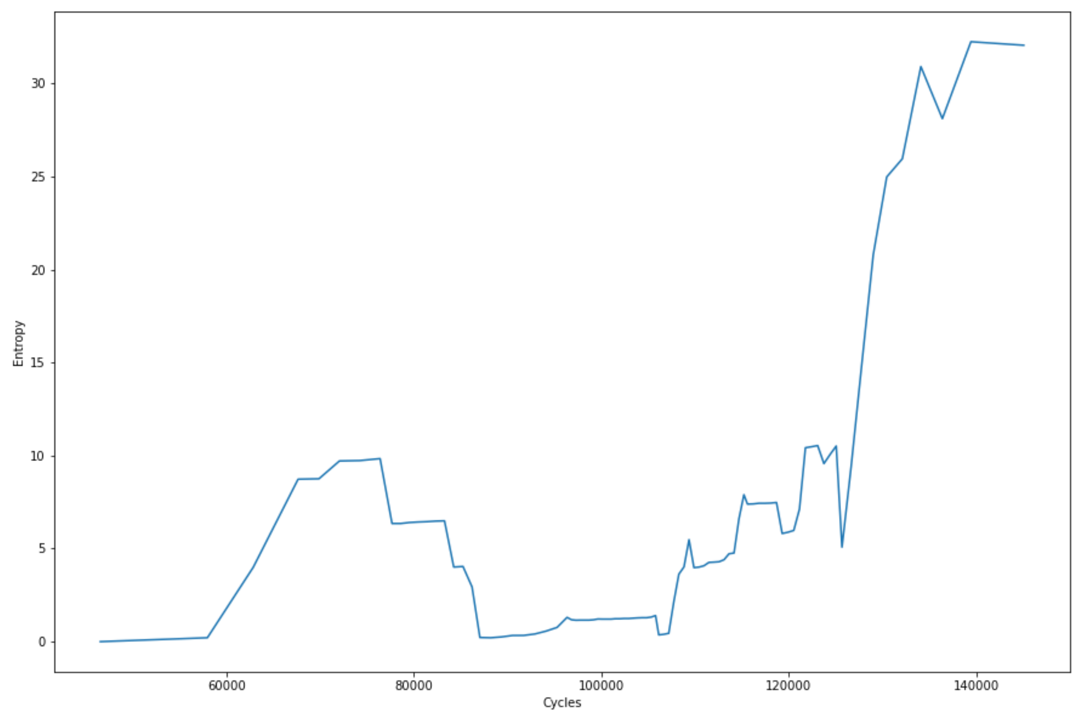
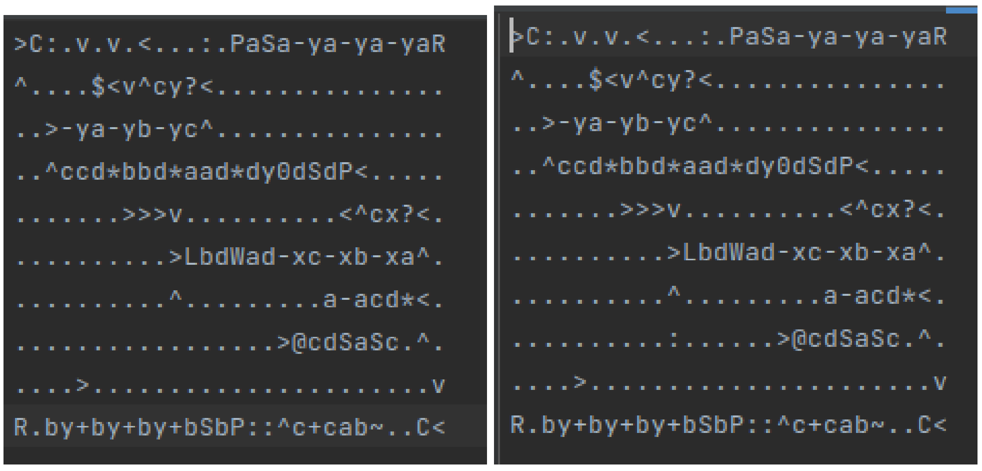


FIGURE 5.5: Entropy – Fungera-3



```
>C:.v.v.<...:.PaSa-ya-ya-yaR
^....$<v^cy?<...............
..>-ya-yb-yc^...............
..^ccd*bbd*aad*dy0dSdP<.....
.......>>>v..........<^cx?<.
..........>LbdWad-xc-xb-xa^.
..........^.........a-acd*<.
.................>@cdSaSc.^.
....>......................v
R.by+by+by+bSbP::^c+cab~..C<
```

```
>C:.v.v.<...:.PaSa-ya-ya-yaR
^....$<v^cy?<...............
..>-ya-yb-yc^...............
..^ccd*bbd*aad*dy0dSdP<.....
.......>>>v..........<^cx?<.
..........>LbdWad-xc-xb-xa^.
..........^.........a-acd*<.
..........:......>@cdSaSc.^.
....>......................v
R.by+by+by+bSbP::^c+cab~..C<
```

FIGURE 5.6: Ancestor – Fungera-3, original and mutated.

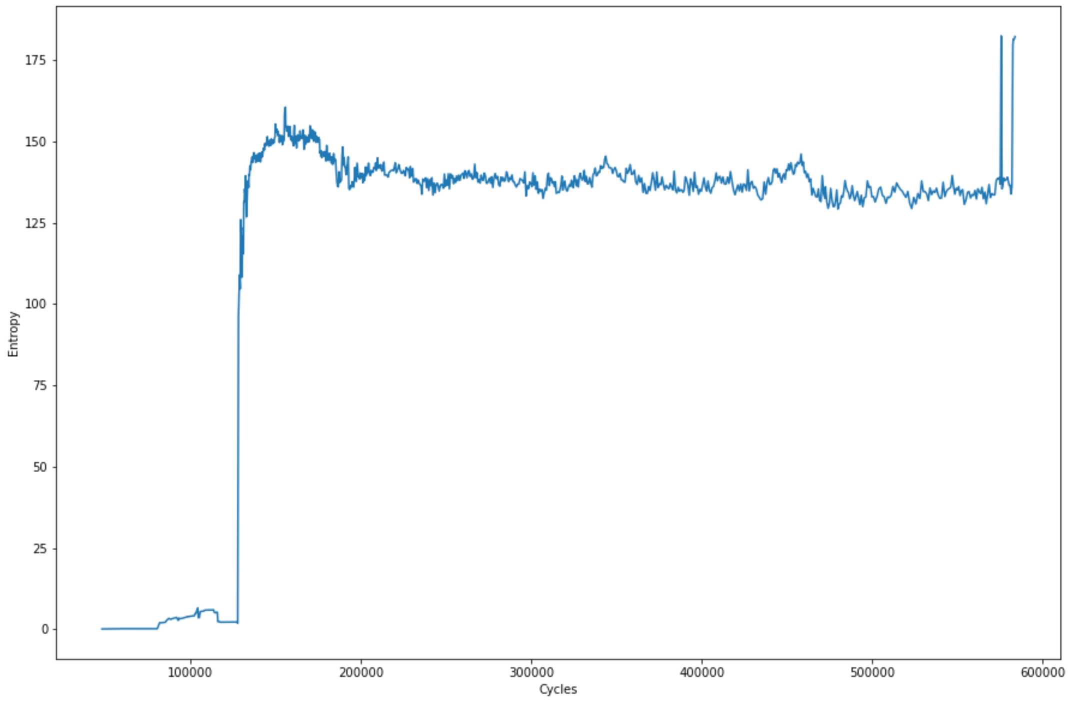


FIGURE 5.7: Entropy – Fungera-4.

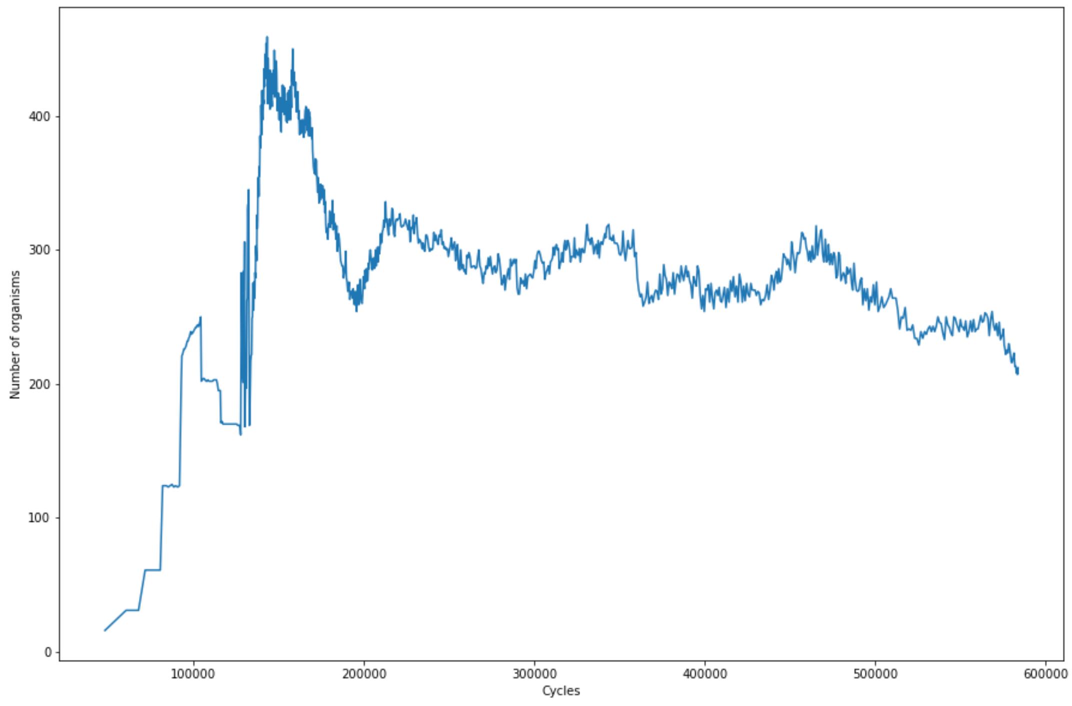


FIGURE 5.8: Number of organisms – Fungera-4.

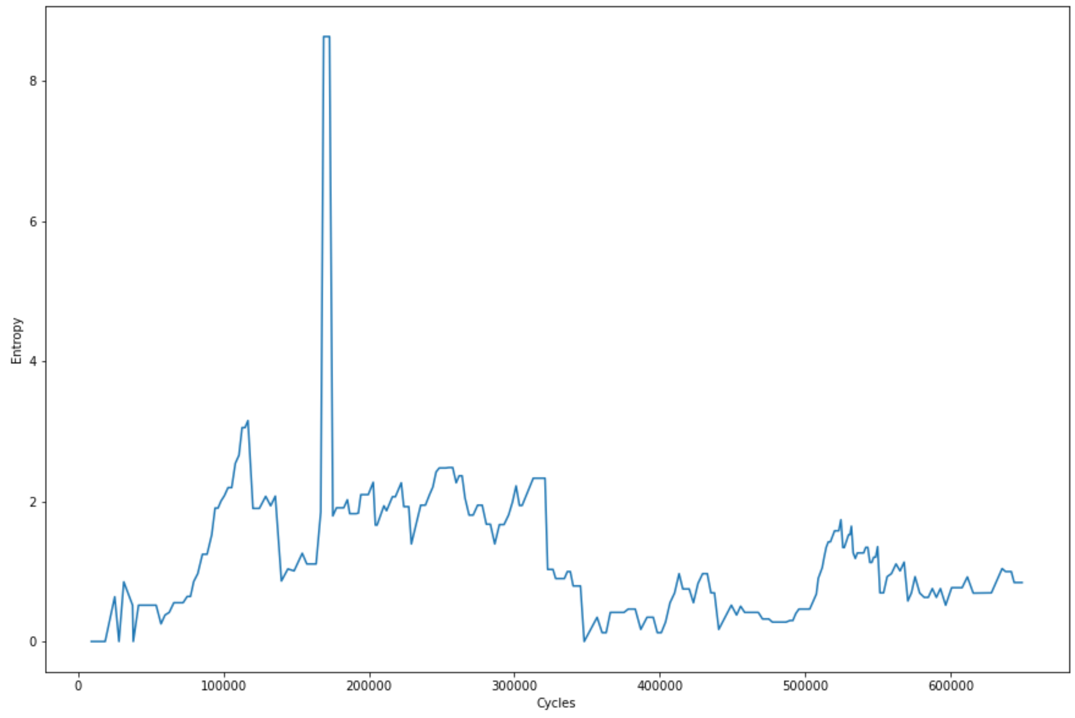


FIGURE 5.9: Entropy – Fungera-2/Cellgera.

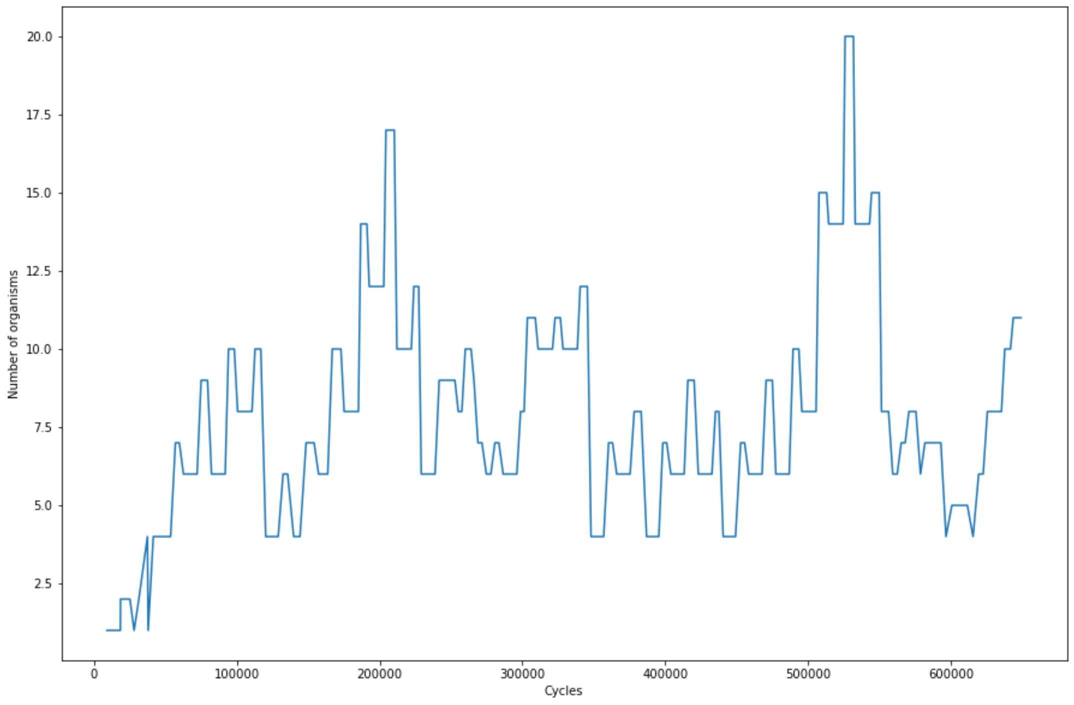


FIGURE 5.10: Number of organisms – Fungera-2/Cellgera.

Chapter 6

Conclusions

6.1 Conclusions

As we can see, even some tiny changes in instruction sets and ancestor organisms have shown to greatly influence the simulation behavior. In the basic Fungera instruction set, the ancestor was not robust enough and therefore it often became extinct before the first million cycles and aberrant organisms prevailed. The same thing was in the instruction set with directed jumps, non-replicators replaced and replicators, and the whole simulation died. What's interesting is that the same parameters instruction set with direction-independent jumps and calls behaved very differently. It had a stable population and entropy, at least to the degree to which the simulation ran. The instruction set with error correction proved to be more resilient because of repair mechanisms but replicated much slower.

The implementation of analysis tools and new instruction sets is located in this GitHub repository: `https://github.com/Arattel/fungera_analysis/tree/master`

6.2 Future work

The experiments with instruction sets were severely bound by time constraints: one run with an instruction set with direction-independent jumps took about 24 hours. Therefore, there was not enough time in the simulation to test all hypotheses and come to some conclusion about the exact relationship between new species. For example, in Tierra simulation parasites started appearing after many millions of cycles. In the case of Fungera, the complexity is much bigger. Therefore, in order to test our hypotheses, we need more runs and faster runs.

So, further research for this looks like this:

1. Testing different ancestors.
2. Trying to write parasites manually in different instruction sets in order to check whether these sets are conductive to parasitism.
3. More runs for better data.

Bibliography

Adami, Christoph (1999). *Introduction to Artificial Life*. Berlin, Heidelberg: Springer-Verlag. ISBN: 0387946462.

Adami, Christoph and et al. (2015). *Avida: Default Ancestor Guided Tour*. https://github.com/devosoft/avida/wiki/Default-Ancestor-Guided-Tour.

Adami, Christoph and C. Titus Brown (June 1994). "Evolutionary Learning in the 2D Artificial Life System "Avida"". In: *Artifical Life IV*.

Baker, Christopher R., Victor Hanson-Smith, and Alexander D. Johnson (2013). "Following Gene Duplication, Paralog Interference Constrains Transcriptional Circuit Evolution". In: *Science* 342.6154, pp. 104–108. ISSN: 0036-8075. DOI: 10.1126/science.1240810. URL: https://science.sciencemag.org/content/342/6154/104.

Callaway, Ewen (2017). "Oldest Homo sapiens fossil claim rewrites our species' history". In: *Nature*. DOI: doi:10.1038/nature.2017.22114..

Chatterjee, Nimrat (2017). "Mechanisms of DNA damage, repair and mutagenesis". In: *Environ Mol Mutagen*. DOI: doi:10.1002/em.22087.

Christodoulou, Fay et al. (Feb. 2010). "Ancient animal microRNAs and the evolution of tissue identity". In: *Nature* 463, pp. 1084–8. DOI: 10.1038/nature08744.

De Dinechin, Floren (1997). "Self-replication in a 2D von Neumann architecture". In: URL: https://pdfs.semanticscholar.org/646a/c824275a688228dc06d2144e25b7b9b00b97.pdf.

Diss, Guillaume et al. (2017). "Gene duplication can impart fragility, not robustness, in the yeast protein interaction network". In: *Science* 355.6325, pp. 630–634. ISSN: 0036-8075. DOI: 10.1126/science.aai7685. URL: https://science.sciencemag.org/content/355/6325/630.

Fernandez, Ariel and Michael Lynch (June 2011). "Non-adaptive origins of interactome complexity". In: *Nature* 474, pp. 502–5. DOI: 10.1038/nature09992.

Forbes, Andrew A. (2010). *Evolution Is Change in the Inherited Traits of a Population through Successive Generations*. URL: https://www.nature.com/scitable/knowledge/library/evolution-is-change-in-the-inherited-traits-15164254/. (accessed: 16.05.2021).

Good, Benjamin et al. (Nov. 2017). "The Dynamics of Molecular Evolution Over 60,000 Generations". In: *Nature* 551. DOI: 10.1038/nature24287.

Greenbaum, Benjamin and Andrew Pargellis (2016). "Digital Replicators Emerge from a Self-Organizing Prebiotic World". In: *The 2019 Conference on Artificial Life* 28, pp. 60–67. URL: https://www.mitpressjournals.org/doi/abs/10.1162/978-0-262-33936-0-ch016.

Kapheim, Karen M. et al. (2015). "Genomic signatures of evolutionary transitions from solitary to group living". In: *Science* 348.6239, pp. 1139–1143. ISSN: 0036-8075. DOI: 10.1126/science.aaa4788. eprint: https://science.sciencemag.org/content/348/6239/1139.full.pdf. URL: https://science.sciencemag.org/content/348/6239/1139.

Lieberman, Erez et al. (Nov. 2007). "Quantifying the evolutionary dynamics of language". In: *Nature* 449, pp. 713–6. DOI: 10.1038/nature06137.

Matute, Daniel R. et al. (2010). "A Test of the Snowball Theory for the Rate of Evolution of Hybrid Incompatibilities". In: *Science* 329.5998, pp. 1518–1521. ISSN: 0036-8075. DOI: 10.1126/science.1193440. URL: https://science.sciencemag.org/content/329/5998/1518.

McKay, Chris P (2004). "What Is Life—and How Do We Search for It in Other Worlds?" In: *PLoS Biol.* DOI: doi:10.1371/journal.pbio.0020302.

Moyle, Leonie C. and Takuya Nakazato (2010). "Hybrid Incompatibility "Snowballs" Between Solanum Species". In: *Science* 329.5998, pp. 1521–1523. ISSN: 0036-8075. DOI: 10.1126/science.1193063. URL: https://science.sciencemag.org/content/329/5998/1521.

Pagel, Mark, Quentin Atkinson, and Andrew Meade (Oct. 2007). "Frequency of Word-Use Predicts Rates of Lexical Evolution throughout Indo-European History". In: *Nature* 449, pp. 717–20. DOI: 10.1038/nature06176.

Pargellis, A (Feb. 2001). "Digital Life Behavior in the Amoeba World". In: *Artificial life* 7, pp. 63–75. DOI: 10.1162/106454601300328025.

Poliakov, Mykhailo (2020). "Evolution of digital organisms in truly two-dimensional memory space: Bachelor's thesis". In:

Ray, Thomas (1991). "Evolution, Ecology and Optimization of Digital Organisms". In: URL: https://www.cc.gatech.edu/~turk/bio_sim/articles/tierra_thomas_ray.pdf.

– (1993a). "An Evolutionary Approach to Synthetic Biology: Zen and the Art of Creating Life". In: URL: http://www.sci.brooklyn.cuny.edu/~sklar/teaching/f05/alife/papers/ray-zen.pdf.

Ray, Thomas S. (1993b). "An Evolutionary Approach to Synthetic Biology: Zen and the Art of Creating Life". In: *Artificial Life* 1.1-2, pp. 179–209.

Soria-Carrasco, Víctor et al. (2014). "Stick Insect Genomes Reveal Natural Selection's Role in Parallel Speciation". In: *Science* 344.6185, pp. 738–742. ISSN: 0036-8075. DOI: 10.1126/science.1252136. URL: https://science.sciencemag.org/content/344/6185/738.

Tan, Chris Soon Heng et al. (2009). "Positive Selection of Tyrosine Loss in Metazoan Evolution". In: *Science* 325.5948, pp. 1686–1688. ISSN: 0036-8075. DOI: 10.1126/science.1174301. URL: https://science.sciencemag.org/content/325/5948/1686.

Tusso, Sergio et al. (2020). "Experimental evolution of adaptive divergence under varying degrees of gene flow". In: *bioRxiv*. DOI: 10.1101/2020.11.02.364695. URL: https://www.biorxiv.org/content/early/2020/11/03/2020.11.02.364695.

Wagner, Peter J., Matthew A. Kosnik, and Scott Lidgard (2006). "Abundance Distributions Imply Elevated Complexity of Post-Paleozoic Marine Ecosystems". In: *Science* 314.5803, pp. 1289–1292. ISSN: 0036-8075. DOI: 10.1126/science.1133795. URL: https://science.sciencemag.org/content/314/5803/1289.

Werner, Thomas et al. (2010). "Generation of a novel wing colour pattern by the Wingless morphogen". eng. In: *Nature* 464.7292, pp. 1143–1148. ISSN: 0028-0836.

Wikipedia (2021). *Esoteric programming language — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Esoteric%20programming%20language&oldid=1012423150. [Online; accessed 29-May-2021].

Wikipedia (2021). *Primordial soup — Wikipedia, The Free Encyclopedia*. [Online; accessed 17-May-2021]. URL: https://en.wikipedia.org/w/index.php?title=Primordial_soup&oldid=1020031403.

Wikipedia (2021a). *RNA world — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=RNA%20world&oldid=1025093805. [Online; accessed 29-May-2021].

– (2021b). *Speciation — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Speciation&oldid=1022365674. [Online; accessed 29-May-2021].