UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Evolution of digital organisms in truly two-dimensional memory space

Author:
Mykhailo POLIAKOV

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 2020

Declaration of Authorship

I, Mykhailo POLIAKOV, declare that this thesis titled, "Evolution of digital organisms in truly two-dimensional memory space" 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:

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Evolution of digital organisms in truly two-dimensional memory space

by Mykhailo POLIAKOV

Abstract

Artificial life is the field of study where researchers use simulations to understand natural life. One of the notable simulations of artificial life is called Tierra. In Tierra, self-replicating programs acting as organisms compete for CPU time and RAM under the pressure of natural selection. In time, various types of organisms develop, including parasites, hyperparasites, and even some form of social relationships. The problem with this simulation is that informational complexity stalls after some initial growth; evolution stops producing new types of organisms and is not open-ended. Within the Tierra simulator, the memory address space is one-dimensional; no matter how far away the resource is located, the effort is the same. This thesis is focused on replacing one-dimensional or pseudo-two-dimensional memory space of Tierra and Tierra-like simulators, like Avida or Amoeba, with real two-dimensional space and refining the concepts of the location and local access to achieve a more open-ended evolution.

Acknowledgements

I want to thank Oleg Farenyuk for bringing me into the world of artificial life, providing wonderful ideas, and guiding me through the process of creating this work. I am grateful to Dr. Thomas Ray and Dr. Bohdan Ostash for reviewing the thesis and providing valuable suggestions on how to improve it.

Contents

List of Figures

List of Tables

List of Abbreviations

OS	**O**perating **S**ystem
CPU	**C**entral **P**rocessing **U**nit
RAM	**R**andom **A**ccess **M**emory
DNA	**D**eoxyribo**N**ucleic **A**cid
RNA	**R**ibo**N**ucleic **A**cid
MIMD	**M**ultiple **I**nstructions, **M**ultiple **D**ata
IP	**I**nstruction **P**ointer
TUI	**T**ext **U**ser **I**nterface
GUI	**G**raphical **U**ser **I**nterface
API	**A**pplication **P**rogramming **I**nterface

Chapter 1

Introduction

1.1 Historical origins of artificial life

It is possible to trace humanity's attempts to understand and replicate the nature of life from antiquity (Langton, 1989). Notably, in Greek mythology, Talos is the giant bronze robot created by the god Hephaestus that is guarding Crete against invaders and pirates (Mazlish, 1995). In the middle ages, Al-Jazari created four automatic robot musicians, who were sitting in a boat and entertained guests at parties (Sharkey, 2007). Famous Leonardo Da Vinci designed and built a mechanical knight capable of humanistic movements during the Italian Renaissance (Panse, 2019). In 1739, French engineer Jacques de Vaucanson created a mechanical duck, which could fake metabolism (Mazlish, 1995). Search for mentions of artificial life in the Google books database, interestingly, not only shows a peak in 1997 but also in 1821. In this period, was a time when Frankenstein or The Modern Prometheus by Mary Shelley was published, and many subsequent works as well (Aguilar et al., 2014).

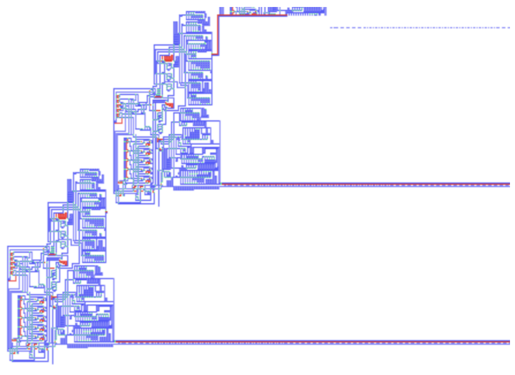
1.2 First cellular automata

John von Neumann, famous for his contribution to mathematics, physics, and computing, created self-replication in cellular automation that he simulated with paper and pencil in 1949. Two years later, he designed an elaborate two-dimensional cellular automaton that would automatically duplicate its original configuration of cells (Wolfram, 2002, p. 1179) (Figure 1.1a). Interestingly, von Neumann created working cellular automaton one year before DNA was discovered. Although he did not use the term artificial life, this can be considered a first artificial life model.

John Conway created the Game of Life in 1970, which provided a new force for research of cellular automata. It models the effects of reproduction, population, and survival. Cells in an infinite two-dimensional orthogonal grid can be in one of two states, populated or unpopulated. Each cell interacts with its eight neighboring cells and, at each step, the following transitions happen (Marinescu, 2017) (Figure 1.1b):

- A populated cell with fewer than two populated neighbors dies. This transition is for under-population.
- A populated cell with more than three populated neighbors dies. This transition is for over-population.

- A populated cell with two or three populated neighbors lives. This transition is for survival.
- An unpopulated cell with three populated live neighbors becomes a populated cell. This transition is for reproduction.


(A) Neumann's automata (Pesavento, 1995)

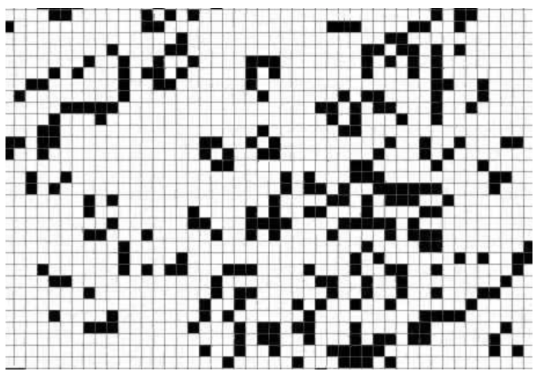

(B) Game of Life (Bettilyon, 2018)

FIGURE 1.1: Cellular automata

1.3 Model of the evolution

Nevertheless, the branch of science named artificial life officially came into being at a workshop only in 1987 at the Los Alamos National Laboratory organized by a computer scientist Christopher Langton. The reason behind its creation was the need to research complex topics in the global economy, natural processes, biology, and evolution in particular (Wilson, 2001, p. 34). The idea was to create new instruments based on computer science and mathematics of nonlinear systems to solve the problems in these subjects. At this workshop, scientists presented mathematical models for the origin of life, self-reproducing automata, programs using the mechanisms of Darwinian evolution, models of growth, and development of artificial plants. While they worked together, it became obvious that all the participants shared a similar set of visions in their prior research (Pfeifer, 2001, p. 4).

They proved that linear models could not describe many natural phenomena. However, nonlinear models cannot be solved easily analytically. They are instead better suited for investigation using bottom-up computer simulations with relatively simple rules. Langton believed at the time that such systems would greatly expand our knowledge of life, nature, and evolution (Langton, 1989, p. 16). Biologist Tom Ray developed a system of self-replicating computer programs competing and evolving for CPU time and memory space in 1991, demonstrating the first instance of artificial evolution by natural selection (Aguilar et al., 2014). Its successor, Avida, by Chris Adami, is used frequently by researchers to conduct biological experiments (Adami & Brown, 1994). Amoeba is the simulator that shows the evolution from the primordial soup to biotic organisms (Pargellis, 2001).

Chapter 2

Related works

2.1 Tierra

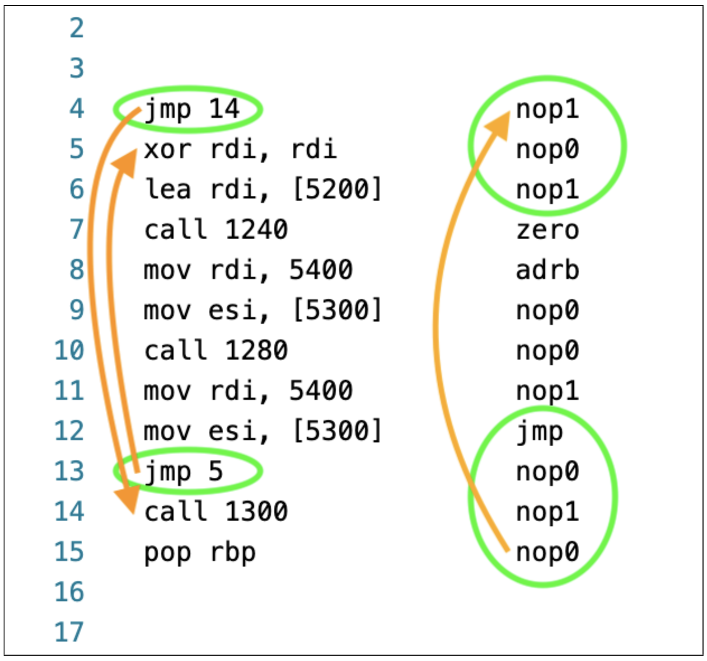
2.1.1 Simulation overview

Tierra is one of the most successful and best known artificial life simulations, developed by biologist Tom Ray in the early 1990s. Natural life is using energy to arrange matter. As such, artificial life can use CPU time to arrange memory space. Biological evolution evolves through natural selection as individuals compete for resources. Artificial life may evolve through the same process, as replicating algorithms compete for CPU time and memory space, and organisms develop approaches to exploit one another. CPU time is the analog of the energy resource. Memory is the analog of the spatial resource. The memory, the CPU, and the OS are acting as elements of the physical environment. In Tierra, memory is a line, so the space is one-dimensional. An organism consists of a self-replicating machine code program that is directly executed by the CPU. Machine codes can be represented as assembler language (Figure 2.1) to make it human-readable. When executed by the CPU, machine codes manipulate bits, bytes, CPU registers, and the instruction pointer. In biology, the analogy is RNA. It similarly has a structure that bears the genetic information and controls the metabolism of organisms. A block of RAM is called a *soup*, reference to Primordial soup, which contains all organisms. The *genome* of the organisms is a series of machine instructions that construct the organism's self-replicating code (Ray, 1991, p. 5).

The simulation is working on a parallel MIMD virtual computer with a CPU for each organism. There is no true parallelism because each CPU executes in a time slice in turn. Each CPU of this virtual computer includes two address registers, two numeric registers, a flags register to indicate error conditions, a stack, and an instruction pointer. The instruction set of a CPU does simple arithmetic operations, moves data between the CPU registers and the RAM, and controls the IP. The organism's CPU is a simple version of the real CPUs. The classic Tierran instruction set contains 32 instructions. The instruction set contains a low number of instructions, and these instructions do not contain numeric operands to significantly lower the number of possible opcode combinations compared to the real CPUs. This unusual behavior allows mimicking a number of codon combinations in the DNA. The instruction set contains both typical instructions of most assembler languages (e.g., `MOV`, `CALL`, `RET`, `POP`, `PUSH`) and some special instructions to allow easy self-replication and evolution.

For example, these include the instruction to allocate memory for child organism and to separate a child organism, thus allowing for the creation of a new organism (Ray, 1991, p. 6).

Another vital feature is addressing by the template. In most assemblers, when a JMP instruction is performed, the IP jumps to another address in memory specified in the instruction. In the biological system, by contrast, one molecule presents a template on its surface, which is complementary to some surface of another molecule. Diffusion draws the pair together, and the complementary formations allow them to communicate. Addressing by the template is done by the Tierran `JMP` instruction. Each JMP instruction is followed by a sequence of two kinds of `NOP` operations: `NOP0` and `NOP1`. This sequence is the Tierran template. The system will seek out in both directions from the `JMP` instruction scanning for the next appearance of the complementary sequence. For example, if the template is `NOP0 NOP1 NOP0`, then its complement is `NOP1 NOP0 NOP1` (Figure 2.1). When the system finds the complementary template, the IP will move to the end of the complementary template and continue operation. Also, Tierra uses templates to denote the start and end of the organism's genome. For optimization reasons, the finding range is limited (Ray, 1991, p. 7).



```
 2
 3
 4   jmp 14                    nop1
 5   xor rdi, rdi              nop0
 6   lea rdi, [5200]           nop1
 7   call 1240                 zero
 8   mov rdi, 5400             adrb
 9   mov esi, [5300]           nop0
10   call 1280                 nop0
11   mov rdi, 5400             nop1
12   mov esi, [5300]           jmp
13   jmp 5                     nop0
14   call 1300                 nop1
15   pop rbp                   nop0
16
17
```

FIGURE 2.1: Normal addressing vs. template addressing

2.1.2 Operation system

The Tierran simulation runs on a virtual computer, and the OS provides memory allocation services. Each organism has exclusive write privileges inside its allocated block. The size of an organism is just the size of its allocated block, which usually matches the size of the genome. While write privileges are protected, read and execute privileges are not. An organism can read the code of another organism and also execute it, but it cannot rewrite the code of another organism. It means that Tierran organisms have a digital analogy of a semipermeable membrane. Each organism may have exclusive write privileges in at most one or two blocks of memory:

the original one, and a second block which it may create when executing the MAL instruction. In the second block, the child organism builds gradually. When Tierran organisms divide, the child organism in the second cell has its separate CPU with IP, stack, and registers. It becomes a separate organism that can allocate its second block of memory. The parent organism can also create its next children. The time slicer executes the instructions for each organism CPU in turn and simulating parallelism. It is possible to configure the slicer in a way to give advantages to small organisms, large organisms, or be size-neutral. (Ray, 1991, p. 9) (Figure 2.2).

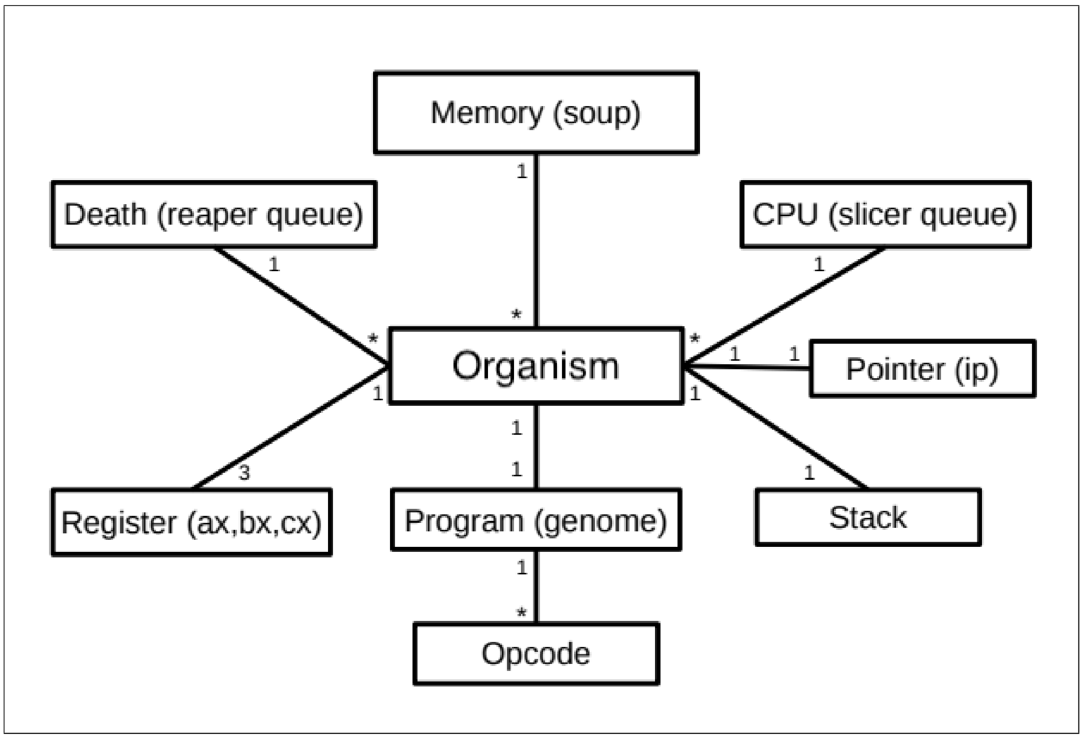


FIGURE 2.2: Tierran operation system model (Hickinbotham & Stepney, 2015, p. 2)

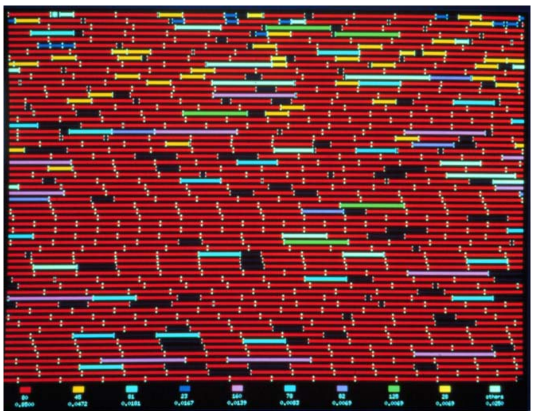
Self-replicating organisms in a fixed size memory would quickly fill the memory. It is reasonable to avoid this situation by creating mortality. The Tierran operating system includes a reaper queue that starts killing organisms by removing them from a queue when the memory fills to some specified level. Organisms at the top of the queue deallocate their memory resource, and the queue removes them from the simulation. When an organism executes an instruction that generates errors, it moves up the queue. The result of the reaper queue is causing organisms that generate errors to die more quickly. Robust organisms, therefore, stay longer, but generally, the probability of death grows with time (Ray, 1991, p. 9).

The evolution starts from a single self-replicating program, which is 80 instructions long. This program is the ancestor organism. Its task is only to self-replicate, and it has no other unusual evolutional behaviors. In order for evolution to happen, there must be some modification in the code of the organisms. The OS randomly flips bits in the memory, and the instructions may sometimes produce unexpected results. This randomness is similar to mutations made by cosmic rays and has the effect of restricting any organism from living forever, as it will ultimately mutate to death. Also, bits flip randomly at some rate while copying instructions during the replication of organisms. In addition to mutations, the execution of instructions is sometimes flawed. For example, the increment instruction normally adds one to its register, but it sometimes adds two or zero. (Ray, 1991, p. 10).

2.1.3 Evolutionary results

Once the memory is full of replicating organisms, individuals are initially short-lived, generally reproducing only once before dying (Ray, 1991, p. 19). More slowly, there appear new genotypes of the size of the ancestor, and then different sizes. There are changes in the code of each size class, as new modifications appear, some of which grow in number, eventually succeeding the ancestor. Over time, new size classes appear, and the community becomes diverse, there is a greater variety in organism types. Several different behavior types of organisms appear which demonstrate micro-evolutional results (Ray, 1991, pp. 12-14) (Figures 2.3a and 2.3b):

- **Parasites** cannot replicate in an isolated environment, and it needs the code of another living organism to replicate. The first parasite has a length of 45 instructions and can replicate faster than the ancestor organism. They use other organism replication code. However, the relationship is harmless for the host.
- **Immunity to parasites** develops in children of ancestor organisms. If too many hosts become immune to parasites, parasites become extinct. Hosts and parasites organisms living together demonstrate the Lotka-Volterra population cycling. Some parasites might circumvent this immunity.
- **Hyper-parasites** use the parasites to replicate their code, so the child of parasite becomes another hyper-parasite. In some time, the social Hyper-parasites appear, which drive parasites to extinction. They are smaller than the ancestor, so they need to group up in order to reproduce.
- **Cheaters** are the evolution of hyper-parasites. They invade communities of hyper-parasites and make hyper-parasites reproduce the cheaters. Cheaters are on top of the simulation food chain.


(A) Hosts (red) are very common. Parasites (yellow) are starting to appear

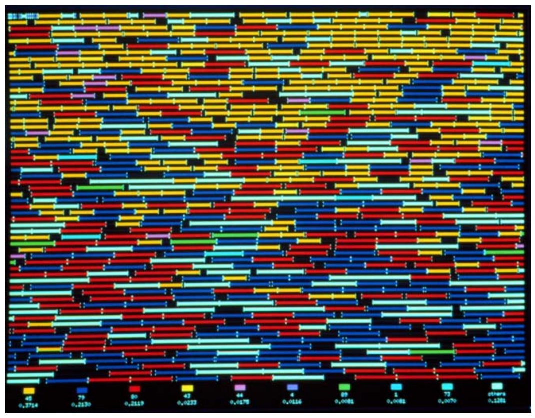

(B) Immune hosts (blue) are increasing, driving the parasites into groups

FIGURE 2.3: Tierran memory snapshots with organism types (Ray)

When the simulator is running for hundreds of millions of instructions, various organisms emerge. During this time, it is possible to look at macro-evolution. Tierra appears to show two models of evolution depending on what organism size time slicer favors: phyletic gradualism and punctuated equilibrium. If the simulation favors large organisms, it usually leads to continuous incrementally increasing sizes until the sizes stay the same for a long time so we can look at phyletic gradualism. If the simulation is neutral, there is a pattern with periods of stagnation, which follow periods of fast evolutionary change, this appears to be the model of punctuated equilibrium (Ray, 1991, p. 15) (Figure 2.4).

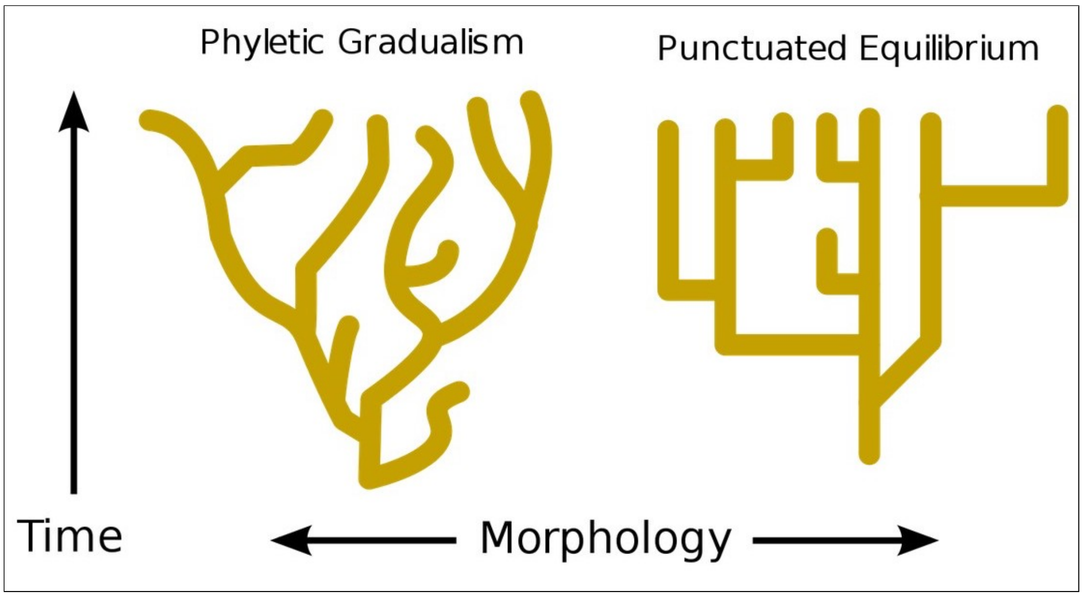


FIGURE 2.4: Phyletic gradualism vs. punctuated equilibrium (Jensen, 2015)

2.2 Avida

Avida is a Tierra-inspired artificial life simulator by Chris Adami with further expansions and redevelopment by Charles Ofria. The critical difference between Tierra and Avida is how they organize the memory. In Avida, each organism has its memory, and it is not available to other organisms even for read access. Thus, there is no parasitism in Avida (Adami & Brown, 1994, p. 1). Authors call the system two-dimensional, but it is also possible to interpret that the system has multiple, isolated one-dimensional memories . The genome of the organism initializing in the program memory. Execution starts with the first command in memory and proceeds sequentially: Instructions execute one after another unless there is a jump. The memory space is circular, and when the CPU executes the last command in memory, it will loop and resume execution with the first command again. However, the memory has a clear entry point, necessary for the creation of child organisms. The looped memory makes the organism genomes simpler than in Tierra, but allow them to achieve similar results. In Avida, there are only 24 instructions compared to Tierran 32 (Ofria, 2003, p. 11).

Another new Avida feature is called labeled heads in the CPU. Heads are pointers to locations in the memory, like the IP. They eliminate the necessity of absolute addressing of memory positions. There are four heads; one of them is the instruction pointer. The other three heads are read head, write head, and flow control head. The read and write heads are for the self-replication. The read head indicates the position in memory from which to read commands, and the write head likewise indicates the position to which write commands. The flow control head for jumps and loops. The CPU also includes three registers and two stacks, with the ability to switch between them, the visualization of the Avida CPU and memory in Figure 2.5 (Ofria, 2003, p. 12).

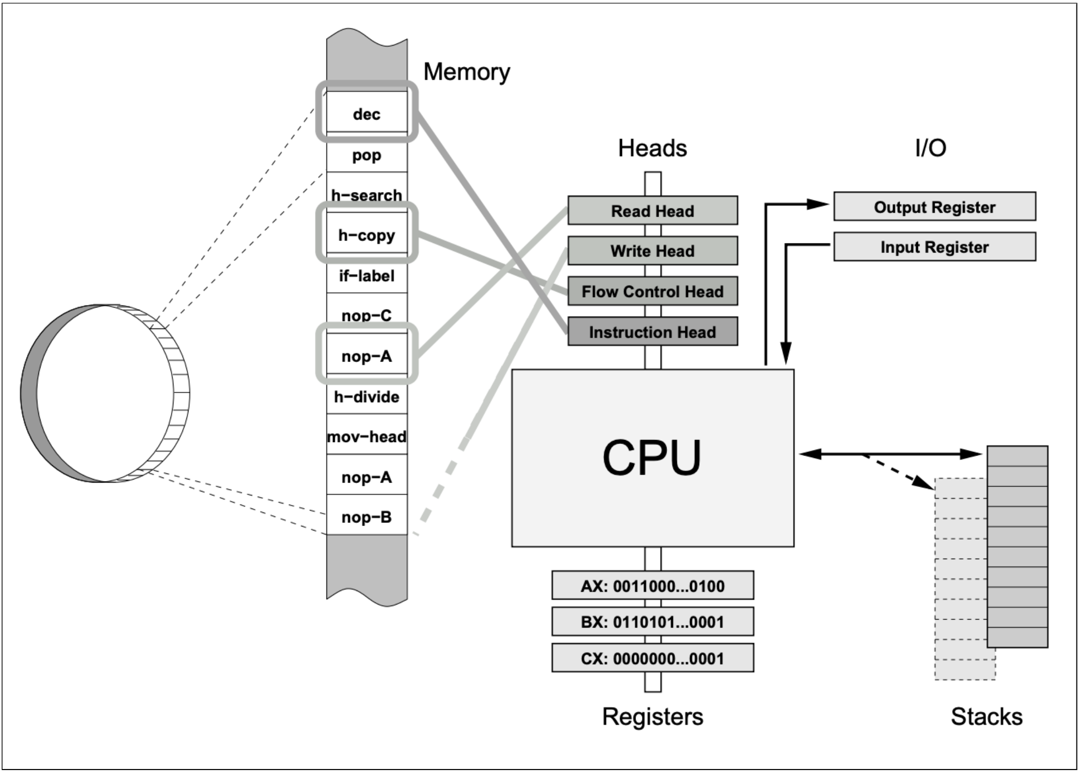


FIGURE 2.5: CPU and memory structure in Avida (Ofria, 2003, p. 74)

In Avida, the time slicer will give different organisms different amount of time to execute one command (Ofria, 2003, p. 22). This amount depends on the organism's count of rewards. The organism receives the reward when it completes any successful operation specified in the simulation configuration. So, better algorithms receive more CPU time, and failing organisms might execute very slowly. The organism also passes down the rewards to its children. Death occurs only when memory fills up, at which time the system deletes the oldest organisms.

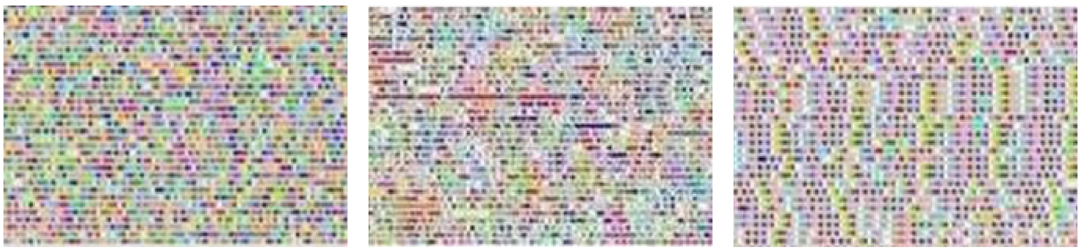
Avida is much more like a proper software package compared to Tierra. It is highly configurable, and the reward system allows to control the evolution priorities and perform user-specified tasks. Avida also has a GUI module, various analytical and statistics tools, and well-written documentation (Ofria, 2003, p. 9). Researchers use Avida to study evolutionary dynamics, monitor populations, and introduce new additions; for example, introducing organisms of different sex and these experiments raise general interest in biologists (Pollac & Bedau, 2004, p. 341).

2.3 Amoeba

The main difference in Amoeba simulation from Tierra and Avida is that there is no ancestor organism placed at the start of the simulation. Instead, replicators are emerging from the memory full of random instructions. The memory can be one-dimensional, like in Tierra or multi one-dimensional, like in Avida, depending on the version of Amoeba. The Amoeba uses a simpler addressing scheme than that Tierra and Avida use. Instead of having templates from NOP0 and NOP1, any instruction has its randomly assigned label, and the label has its complement label; thus, it is possible to use any instruction for template formation (Pargellis, 2001, pp. 4-5). The replication process involves four main stages: register initiation, memory allocation, copying, and division, the same as in Tierra. Register initiation involves determining the cell's size. Memory allocation, done with the MALL command, dynamically allocates a virtual CPU to the child cell. The CPU is not activated, and mutations do not happen until the parent organism has executed a divide command (Pargellis, 2001, p. 6).

From the random memory, firstly, emerge very inefficient ancestor organisms, that are more prebiotic than biotic. They typically reproduce the code only once before they are killed. These organisms might lose their instruction pointer after copying its code to a child, or its copies only a portion of its code and child gets killed. Afterward, more advanced organisms appear with a lot of unused code (probiotic) (Pargellis, 2003, p. 5), and they evolve into a colony of organisms more similar to Tierran organisms, which are biotic. Even some Tierran-like parasites appear as well, but not frequently. So, we can see prebiotic organisms appear from randomness and their evolution to biotic organisms via probiotic organisms (Pargellis, 2001, pp. 9-10).

In Figure 2.6, we can observe the state of memory during different evolution stages in Amoeba. In Figure 2.6a, the memory only consists of random instructions. In Figure 2.6b, the memory becomes more organized (horizontal lines are organisms), and finally, in Figure 2.6c, we can see patterns of biotic replicators. The biotic world is not as diverse as in Tierra; for example, hyperparasites never form, and the parasite and host coexistence ecology never persists more than 1000 cycles (Pargellis, 2001, p. 12). Overall, the primary wonder of Amoeba is the appearance of replicators from random code without the ancestor programmed by a human.



(A) Random memory (B) Prebiotic organisms (C) Biotic organisms

FIGURE 2.6: Memory space in Amoeba (Greenbaum & Pargellis, 2016, p. 4)

Chapter 3

Two-dimensional artificial life simulator

3.1 An idea to increase the complexity

Artificial life simulators such as Tierra and Avida produced a rich diversity of organisms initially, but eventually complexity stalls (Standish, 2008, p. 1). There were multiple attempts to improve Tierra. For example, this might be exploring environmental biases in Tierra, like making the number of mutations not depending on organism size (Hickinbotham & Stepney, 2015, p. 7), or replacing instructions with rule-sets (Sugiura, 2003, p. 1). Nevertheless, there is no decisive evolutionary improvement recorded in these works. There should be another issue that prevents evolution, and the main suspect is the one-dimensional memory space of Tierra. It doesn't matter how far (in the memory address space) the resource is located, the effort to use it is still the same. So, such memory space is close to infinite-dimensional space in a physical sense. The instructions could operate with memory cells far away from the organism, but the processing takes the same amount of CPU time as if these cells are nearby. Authors of Avida and Amoeba recognize the problem, but only manage to create a pseudo-two-dimensional solution. There are multiple one-dimensional memory lines, but the code is executing only along these lines, and there is no ability to interact between them. The simulation in the real two-dimensional space could be a solution. While it is reasonably suggested that enforcing real-world properties on the computer model is inefficient and RAM topological properties could be efficiently used to simulate artificial life (Ray, 1993, p. 18), we still believe it could be one of the key points to the breakthrough, provided one can develop the correct approach to implement it.

3.2 Befunge

The two-dimensional programming languages are mostly created for fun and considered esoteric (i.e., not for software development). Firstly, it would not be practical to use them in real applications. Secondly, they are hard to compile, and only a few of them are Turing-complete. One of the canonical examples of such languages is Befunge, which is stack-based. Chris Pressey created the language in 1993 as an attempt to make a language that is as hard to compile as possible. However, several compilers have appeared. The original Befunge has restricted grid and stack sizes.

The Befunge-98 removes these limitations and thus is Turing-complete. There are IP direction change operations (<, >, v, ^), and most other Befunge operations are stack manipulation. The most exciting operation is p, which allows changing any character in the grid to another character, thus allowing for self-replicating code. The same instruction could execute from every direction (Esolang, the esoteric programming languages wiki, 2020). Here is the classic "Hello World!" written in Befunge:

```
>             v
v  ,,,,,"Hello"<
>48*,         v
v,,,,,,"World!"<
>25*,@
```

Befunge's self-replicating abilities are very similar to the ones provided by commands from the Tierran instruction set. By combing the two-dimensional nature of Befunge with the Tierran instruction set, we get an instruction set for a two-dimensional artificial life simulator. The two-dimensional simulator we discuss in this work is called **Fungera** (Be*funge* + Tie*rra*).

3.3 Organism structure

Since Fungera is a two-dimensional simulation, the organism's CPU should be changed accordingly. It is very similar to the Tierran CPU, with the main exception is a new element, the delta. The delta vector controls the direction, in which the IP goes. The IP can go up, down, right, and left on Fungera's memory grid. After the end of each cycle, we add delta to the IP to make the IP go in the direction we specify. In addition, the organism has two memory blocks (one for the organism itself and an optional one for its child), four general-purpose registers, instruction pointer, and a stack with configurable size. All CPU elements can only contain two-element vectors, for simplicity. Complete CPU structure is in table 3.1.

CPU element	Description
RA	General-purpose register A
RB	General-purpose register B
RC	General-purpose register C
RD	General-purpose register D
IP	Instruction pointer
Delta	IP direction
Stack	Stack of configurable size (default 8)
Main memory block	Allocated memory block for organism itself
Child memory block	Optional memory block for child allocation

TABLE 3.1: CPU structure in Fungera

3.4 Instruction set

Instruction set in Fungera is a crossing between the Tierran instruction set and Befunge. The instruction set also includes various modifiers for registers or vector elements. For example, to do subtraction between registers RA and RC and putting the result in RA, in Tierra, we would call `SUB_AC` instruction, but in Fungera, we write `~aca`. The Fungeran subtract instruction needs three register modifiers to perform the operation. Modifiers allow flexibility since we do not need to create an instruction for each operation we perform and make the two-dimensional genome code more readable. Sometimes, we need to perform an operation only on the first or second element of the vector. For example, `-xa` will decrement the first element of the vector in RA or `+yd` will increment the second element of the vector in RD. These modifiers are similar to modifiers in SALIS by Paul Oliver, a modern retake on Tierra (Oliver, 2019). Also, we can call the instruction from all directions, `*bac` is the same as `cab*`, if IP moves from right to left.

The Fungeran instruction set includes Befunge style IP direction modifiers, and two no-operation template constructors. In Fungera, the usage of templates is restricted to denoting the start and end of the organism. Therefore there is no jump instruction present. Because IP can go in all directions, it is possible to move IP everywhere with direction modifiers, and the organism will consume CPU time with each step it takes. Fungera contains fewer arithmetic operations than Tierra; there is no shift, negation, division, or multiplication present. We do not need them to replicate organisms. We can find the template by using find template instruction (`&.:.` will find a template `:.:` address in the direction of IP). One of the most complex instruction is for conditioning. It compares a vector or an element of the vector to zero (zero vector). If it is zero, the IP skips the instruction after the condition with modifiers. For example, `?xb^v` will execute `v` if RB is [1, 2] and `^` if RB is [0, 2]. For replication, it is possible to allocate child organism memory, read and write instructions to/from memory, and split the child to separate organisms. The full list of instructions is in the table 3.2.

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	Decrement 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 3.2: Instruction set in Fungera

3.5 Memory

In Fungera, the memory has a two-dimensional grid design, which all organisms share. As a result, the address of each memory cell is a two-element vector denoting position on both the x-axis and y-axis. The memory does not have connected boundaries, like in Avida's memory, so it is not circular. The comparison between Tierra, Avida, Amoeba and Fungera memory designs is in Figure 3.1. There are no restrictions for the organism to read the memory, but a particular organism can write to the memory only when it has the child's memory allocated, and this write permission is not restricted – it can write anywhere. The mutation occurs only by adding random instructions to the memory at the configurable rate, and the instructions always produce the expected result.

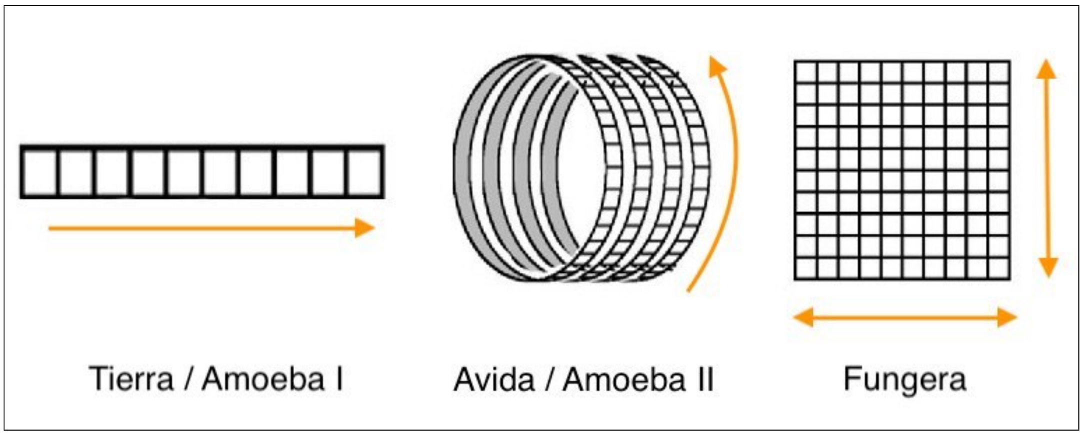


FIGURE 3.1: Tierra, Avida, Amoeba vs. Fungera memory design

3.6 Queue

The Queue contains each organism in the simulation. Each organism makes a cycle with its CPU. When CPU cycles, the IP moves in delta direction and executes the genome code. Due to mutation, some code will not make sense or break simulation and therefore produce an error. The queue counts errors for each organism and controls the organism population. The death occurs in three ways: the queue can kill some percentage of organisms (typically 50%) when the allocated memory fills the required amount (typically 85%); kill one organism that produces too many errors (typically 1000); kill organisms after some cycles (typically 25000) without created children. The queue provides time slicer services to simulate parallelism; however, in Fungera, it is the same for all organisms and is not configurable like in Avida.

3.7 Ancestor

The ancestor in Fungera has a size of [17, 23] and can replicate in all directions one by one, and it needs around 18000 cycles to replicate once. It uses templates to find its start and end and thus determines its size by subtracting them. After that, it allocates the child of its size to the right, when it can find unallocated space for the child's memory block. In a loop, the organism copies itself to the child memory.

When copying is complete, the child becomes a separate organism. The organism continues to create other children up, down, and to the left. The ancestor can be optimized, and we expect to see this optimization during evolution. Here is complete genome of Fungeran ancestor organism:

```
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.............<.....
@>...................:^
^..<...................
```

Algorithm of the Fungeran ancestor is shown on the Figure 3.2.

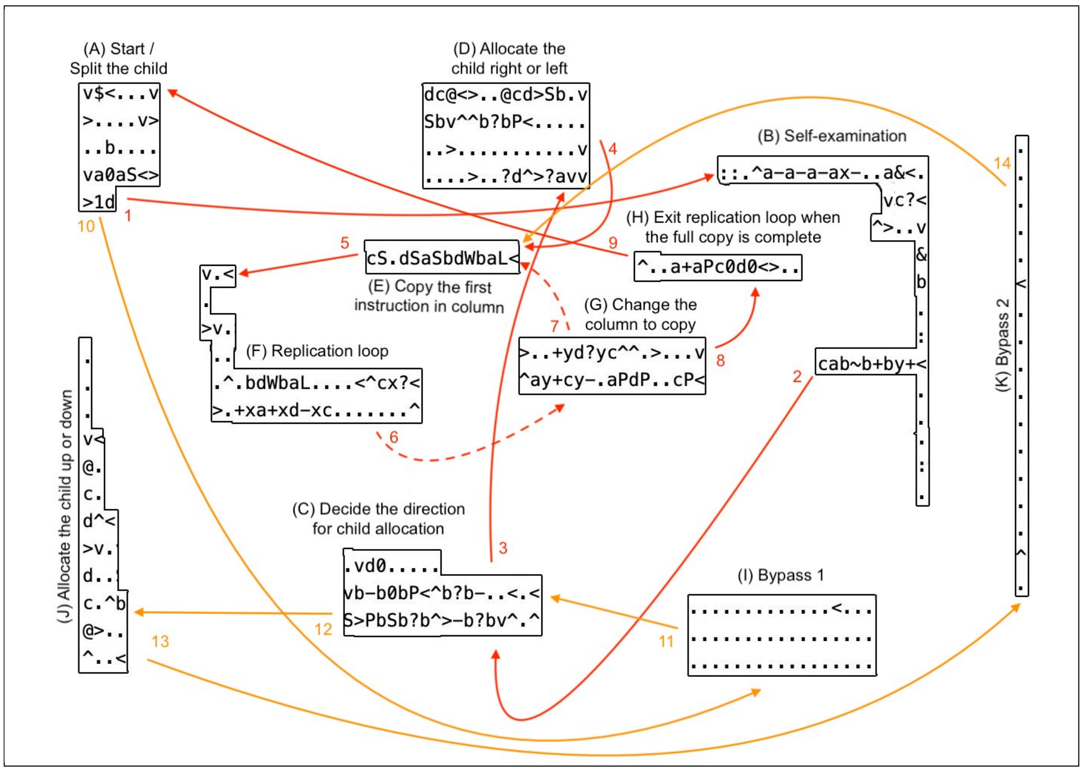


FIGURE 3.2: An overview of the Fungeran ancestor algorithm

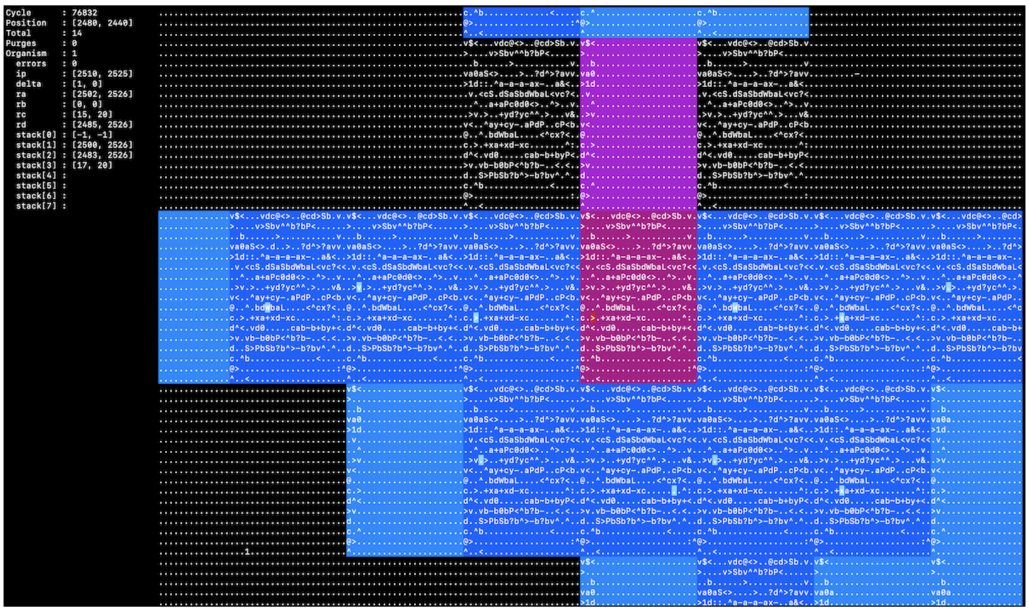
The ancestor works as follows:

- In the region **(A)**, we start executing the code from the top-left position of the organism and save the direction of the organism's replication. We determine the organism size in the region **(B)** (arrow 1). For subsequent replications, we can directly find the direction of the replication (we save the organism size into the stack) and start the replication (arrow 10). When the replication process is complete, we split the child organism and return to the initial starting position.
- In the region **(B)**, we determine the size of the organism using templates. We store the top-left position in register A (by finding the template address and subtracting [4, 3] from the template address). We store the bottom-right position in register B (by finding the template address and adding [1, 2] to the template address). By subtracting two positions, we find the organism size and store it in the register C. After that, we can determine the direction in which to allocate in the region **(C)** (arrow 2).
- In the region **(C)**, we determine the direction (vertical or horizontal) in which to allocate the child organism. The CPU stack is actively used here. Afterward, we can go to the regions that further determine the direction – left/right or up/down (arrows 3, 12).
- In the region **(D)**, we allocate the child's memory left or right. When the child's memory is allocated, we can start the replication (arrow 4).
- In the region **(E)**, we start the replication. This is the entry point to code that copies the first instruction of the column; the other instructions in the column are copied by the region **(F)** (arrow 5).
- The region **(F)** is the column replication loop. When the copy of the column is completed, we change current column in the region **(G)** (arrow 6).
- In the region **(G)**, we change the column from where to copy instructions and start copying a new column in the region **(E)** (arrow 7), or if we finished copying all columns, we exit the replication loop in the region **(H)** (arrow 8), and split the child in the region **(A)** (arrow 9).
- In the region **(J)**, we can choose to replicate up or down. Regions **(I)** and **(K)** are used as bypasses if we need to pass some regions. We use them to get to the replication loop directly from the region **(J)** (arrows 13, 14).

3.8 Implementation

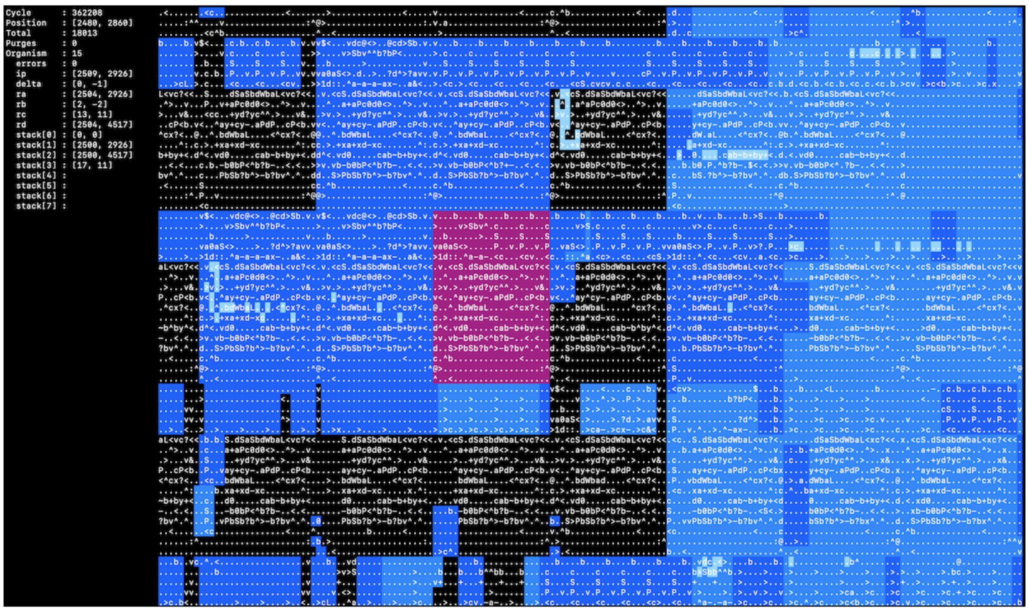
Fungera is a Python package with heavy NumPy usage for matrix-vector computation, and the TUI is curses-based. It has separate class modules for memory, organism CPU, and queue. There is also a debugger in the TUI, and it includes a mode for running with TUI detached for maximum speed. In the TUI, it is possible to view each organism's CPU elements and surf the memory. The system can save its state on-demand or at a configurable interval. The state saving is crucial for performing further analysis since the TUI becomes slower with more organisms in simulation. The true parallelism is not yet available, and the simulation only works on one core of the CPU, and thus is quite slow on later simulation stages.

In Figure 3.3, we can see the Fungeran TUI at the beginning of the simulation. The selected organism has the red notation, and its child has purple notation. Not selected organisms are in blue, and their children are in light blue. We also denote the IP address of each organism. We can also see that two organisms killed, because of their mutation produced invalid code. In Figure 3.4, we can see the Fungeran TUI in the later stages of the simulation. There are a lot of small organisms. These organisms use ancestor code to create organisms similar to themselves in size, but they do not self-replicate their code, and it rarely takes any role in this replication. The implementation of Fungera is on the GitHub repository (Poliakov, 2020).



```
Cycle       : 76832
Position    : [2480, 2440]
Total       : 14
Purges      : 0
Organism    : 1
  errors    : 0
  ip        : [2510, 2525]
  delta     : [1, 0]
  ra        : [2502, 2526]
  rb        : [0, 0]
  rc        : [15, 20]
  rd        : [2485, 2526]
  stack[0]  : [-1, -1]
  stack[1]  : [2500, 2526]
  stack[2]  : [2483, 2526]
  stack[3]  : [17, 20]
  stack[4]  :
  stack[5]  :
  stack[6]  :
  stack[7]  :
```

FIGURE 3.3: Fungeran TUI at the start of simulation (around 80000 cycles)



```
Cycle       : 362208
Position    : [2480, 2860]
Total       : 18013
Purges      : 0
Organism    : 15
  errors    : 0
  ip        : [2509, 2926]
  delta     : [0, -1]
  ra        : [2504, 2926]
  rb        : [2, -2]
  rc        : [13, 11]
  rd        : [2504, 4517]
  stack[0]  : [0, 0]
  stack[1]  : [2500, 2926]
  stack[2]  : [2500, 4517]
  stack[3]  : [17, 11]
  stack[4]  :
  stack[5]  :
  stack[6]  :
  stack[7]  :
```

FIGURE 3.4: Fungeran TUI at the later stages simulation (around 360000 cycles)

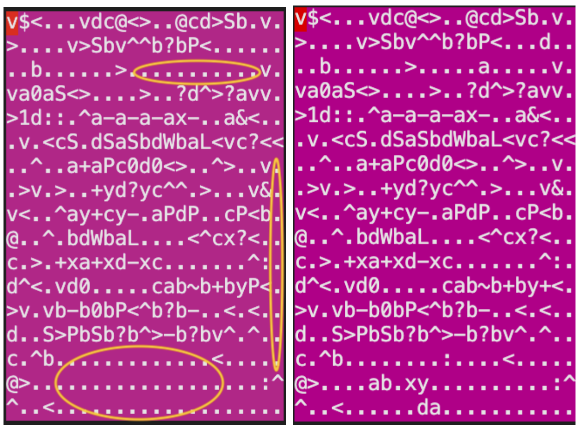
Chapter 4

Results

4.1 Micro-evolution

Over 170 different size classes emerged as a result of constant mutations on the large scale simulation (memory size [5000, 5000]), but only a few can continually reproduce and evolve. Others become extinct after one generation and do not produce any child organisms. Different behaviors in size classes themselves occur as well, but analyzing them all in detail is a complicated and time-consuming task, so it is planned for future work. We can organize the size classes that continuously evolve in Fungera in the list of notable species:

- Ancestor and its descendants with mutations in non-critical regions have the same behavior as the ancestor. Non-critical region means that the mutation in memory cells in the area does not change the overall outcome. Some non-critical regions are highlighted in yellow in Figure 4.1a. An example of an organism with non-critical mutations is in Figure 4.1b. This species always have an ancestor size [17, 23]. They can reproduce isolated in the empty memory.



```
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+byP<.
>v.vb-b0bP<^b?b-..<.<..
d..S>PbSb?b^>-b?bv^.^..
c.^b.............<.....
@>...................:^
^..<...................
```

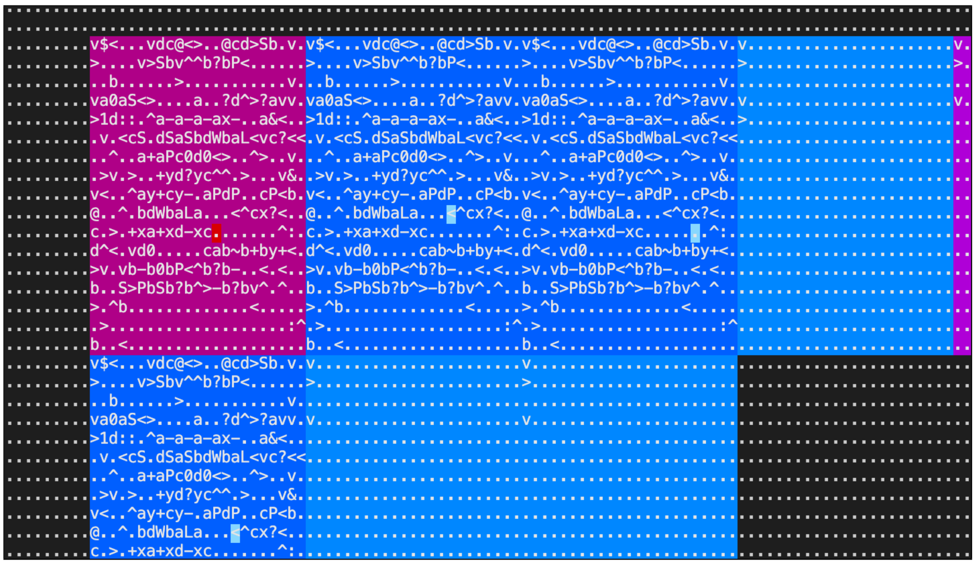
(A)

```
v$<...vdc@<>..@cd>Sb.v.
>....v>Sbv^^b?bP<...d..
..b......>.....a.....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+by+<.
>v.vb-b0bP<^b?b-..<.<..
d..S>PbSb?b^>-b?bv^.^..
c.^b........:....<.....
@>....ab.xy..........:^
^..<......da...........
```

(B)

FIGURE 4.1: Non-critical regions (A) and mutations in them (B)

- Ancestor size organisms with different behavior have mutations in critical regions of their genome. For example, this can include an organism that reproduces only right and down, but faster (Figure 4.2). They are still able to reproduce isolated in the empty memory.



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

FIGURE 4.2: The organism with mutation that allows faster reproduction

- Ancestor size organisms that produce organisms of different sizes, which, in turn, lead to the creation of new species in Fungera. While, in theory, organisms of different size classes than ancestors can reproduce isolated in empty memory, none of the organisms analyzed in detail at the time of writing were able to reproduce themselves. Typically, their descendants cannot reproduce isolated in the empty memory.
- Organisms of close size to the ancestor, typically [16, 23] or [17, 22] with the mutation that allows their IP to travel outside its allocated borders. We believe that this mutation is critical for further evolution and the emergence of parasitism. These species allocate their child block at some dead code of ancestor-like organisms and do not rewrite it, which could either lead to the creation of parasites or even to the *resurrection* of an old organism. The example of resurrection by an organism of size [16, 23] is in Figure 4.3. The same organism also produced one of the initial small size organisms in a particular simulation.

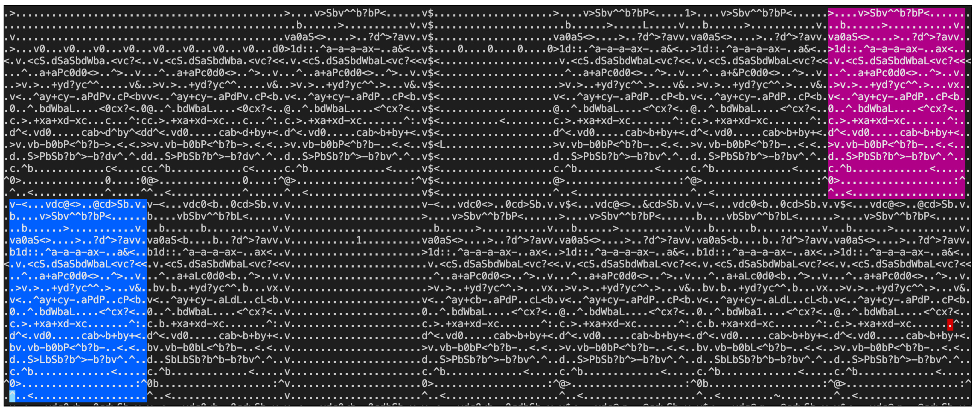


FIGURE 4.3: The [16, 23] organism allocates child memory block without rewriting it

- Currently, Fungera simulations produce a high volume of small, aberrant creatures. At first, we thought they were good candidates for parasites. But analysis shows that these organisms do not replicate their code. It is rarely executed at all. CPUs allocated for them execute ancestor code found in memory to produce similar-sized aberrant organisms. They cannot reproduce isolated. We called those objects *microvesicles*, by analogy with extracellular vesicles produced by some biological cells. Microvesicles sizes can range from [5, 5] to [1, 1]. Because of their small size, those objects multiply in number a lot faster than full-sized organisms. Depending on the configuration, they can use the dead code of other organisms or only the code of currently living organisms. Their behavior is similar to some extent to the behavior of the prions. Typically, a swarm of microvesicles of sizes [5, 5] and [1, 1] will form around the host organism, like in Figure 4.4. A microvesicle of size [1, 1] is selected on the Figure. Some small colonies of [7, 7] size also appear during the simulation's late stages.

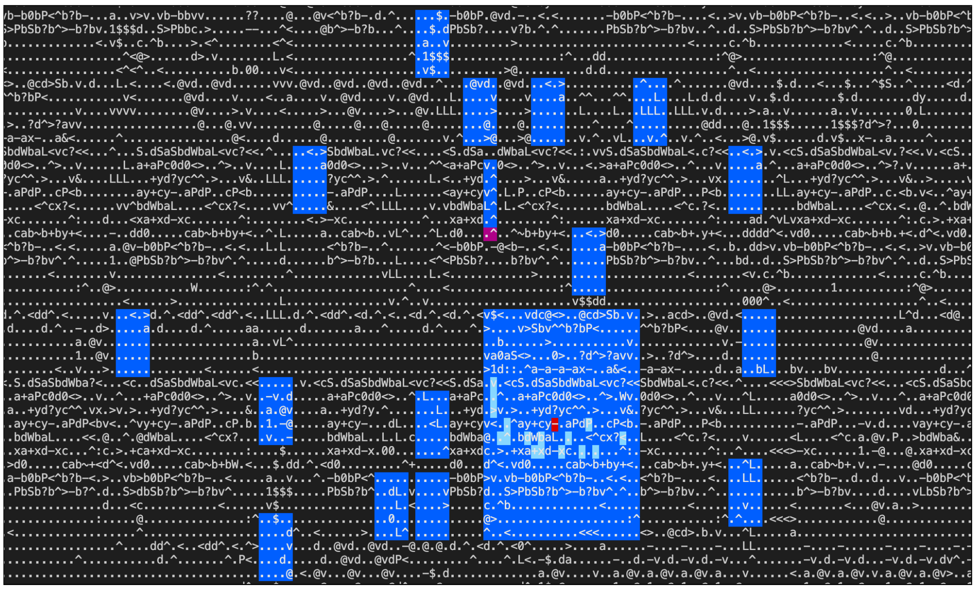


FIGURE 4.4: Swarm of microvesicles and their host organism

We plan to perform a detailed analysis of further evolution in future research since it is time-consuming. As of the time of writing, only a small part of evolved organisms were analyzed in detail. All small organisms we analized are microvesicles. Organisms with a full-sized replication loop also do not continually reproduce if they are considerably larger than the ancestor. We suspect the reason behind it might be the template searching logic, but we did not prove this hypothesis yet.

4.2 Macro-evolution

The typical large scale Fungeran simulation is running on the memory of size [5000, 5000]. It can contain more than a hundred thousand organisms with memory being 50% full. On large scale simulation, the memory does not fill up often, unlike in Tierra. The run, which we describe below is still in progress at the time of writing. We can take a look at how each simulation stage affects described organism species and size classes:

- Ancestor size organisms gradually increase in quantity, reaching the maximum count of 6000 organisms by cycle 300000, after which they decline, because they should compete for memory space with first microvesicles, and continued mutation affects the number of healthy descendants. They decline to 1000 organisms by cycle 500000, and after that, they increase their number to 2500 by cycle 575000. They again decline afterward. However, they are not affected as much as other organisms during the first killing due to memory overflow. Their number is around 1000 after the first memory overflow at cycle 584700 (orange line), and they rise to 2000 afterward. (Figure 4.5).

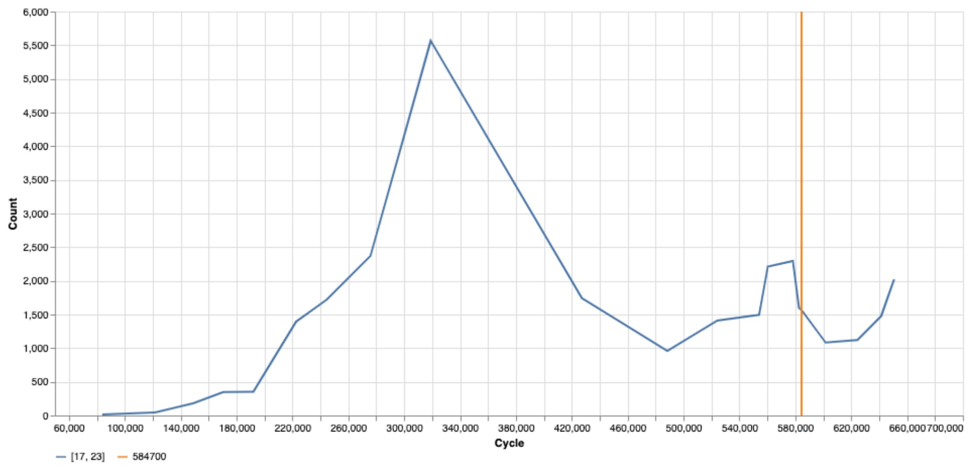


FIGURE 4.5: [17, 23] organisms in the large scale simulation

- Organisms of close size to the ancestor do not have significant numbers, compared to the ancestor sized organisms or microvesicles. [17, 22] organisms are a first different size from the ancestor class to emerge at around cycle 100000. Interestingly, its behavior mimics ancestor size behavior, but numbers are a lot smaller. [16, 23] organisms are not numerous as well but can play a crucial role in the simulation, by creating microvesicles and other aberrant forms (Figure 4.6).

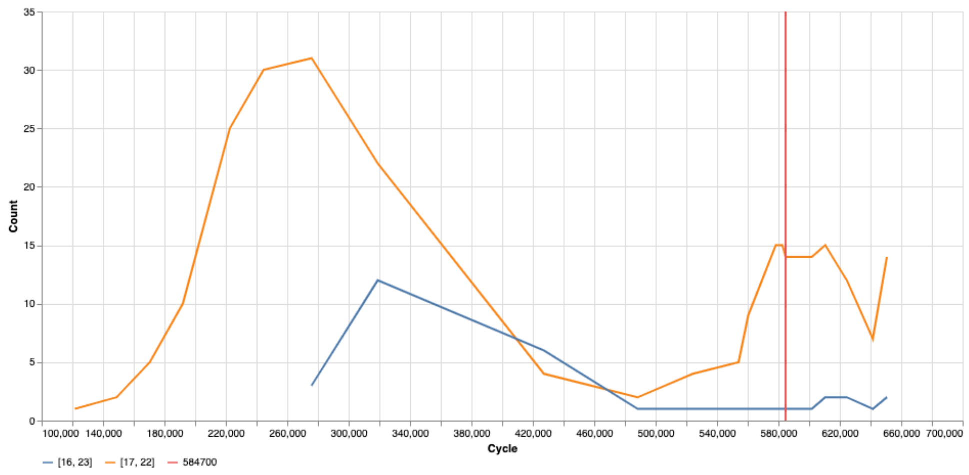


FIGURE 4.6: [16, 23] & [17, 22] organisms in the large scale simulation

- Microvesicles have various sizes, but in the simulation, microvesicles of size [5, 5] tend to be dominating species. They start appearing at cycle 430000 and rapidly increase their numbers to around 135000 by cycle 580000. They cannot survive without a living host, and we can see they start declining when ancestor sized organisms numbers decreasing as well. The memory overflow killings affect them the most, cutting the population in half at cycle 584700 (red line). Another widespread microvesicle size is [1, 2]. It is older than [5, 5] (appears around cycle 300000), but the maximum number of them is 10000, and they are second-most-populous organism size (Figure 4.7). Other sizes, for example, [4, 4], do not achieve the same numbers as [5, 5] or [1, 2] microvesicles. Microvesicles size can also go to [1, 1], a single memory cell pseudo-organism, which appears later than other microvesicle types (Figure 4.8). Memory overflow killings affected microvesicles the most since they are numerous and tend to produce many errors. However, the [5, 5] microvesicles start to grow again fast afterward, partially recovering their pick numbers at cycle 650000.

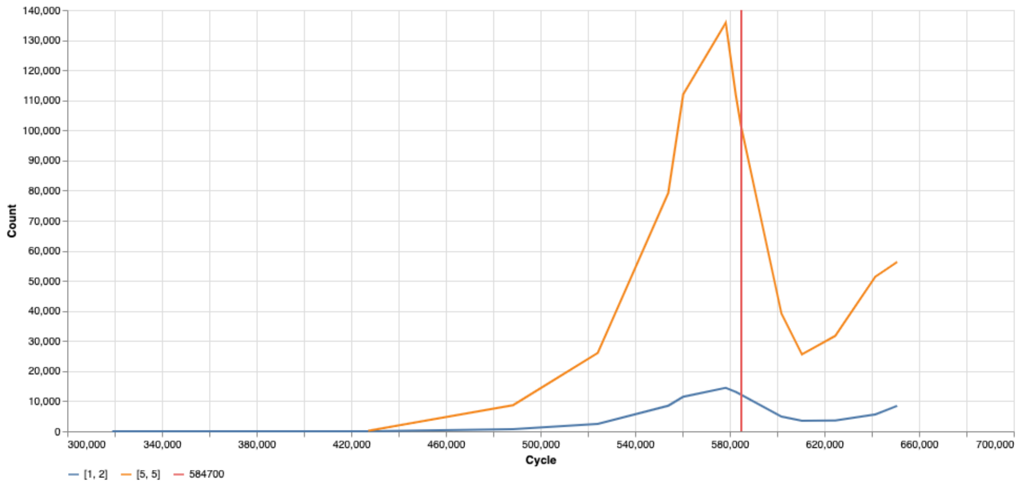


FIGURE 4.7: [5, 5] and [1, 2] microvesicles in the large scale simulation

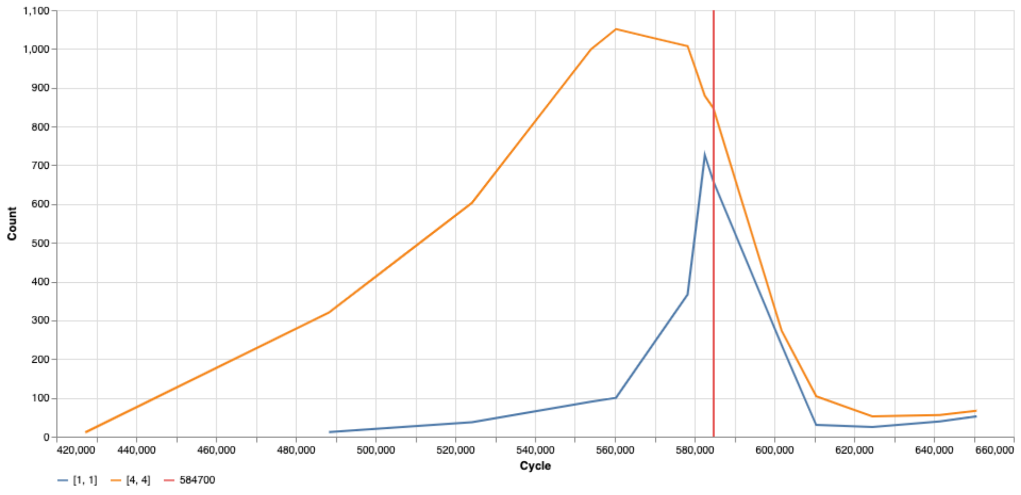


FIGURE 4.8: [4, 4] and [1, 1] microvesicles in the large scale simulation

In table 4.1, we can observe top size classes by organism count. This table includes all organisms that were created at the simulation overall by cycle 600000. The share of [5, 5] microvesicles is 85% and microvesicles overall more than 96%, with only 4% being ancestor size or may have own replication loops. This behavior is a clear sign of evolutionary optimization (though undesirable for our simulations) since microvesicles need only a fraction of resources (often less than 1000 cycles) to create new similar-sized objects compared to ancestor size organisms (18000 cycles).

Size class	Count	Share
[5, 5]	743858	85.54%
[1, 2]	68935	7.92%
[17, 23]	29458	3.38%
[4, 4]	10617	1.22%
[5, 4]	3288	0.37%
[1, 1]	2810	0.32%
[2, 3]	1878	0.21%
[3, 4]	1252	0.12%

TABLE 4.1: Top 10 size classes by organism count in Fungera

We save each organism that was born in separate storage with the identification of its parent organism so that we can identify the ancestors for each organism up until the first organism in the simulation itself. In the Figure 4.9 we can see the simplified evolution tree for the simulation. At cycle 120000, the first organism of different size appears (size [17, 22]). At cycle 300000, there are multiple organisms close to ancestor size, for example [16, 23], [16, 21], and [16, 22]. At that time, there also appears the first microvesicle [1, 2], descendant from (produced by) [17, 23]. Later, the [5, 5] microvesicle appears from [16, 23] size class. Though [5, 5] microvesicles are mostly created by other [5, 5] microvesicles on their own[1], most of them are descendants of first [5, 5] microvesicles created by one particular organism of size [16, 23]. It is possible that the organism's surrounding in the memory was favorable for fast creation of those microvesicles. At later stages, microvesicles of various sizes appear from both [5, 5] and [1, 2] size classes. It is also common for microvesicles to change size classes of descendants in turn (for example [5, 5] → [5, 5] → [4, 4] → [3, 3] → [5, 5]). In Fungera, the model for evolution is the punctuated equilibrium since we have a very sharp change from [16, 23] to [5, 5] or from [17, 23] to [1, 2]. After these changes, there are long periods of stability in the population.

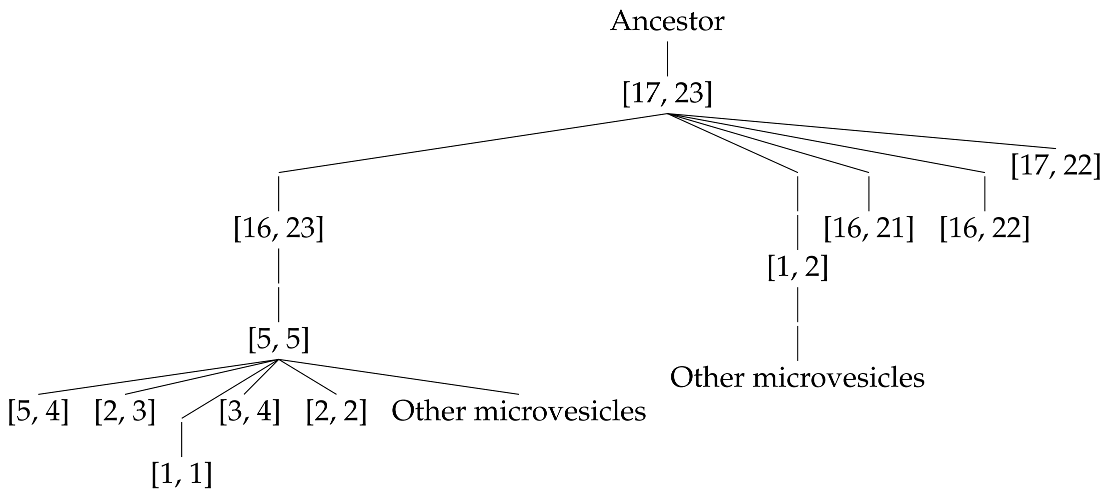


FIGURE 4.9: Simplified evolution tree for the simulation

[1]By using code found nearby in memory. Let us reiterate that their own code rarely takes any part in this reproduction.

4.3 Complexity

The large [5000, 5000] simulation, which is still in progress during the time of writing, produced overall more than 1200000 organisms by cycle 650000. Compared to standard Tierra simulation, in which the memory length 60000, described Fungeran simulation has 416 times more cells. The run on the core of the 9th generation Intel Core i7 CPU took more than 72 hours to reach cycle 650000. In Figure 4.10, we can see how the simulation can do 8000 cycles per second on the start, and it decays to only 2.5 cycles per second on the later stages of the simulation. The number of cycles per second does not depend on organisms count (Figure 4.11). It means that more massive simulations might not be possible, without utilizing true parallelism in the simulation.

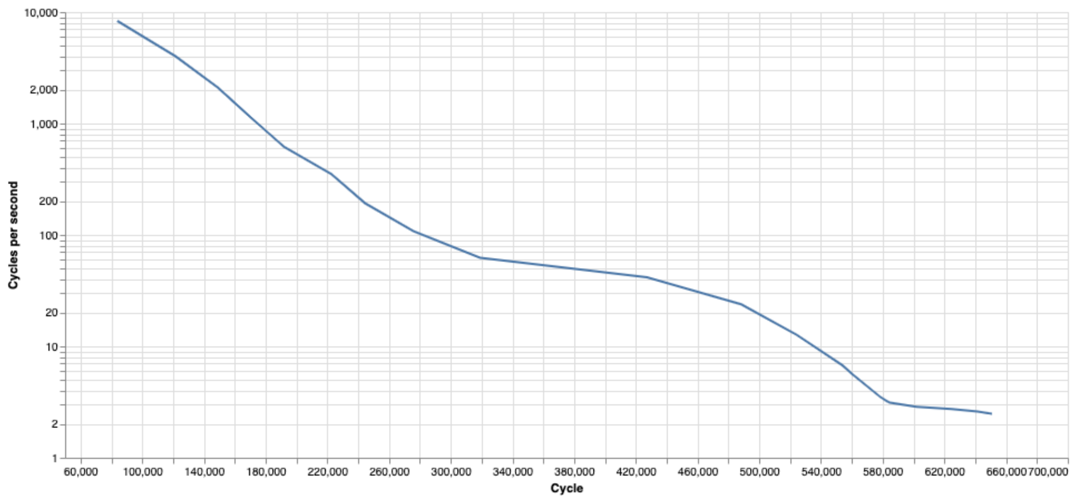


FIGURE 4.10: Cycles per second in the simulation

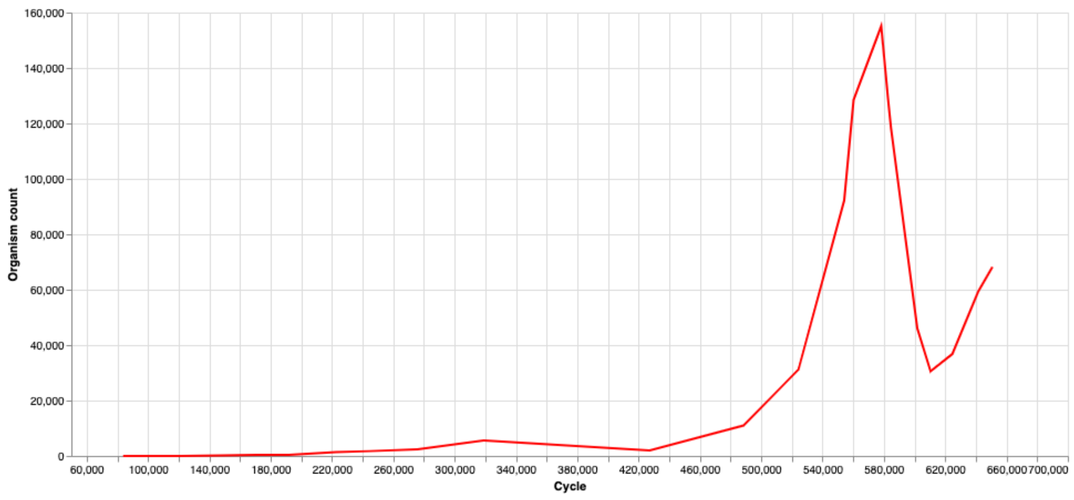


FIGURE 4.11: Total organism count in the simulation

Chapter 5

Conclusion

5.1 The behavior of the simulation

The two-dimensional evolution simulator is capable of showing different aspects of evolution. Over 170 different size classes emerged as a result of constant mutations on the large scale simulation, and more than 1200000 organisms were created by cycle 650000. The evolution process is similar to the punctuated equilibrium model. Different types of species appeared, including various types of aberrant forms we called microvesicles with some signs of collective behavior. The simulation is still running by the time of writing, and it bottlenecks reaching cycle 700000. We need to introduce true parallelism and optimizations in order to continue the research further. Simulations in small memory sizes are planned too in hope to spur faster evolution of non-trivial new forms.

5.2 Further work

While we showed that evolution in two-dimensional memory space is possible, there is still much work ahead to make it open-ended. Some of the steps for further research include:

- Analyzing organisms for different behaviors within the same size class and search for possible parasitism, symbiosis, or even sociality/multicellularity. Further research all behaviors of different species.
- Rewriting the core backend in a compiled language like C++. Python, even with NumPy, does not provide fast enough processing speed for such a task.
- Conversion to C++ should make creating true parallelism easier. We could not achieve it with Python, because of its global interpreter lock (GIL). Still, there will be many issues with speed and consistency agreement. Also, the results would be harder to reproduce.
- Creating a separate GUI package with Qt, the backend should serve only as API for GUI.
- The world rules might need some adjustments to make it as natural as possible, but not complicated at the same time.

5.3 Afterword

After the thesis was already written, we discovered a paper that implements a simulation with a very similar idea about the memory by Florent de Dinechin in 1997. The simulation is called Ziemia (Earth in Polish). The memory structure is a two-dimensional grid, the same as we present in Fungera (De Dinechin, 1997, p. 2). However, there are a few notable differences between Ziemia and Fungera. There is no template matching in Ziemia, and it is replaced entirely by physical loops (same as we presented in Fungera). The difference is that we still use templates to denote the start and the end of the organism. While organisms in Ziemia are running in a two-dimensional memory grid, they are still sequential (De Dinechin, 1997, p. 3). Each cell of the organism stores a pointer to the next cell. The ancestor is much more straightforward compared to Fungera (50 instructions compared to 17 * 23 = 391, respectively). Because of this, it is relatively easy to break the sequential ancestor with mutations in a two-dimensional memory. The organism's memory space is private because there is no clear way to define the organism's location. Therefore, no parasitism is possible. The descendants of ancestors quickly die, and the simulation halts (De Dinechin, 1997, p. 5). Another work that could be related is a master's thesis called *A model of early evolution in two dimensions* by Carlo Maley (1993). We could not find the digital copy of this work, and we did not receive an answer to the email regarding access to the work from the author.

It seems that one of the main reasons why microvesicles (described in the thesis) are appearing, is that the template matching range is limited to the maximum value in organism size vector. When the microvesicle is created, its IP travels outside of its boundaries. There is a chance that it will go to the start of some ancestor-like organism. So, these microvesicles rely entirely on the ancestor-like code because its IP behaves precisely like an ancestor's IP normally would. Because we limited the template matching range, we incidentally leaked information about the organism size available only to the simulator to the organism. The evolution efficiently found and exploited this bug similar to Meltdown bug of modern microprocessors. The microvesicles have a small size, and in the region B of the ancestor (Figure 3.2), the microvesicles would not find the template it needs and register A will stay [0, 0] (because of the limitation). After subtraction, register A will become [-4, -3]. In analogy, register B will become [1, 2], and the size that the microvesicle will self-examine is [1, 2] - [-4, -3] = [5, 5]. Because of this small self-examined size, it will exit the reproduction loop faster and reproduce faster as well. Other sizes behave the same (including microvesicles of size [1, 1]). Because of the organism size leakage problem and the fact that microvesicles do not use any of its code, this behavior can be considered as a bug, and we should tackle this issue in future work.

Bibliography

Langton, C. (1989). Artificial Life: Proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems. Addison-Wesley Longman Publishing. https://archive.org/details/artificiallifepr00inte

Mazlish, B. (1995). The man-machine and artificial intelligence. *Stanford Humanities Review*, *4*(2). http://www2.psych.utoronto.ca/users/reingold/courses/ai/cache/mazlish.html

Sharkey, N. (2007). *A 13th Century Programmable Robot*. https://web.archive.org/web/20070629182810/http://www.shef.ac.uk/marcoms/eview/articles58/robot.html

Panse, S. (2019). *Leonardo's Robot: Leonardo da Vinci's Mechanical Knight and Other Robots*. https://www.ststworld.com/leonardos-robot

Aguilar, W., Froese, T., & Gershenson, C. (2014). The past, present, and future of artificial life. *Frontiers in Robotics and AI*. https://www.frontiersin.org/articles/10.3389/frobt.2014.00008

Wolfram, S. (2002). A new kind of science. Wolfram Media. https://archive.org/details/newkindofscience00wolf

Marinescu, D. (2017). Nature-Inspired Algorithms and Systems. *Science Direct*. https://www.sciencedirect.com/topics/computer-science/cellular-automata

Pesavento, U. (1995). An implementation of von neumann's self-reproducing machine. https://web.archive.org/web/20070621164824/http://dragonfly.tam.cornell.edu/~pesavent/pesavento_self_reproducing_machine.pdf

Bettilyon, T. (2018). How i optimized conway's game of life. https://medium.com/tebs-lab/optimizing-conways-game-of-life-12f1b7f2f54c

Wilson, R. (2001). The MIT Encyclopedia of the Cognitive Sciences. The MIT Press. https://books.google.com.ua/books?id=-wt1aZrGXLYC

Pfeifer, R. (2001). Artificial Life. University of Zurich Department of Informatics. https://www.ais.uni-bonn.de/SS09/skript_artif_life_pfeifer_unizh.pdf

Adami, C., & Brown, T. (1994). Evolutionary Learning in the 2D Artificial Life System Avida. https://arxiv.org/pdf/adap-org/9405003.pdf

Pargellis, A. (2001). Digital Life Behavior in the Amoeba World. https://www.researchgate.net/publication/11879815_Digital_Life_Behavior_in_the_Amoeba_World

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

Hickinbotham, S., & Stepney, S. (2015). Environmental bias forces parasitism in Tierra. https://www.mitpressjournals.org/doi/pdf/10.1162/978-0-262-33027-5-ch055

Ray, T. *Tierra Photoessay*. https://web.mat.upc.edu/francesc.comellas/old-files/buran/Tierra_Photoessay.html

Jensen, R. (2015). *Punctuated Equilibrium*. https://www.artofreasoning.com/?p=655

Ofria, C. (2003). Avida: A software platform for research in computational evolutionary biology. http://www.cs.cas.cz/~petra/EA/AvidaIntro-ALife.pdf

Pollac, J., & Bedau, M. (2004). Artificial Life IX: Proceedings of the Ninth International Conference on the Simulation and Synthesis of Living Systems. The MIT Press. https://books.google.com.ua/books?id=cMFQ3qfzEW4C

Pargellis, A. (2003). Self-organizing Genetic Codes and the Emergence of Digital Life. https://www.researchgate.net/publication/220657508_Self-organizing_genetic_codes_and_the_emergence_of_digital_life

Greenbaum, B., & Pargellis, A. (2016). Digital Replicators Emerge from a Self-Organizing Prebiotic World. https://www.mitpressjournals.org/doi/abs/10.1162/978-0-262-33936-0-ch016

Standish, R. (2008). Open-ended artificial evolution. https://arxiv.org/pdf/nlin/0210027.pdf

Sugiura, K. (2003). Evolution of Rewriting Rule Sets Using String-Based Tierra. https://www.researchgate.net/publication/221531303_Evolution_of_Rewriting_Rule_Sets_Using_String-Based_Tierra

Ray, T. (1993). An evolutionary approach to synthetic biology: Zen and the art of creating life. http://www.sci.brooklyn.cuny.edu/~sklar/teaching/f05/alife/papers/ray-zen.pdf

Esolang, the esoteric programming languages wiki. (2020). Befunge. https://esolangs.org/wiki/Befunge

Oliver, P. (2019). Salis 2.0. https://github.com/PaulTOliver/salis-2.0

Poliakov, M. (2020). Fungera. https://github.com/mxpoliakov/fungera

De Dinechin, F. (1997). Self-replication in a 2d von neumann architecture. https://pdfs.semanticscholar.org/646a/c824275a688228dc06d2144e25b7b9b00b97.pdf