Studying the Evolvability of Self-Encoding Genotype-Phenotype Maps

Andrew M. Webb and Joshua Knowles

School of Computer Science, University of Manchester, UK
andrew.webb@manchester.ac.uk

Abstract

We introduce a model of reproduction in which the genotype-phenotype (G-P) map is able to evolve. In this model, Each organism implements a G-P map, determining how the organism is encoded in its genome. Crucially, it also determines how the G-P map itself is encoded. We call these maps 'self-encoding'. We relate this model to recent artificial life research, and back to the seminal work of John von Neumann. We simulate populations of organisms that have as their genome and G-P map the axiom and production rules of an L-system. The populations are given the task of optimizing a dynamic fitness function. Our purpose is to study whether the self-encoding property has any effect on the evolution of evolvability, and to look for other factors that lead to the evolution of G-P maps that confer evolvability. We find that evolvability does evolve, but only when we add constraints to the model.

Introduction

Our principal reason for studying the genotype-phenotype (G-P) map is its role in determining evolvability. We use Hansen's definition (Hansen, 2006):

> "Evolvability is the ability of the genetic system to produce and maintain potentially adaptive genetic variants."

Higher evolvability means a greater probability of introducing adaptive variants. It is a variational property, related to the way that variations are introduced into the phenome.

Mutations in the genome are more or less random[1]. The genome is an encoded form of the phenome, with the G-P map implementing the decoding function. It can potentially translate the random mutations in genome into directed changes in the phenome (Wagner and Altenberg, 1996; Hansen, 2006). For example, the G-P map might perform error correction when determining one attribute of the phenome, making it less variable than other attributes. Or it might use a single gene to determine two attributes, ensuring that the two can only vary together. We will say that the G-P map *mediates* variation in the phenome, in the sense that it acts as a mediator between the random source of variation and variation that actually occurs.

A G-P map increases evolvability if it translates random mutations into suitably directed (i.e., sometimes adaptive) changes. Life on Earth is believed to have evolved towards higher evolvability through changes to the G-P map (Dawkins, 2003; Pigliucci, 2008).

When and how does evolvability evolve? In what kinds of environment does it happen? How can beneficial G-P maps be selected for when they only confer a future advantage and when selection can only act on current fitness? Recent work in artificial life has aimed to answer such questions by studying model organisms in which the G-P map can evolve. We introduce our own model of reproduction, in which the G-P map is 'self-encoding'; it determines how the G-P map itself is encoded in the genome.

The purpose of our research is to learn what factors lead to the evolution of G-P maps that confer greater evolvability. Our contributions are as follows.

- We describe and compare recent artificial life research in which model organisms within the Avida platform, or an extension of it, are studied with the purpose of understanding the evolution of the G-P map.
- We suggest potential problems with these models, and with using Avida to study the evolution of the G-P map in general.
- We introduce a model of self-reproduction in which organisms implement a 'self-encoding' G-P map.
- We simulate populations of these organisms, using L-system production rules as their G-P map, that evolve to optimize a dynamic fitness function. We find that evolvability fails to evolve unless we add constraints to the model.

Context and Related Work

In this section, we describe two models of self-reproduction used in recent artificial life research to study the evolution

[1] There are conflicting definitions of 'genotype'. Here we use 'genome' to refer to the total genetic information of an organism, and 'phenome' to refer to the set of all traits of an organism. We use the established term 'genotype-phenotype map' to refer to the mapping between them.

of the G-P map. Since both models are based on the Avida platform, we also describe the default reproduction mechanism in Avida. For comparison, we describe the reproduction mechanism of John von Neumann's *universal constructor*.

Von Neumann's Universal Constructor

John von Neumann developed a two-dimensional, 29-state cellular automaton, and a structure within it that he called the *universal constructor* (Von Neumann and Burks, 1966). The universal constructor has, roughly, the following components.

- A one-dimensional 'tape' **G**, which in inactive; on its own, it doesn't do anything.
- An active 'machine' **P**, which consists of the following four components.
  - A 'decoder' **A**, which implements a decoding function. It reads the contents of the tape **G**, decodes the contents into the specification of a two-dimensional structure, and then builds that structure elsewhere in the cellular automaton.
  - The 'tape copier' **B**, which builds a copy of the tape **G** adjacent to the structure built by the decoder.
  - The 'coordinator' **C**, which coordinates the actions of the decoder and tape copier.
  - The 'ancillary machine' **D**, which can perform any arbitrary action as long as it doesn't interfere with the other components.

If the tape **G** contains an encoded description of **P**, relative to the decoding function implemented by **A**, then the pair constitute a self-reproducing machine; the part **B** constructs a copy of **G**, and the part **A** decodes **G** to construct **P**. In order to be a self-reproducer, the machine has to do two things. First, it has to implement a mechanism for copying the tape. Second, it has to implement a decoding mechanism that decodes the tape into a description of the tape-copying and decoding mechanisms.

This is a convincing model of biological reproduction that includes a G-P map. The analogy to life is as follows. **G** is the genome of the organism. **P** is the phenome. **A** is the G-P map, encompassing the genetic code and development processes. **B** is the mechanism for copying the genome. **D** is any behaviour of the organism not directly related to reproduction.

Mutations in **G** will manifest as a change in **P** in the offspring. Von Neumann noted that if the mutation affects only the ancillary machine, then the offspring will still be a self-reproducer. He believed that any mutation affecting the decoder would render the offspring infertile, as the decoding function it implements would no longer complement the encoding of **G**. A change to the decoder that leaves the offspring fertile corresponds to a change in the G-P map.

When **G** is mutated, how this manifests as a change in **P** of the offspring is mediated by the part **A** implemented by the parent. This includes changes *to* part **A**; it mediates all variations in the offspring. Barry McMullin has been a proponent of studying this model of reproduction (McMullin, 2012).

Avida

Avida is a popular software platform for studying self-replication (Ofria and Wilke, 2004), in which each organism is a program in an assembly-like language. Avida has a cellular structure; each organism has its own private memory, and the organisms are arranged in a grid. The population of Avidans execute their instructions in parallel.

Avida simulations are initialized with a single, hand-designed self-replicating program called the *ancestor*. The default ancestor replicates by setting up and iterating over a copy loop, in which the 'copy' instruction is repeatedly executed to read from and write to successive memory locations. Once it has copied the entire contents of its memory, it executes the special *divide* instruction. This places a new child Avidan in a neighbouring cell, giving it the recently copied instructions.

Avidans can perform additional tasks as well as self-replication, and the experimenter can adjust the 'metabolic rate' of an organism (the rate at which it executes instructions) based on its performance on these additional tasks.

There are multiple types of mutation in Avida. The 'copy' instruction has a low probability of writing the wrong instruction, and 'cosmic ray' mutations can change any instruction in memory at any time.

An Avidan's sequence of instructions, being the hereditary information that is transmitted between generations, is its genome. The phenome of an Avidan is its behaviour when run. Since the genome determines the phenome, the G-P map is fixed.

As McMullin notes, in von Neumann's terminology, the **G** and **P** parts of an Avidan are the same, and consist only of components **B** and **D**. Avidans don't reproduce by applying a decoding function to an encoded description of themselves; they directly read and copy the contents of their own memory.

Implementing the 'von Neumann architecture' in Avida

Hasegawa and McMullin study a self-reproducing ancestor program based on von Neumann's model of reproduction (Hasegawa and McMullin, 2012). One part of the program, identified with **G**, is not executed but is read as data. The other part, identified with **P**, is executed, and does the following.

1. Enter a copy loop, identified with the tape copier **B**. Unlike the copy loop of the default Avidan ancestor, which copies the entire contents of memory, it only copies part **G**.
2. Apply a decoding function, identified with the decoder **A**, to **G**, and write the result to the region of memory adjacent

to the copy of **G**.

3. Execute the divide instruction.

During this process, mutations can happen in any part of the memory. Like von Neumann's constructor, Hasegawa and McMullin's ancestor has the property that when the part **G** is mutated, the change observed in **P** of the offspring is mediated by the decoding function implemented by the parent. However, it is possible for 'mutations' to happen directly in the part **P**. These mutations will cause changes to **P** that are unmediated by the decoding function.

One of their results is that populations quickly reverted to the self-inspection form of reproduction more commonly seen in Avida, without a decoding step.

Evolving the 'hardware' in Avida

Egri-Nagy and Nehaniv implemented a variant of Avida, in which each organism has its own set of data structures and its own high-level programming language, with instructions composed of Avida's lower-level instructions (Egri-Nagy and Nehaniv, 2003). When an organism is born, the first part of its memory is read as a specification of that organism's data structures and instruction set (its 'hardware'). The organism then starts executing instructions as normal.

They refer to the hardware specification as the 'G-P map', because it changes the result of executing the program (the 'genome'). However, unlike in Hasegawa and McMullin's work, the genome doesn't contain an encoded form of the G-P map ; the mechanism of reproduction, like that of the default Avida ancestor, is to enter a copy loop that copies the entire contents of memory to the offspring, introducing copy errors.

The part labelled the 'G-P map' mediates the changes in the behaviour of the organism as a result of mutations in the genome. However, the G-P map itself is subject to random mutations, and the encoding of this part of the organism is fixed; variation in the G-P map is unmediated.

General Limitations

We suggest that Avida may not be an ideal platform for studying the evolution of G-P maps for the following reasons.

- Because it was designed to allow reproduction by self-inspection and copying, without a decoding step, this method of reproduction will inevitably be the most efficient in Avida. One of the results of Egri-Nagy and McMullin's work is that simulations initialized with their ancestor quickly become dominated by 'self-inspection' reproducers.

- The assembly-like programming language used by Avida isn't designed for easily expressing developmental processes.

- Two of the types of mutation in Avida can act on any location in memory; for any Avidan that has parts equivalent to **G** and **P**, some mutations will happen directly in **P**, unmediated by the decoder.

Self-Encoding Genotype-Phenotype Maps

Here we introduce a model of reproduction in which each organism implements a G-P map, which determines how the organism is encoded in its genome. Moreover, it determines how the G-P map *itself* is encoded in the genome. For brevity, in the following we use 'genotype-phenotype map' and 'decoder' synonymously. We call the G-P maps 'self-encoding', and call the organisms 'self encoders'. The model differs from previous work in the following ways.

1. The organisms do not implement a mechanism for copying their genome. In von Neumann's terminology, our organisms are missing part **B**. The reason is that we want to focus on part **A**, the decoder, and we know precisely what the behaviour of part **B** should be. The genome-copying step is built into the model.

2. We initialize the population with randomly generated organisms, rather than hand-designed self-reproducing organisms, in order to avoid accidentally initializing the population in an evolutionary dead-end.

3. Only the genome is mutated. All variation in the phenome is mediated by the decoder, including variation in the decoder itself.

4. Rather than an assembly programming language, our model will use decoding mechanisms that are biologically inspired.

With the copying of the genome built into the model, and the initial decoder population initialized randomly, the initial state is analogous to a population of self-replicating RNA. The random initial 'decoders' are like active molecules that interact with the RNA to produce other molecules. The initial decoders are not self-reproducing; the emergence of self-reproducing decoders corresponds to the emergence of a developmental step on top of an existing replication mechanism.

Each organism in the model consists of a genome and a phenome. The phenome consists of a decoder, and another part that we will apply a fitness function to and adjust the metabolism of the organism based on the result. We call this part the *solution*. See Figure 1.

The population lives on a discrete grid, and is updated iteratively, in parallel, by looping over the grid. Algorithm 1 shows one such iteration. Fitter organisms are rewarded by 'stealing' update cycles from less fit neighbours. When an organism creates a new child, the child receives a mutated copy of the parent's genome. This genome becomes the starting point for the 'embryo', which is a working copy of the

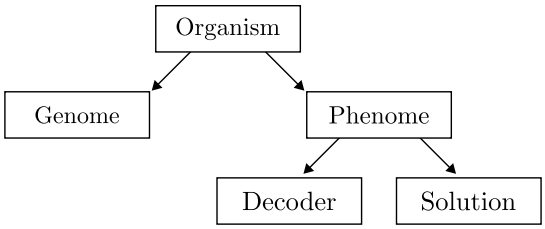


Figure 1: The structure of an organism in our model. Arrows indicate a 'contains' relationship.

data operated on by the parent's decoder. Each time a parent organism updates, it goes through one decoding step, applying its decoder to the embryo of its child. Later in the paper we give an example of a decoder being applied iteratively. When decoding is complete, the embryo is interpreted as the specification of a decoder and a solution. These are given to the child, and the solution is used to set the child's fitness. The child is then placed somewhere in the neighbourhood of the parent.

Algorithm 1 One time step of the simulation.

```
for each cell in grid do
    organism p ← fitter of organism in cell and organism randomly selected from 3 × 3 neighbourhood
    if p has no embryonic child then
        p.child.genome ← mutated_copy(p.genome)
        p.child.embryo ← p.child.genome
    p.child.embryo ← p.decoder(p.child.embryo)
    if decoding is complete then
        (d, s) ← interpret(p.child.embryo)
        p.child.decoder ← d
        p.child.solution ← s
        p.child.fitness ← fitness_function(s)
        place child on grid in 3 × 3 neighbourhood of p
```

Selection is partly implicit, partly explicit; there is an implicit selection pressure to be an efficient and robust reproducer, and we explicitly apply selection pressures by rewarding organisms for the contents of their solutions.

The Decoding Mechanism

We use L-system production rules as the decoding mechanism. L-systems were first used to model plant development (Lindenmayer, 1968), but have since become popular as a more general model of development. An L-system consists of an alphabet, an axiom, which is a string in that alphabet, and a set of production rules. Each production rule specifies how to rewrite a substring to another substring. L-systems grow strings iteratively, starting from the axiom, by applying as many production rules as possible in parallel.

For example, an L-system might have the axiom

A

and the production rules

A → BA,
B → A

The first few strings generated by this L-system are

A
BA
ABA
BAABA
ABABAABA

The L-systems we use are context sensitive (Prusinkiewicz et al., 1990), so the left-hand side of each production rule contains three symbols: the left context, the symbol to rewrite, and the right context. For example, the production rule

A<B>C→DEF

replaces the symbol B, if it has an A on the left and a C on the right, with the symbols DEF. We use context-sensitive L-systems because they allow recursive application of production rules, while also making it possible for the process of applying rules to terminate rather than recursing indefinitely.

Each organism has a set of production rules as its decoder, and its genome is used as the axiom of the L-system. Every L-system has the same alphabet of 10 symbols. The axiom of a parent is copied to its child and mutated. Starting with the child's axiom, the parent's production rules iteratively grow a string. One decoding step consists of applying as many production rules as possible in parallel.

When there are no more rules to apply, decoding is complete. The resulting string specifies both the production rules and the solution of the child. The symbol sequence AA is a punctuation mark separating the two parts. For example, the organism with the production rule

A<B>A → DEFAADEF

operating on the axiom

ABA

will produce the string

ADEFAADEFA

which the AA punctuation separates into the strings

ADEF,
DEFA

with the first string specifying the production rules of the offspring, and the second string specifying the solution.

Since all production rules have three symbols on the left-hand-side and an arbitrary number of symbols on the right-hand-side, the production rules are specified in the string as follows. The first three symbols specify the left-hand-side of a rule. Now there is another kind of punctuation: each of the sequences {BB, BC, BD, CB, CC, CD, DB, DC, DD} indicate the end of the current rule; the right-hand-side of the rule consists of all of the symbols up to, but not including, the next such punctuation. The next rule is then built from the symbols following that punctuation. For example, the string

DEFDEFBCGHIGHI

contains the punctuation sequence BC, and so specifies the two production rules

$$\texttt{D<E>F} \rightarrow \texttt{DEF},$$
$$\texttt{G<H>I} \rightarrow \texttt{GHI}$$

The use of sequences of symbols as punctuation precludes their use as data; it is impossible to build a rule that explicitly rewrites to any substring containing a punctuation sequence, though it can be done indirectly.

The Fitness Function

We adjust the metabolism of the organisms based on the result of applying a fitness function to their solutions. Since L-systems are well suited to growing self-similar patterns, we use a fitness function that rewards strings with extensive repetition, in the hope that decoders well suited to the problem will emerge. The fitness function counts the number of occurrences of the symbol E in the solution, up to a maximum of 150.

Since we are looking for the evolution of evolvability, we use a fitness function that changes periodically in a structured way; if the fitness function was static, then once the population performed perfectly there would be no meaningful sense in which it could be evolvable or not. After a while we switch to rewarding for occurrences of the F symbol, then G, H, I, and back to E.

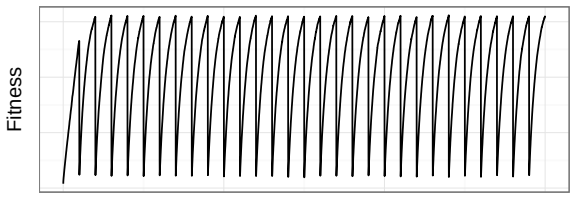
Simulations and Results

We ran simulations in order to find factors leading to the evolution of evolvability. We were also looking for the emergence of stable, self-reproducing decoders: organisms whose production rules, applied to their own axiom string, yield a description of themselves.

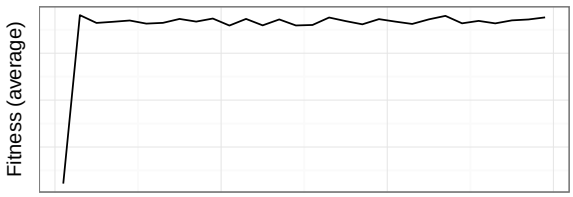
Since the fitness function changes periodically, the average fitness of a population over time will look like Figure 2a, increasing until the change point and then dropping to almost zero. We switch the fitness function every 100,000 fitness function evaluations. We measure the average fitness within each window of 100,000 evaluations, giving a single number measuring how effective the population was at maximizing the fitness function within that window.

Figure 2b shows an example of the window-averaged fitness over time. Within a given window, the height of this graph gives an indication of the evolvability of the population with respect to the fitness function in that window. A positive slope would indicate the evolution of evolvability.

We compared the performance of a population of self encoders, using the context-sensitive L-system decoding mechanism, with two other methods of optimizing the fitness function. One is a simple genetic algorithm, in which there are no production rules; the genome string directly represents a solution to the fitness function. We use the genetic algorithm



(a) An example of average population fitness over time.



(b) An example of the fitness over time, averaged over each fitness function window. Within each window of 100,000 evaluations we use this single number as an indication of evolvability.

as a baseline against which we compare the performance of the other methods.

The other method used for comparison does have a set of production rules, but these are used to construct only a solution, rather than a solution and the production rules of the next generation; at each reproduction step, the genome string is copied to the offspring and mutated, the parent's production rules are applied to the genome string, and the result specifies the solution. The parent's production rules are then also copied to the offspring and mutated. We call this the *solution encoder*, since the organisms control only how they encode the solution, and the encoding of the decoder is fixed. The purpose of comparing against the solution encoder is to determine whether the self-encoding property has an effect on the evolution of evolvability.

Each simulation ran for 3,000,000 function evaluations/births. We ran each simulation forty times for each method. In the fitness plots in the following sections, the solid or dashed line shows the median fitness amongst these runs over time, and the surrounding shaded region lies between the 25th and 75th percentiles.

Simulation 1

In the first simulation, we compared a population of self encoders, solution encoders, and the genetic algorithm, each with a population size of 20×20. Figures 3 and 4 show the results.

The solution encoder performs well. The performance profile of the self encoder population is similar to, if a little worse than, that of the genetic algorithm. The reason is that the population is quickly dominated by organisms with empty sets of production rules. The following are some example

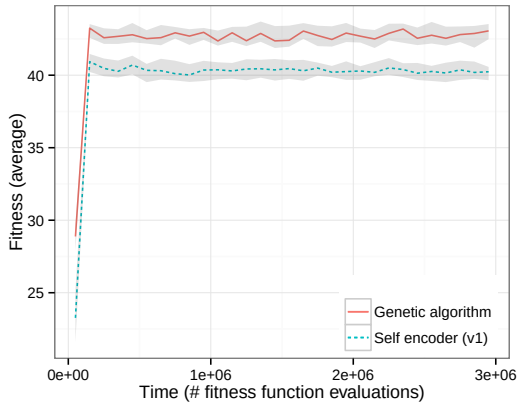


Figure 3: The window-averaged fitness of the self encoder and genetic algorithm over time in simulation 1, with a population size of 20×20.

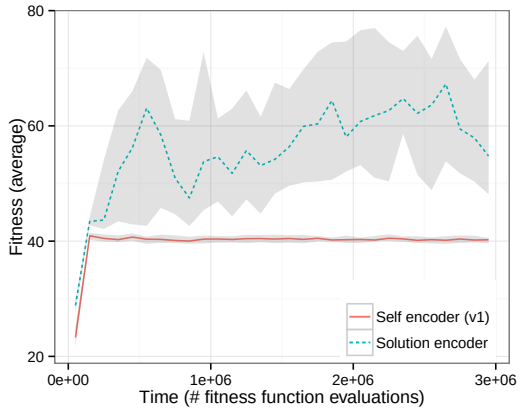


Figure 4: The window-averaged fitness of the self encoder and solution encoder over time in simulation 1, with a population size of 20×20.

genomes from the self encoder population.

```
AAGGGHGGDGGGJGG
AAAGEEEEEEAEEEEEFE
GAAJGJFJJJEJHJJJJJJ
```

The organisms are evolving to have `AA`, the punctuation that indicates the end of the production rule specification and the start of the solution, right at the start of their genome strings; they will have no production rules, nor will any of their descendants, unless mutation disrupts the punctuation.

We suspect that this happened because the initial search over production rule sets is detrimental, and so selection acts against any organism with production rules. To test this, we introduced a variation of the self encoder, in which `AA` only acts as punctuation if it appears in the second half of the output string. The result is to force the self encoders to have non-empty production rule sets. We call the new population the *self encoder (v2)*, and the original the *self encoder (v1)*.

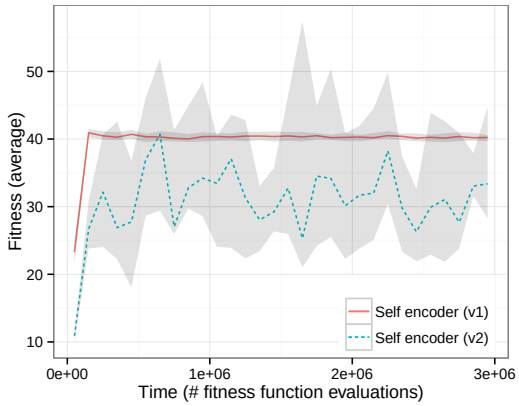


Figure 5: The window-averaged fitness of the self encoder (v1) and self encoder (v2) over time in simulation 1, with a population size of 20×20.

The typical organism in the self encoder (v2) population has a single, long production rule in its decoder. We observed no instances of single-generation self-reproducing organisms, but we frequently found examples of two-generation reproducers, in which an organism and its 'grandchild' have the same production rules. Gestation times were low; typically, organisms went through one or two decoding steps. We found that, often, the genome explicitly contained the solution.

Figure 5 compares the performance of a population of self encoders (v1) and self encoders (v2). The population of self encoders (v2) doesn't perform especially well, but the wide inter-percentile range suggests that, sometimes, the population evolves decoders that confer evolvability.

Simulation 2

The high variance in the evolvability of the self encoder (v2) population in simulation 1 suggests that, sometimes, good sets of production rules do evolve. This motivated us to increase the population size to 20×100 for the next simulation. Since direct competition is between neighbours, the increased population size and rectangular shape has the effect of reducing the rate at which the descendants of a fitter organism spread in the population. This ought to allow multiple competing decoders to co-exist in the population at any time, allowing greater exploration over decoders before any one of them dominates the population.

Figures 6 and 7 show the results. The self encoder (v1) population is again dominated by organisms with empty sets of production rules. The self encoder (v2) performs worse than the self encoder (v1) for the first six or so fitness function windows, then performs better. This appears to show long-term adaptive evolution in the G-P map, the beneficial effect of which remains when the fitness function changes.

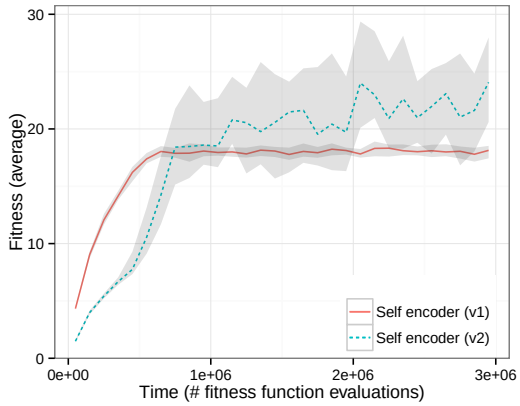


Figure 6: The window-averaged fitness of the self encoder (v1) and self encoder (v2) over time in simulation 2, with a population size of 20×100.

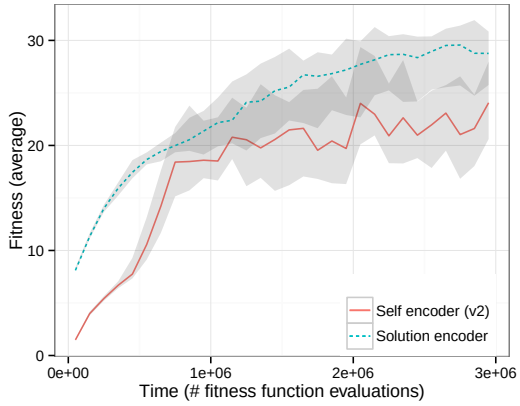


Figure 7: The window-averaged fitness of the self encoder (v2) and solution encoder over time in simulation 2, with a population size of 20×100.

Simulation 3

In the previous two simulations, the self encoder (v2) population became quickly dominated by organisms with decoders consisting of one long production rule. The organisms are not evolving to make use of the 'end rule' punctuation. We measured the relative frequencies of the symbols in the decoded strings over a whole run as follows.

A	B	C	D	E
0.16	0.04	0.04	0.04	0.14
F	G	H	I	J
0.12	0.14	0.14	0.12	0.06

Symbols E-I appear frequently because they are each, in turn, rewarded for. Symbol A appears frequently because it appears in the punctuation that indicates the start of the solution, without which the solution would be empty. Symbols B, C, and D, which appear in the rule-separation punctuation, appear less frequently than symbol J, which doesn't serve any purpose; they are actively selected against. To see what would happen if organisms evolved to have large sets of shorter production rules, we added a restriction: the right-hand-side of each production rule must be less than or equal to ten symbols long.

Figures 8 and 9 show the results of this simulation. Now the self encoder (v2) out-performs the self encoder (v1) almost immediately, showing the adaptive value of searching over decoders. But the self encoder (v1) population is still dominated by organisms that evolve empty sets of production rules. Without the constraint to force non-empty production rule sets, the initial detrimental effect of searching over decoders still prevents the search from ever beginning.

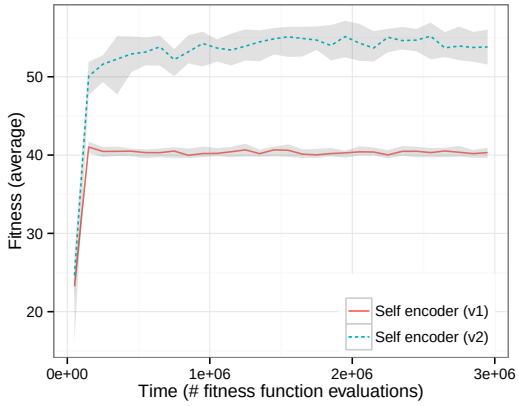


Figure 8: The window-averaged fitness of the self encoder (v1) and self encoder (v2) over time in simulation 3, with a population size of 20×20 and a limit on the size of each production rule.

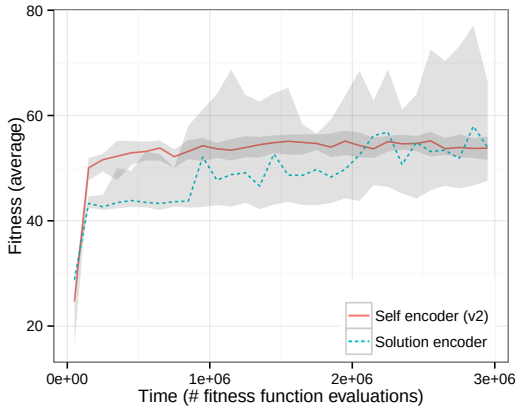


Figure 9: The window-averaged fitness of the self encoder (v2) and solution encoder over time in simulation 3, with a population size of 20×20 and a limit on the size of each production rule.

In the self encoder (v2) population, gestation times were higher than in simulation 2, with organisms typically going through four or five decoding steps. We found that, more frequently than in simulation 2, the solution wasn't contained explicitly within the genome, but was actually grown by applying the production rules. We observed a kind of 'fuzzy' reproduction, in which an organism is similar, but not identical, to its parent or grandparent. For example, the following two sets of production rules are from one organism (left) and its grandchild (right).

B<E>G → AGGAGGAGGA	B<E>G → AGGAGAGEGA
A<G>B → EGAGGAGGAG	A<G>B → EGAGGAGGGA
A<G>G → AGGAGGAGGA	A<G>G → GAGAGABEJF
D<F>H → AGGAGGAGGA	D<F>H → BEFEIFHBEF
E<A>E → IFHAAGGAGG	E<C>C → FDACJEGDAC
E<G>A → GGAGGAGGGH	E<G>A → GGAGGGAGGG
F<E>B → EABEAEBEAB	E<G>G → AGGGAGGAGA
F<E>F → EBEABEAEFE	F<E>F → FHJDGAGEGA
G<A>G → GAGGAGGGAG	G<A>G → EGAGGAGGGG
	G<E>G → AGGAGGGGAG
G<G>A → GBEGAGGAGG	G<G>A → GGGAGAGGAG
G<G>G → AGGJFE	G<G>G → AGGAGBEGAG

Conclusions and Future Work

We have described a model of reproduction, called the self encoder, in which the genome contains an encoded description of the G-P map. In this model, all variation in the offspring, including variation in the G-P map itself, is mediated by the G-P map implemented by the parent. We simulated populations of organisms whose G-P maps were context-sensitive L-systems. Our purpose was to discover whether the self-encoding property has any effect on the evolution of evolvability, as well as to find other factors that lead to the evolution of G-P maps that confer evolvability (with respect to a dynamic fitness function) on the organisms that implement them.

We also looked for the emergence of self-reproducing G-P maps. We observed a kind of 'fuzzy' reproduction, in which organisms had similar G-P maps to their grandchildren or great-grandchildren. In future work we aim to measure the degree of similarity between generations.

We found no conclusive evidence of a difference between the ability of organisms with and without the self-encoding property to evolve evolvability, nor did we rule out such a difference. We found that in the self-encoding population 'good' G-P maps failed to evolve until we added constraints to the model to push evolution towards them. However, once we added the constraints good G-P maps evolved quickly. This was to be expected, as evolvability confers a future advantage, whereas selection acts based on current fitness.

First, we found that populations became dominated by organisms whose L-systems had no production rules. Once we added the constraint that forced them to have non-empty sets of rules, in some cases the population went on to discover G-P maps that increased evolvability. Second, increasing the population size led to the evolution of good G-P maps, since it allowed multiple G-P maps to co-exist for longer; it allowed the exploration of and selection between G-P maps. Third, we found that organisms tended to evolve to have a single long production rule. We added a constraint to force the population to evolve to have a large number of short production rules, and found that as a result the population discovered better G-P maps.

In future work we aim to study a wider range of constraints and mechanisms causing selection for good G-P maps, with an emphasis on mechanisms that are hypothesized to operate in nature.

References

Dawkins, R. (2003). The evolution of evolvability. *On Growth, Form and Computers*, pages 239–255.

Egri-Nagy, A. and Nehaniv, C. L. (2003). Evolvability of the genotype-phenotype relation in populations of self-replicating digital organisms in a Tierra-like system. In *Advances in Artificial Life*, pages 238–247. Springer.

Hansen, T. F. (2006). The evolution of genetic architecture. *Annual Review of Ecology, Evolution, and Systematics*, pages 123–157.

Hasegawa, T. and McMullin, B. (2012). Degeneration of a von Neumann self-reproducer into a self-copier within the Avida world. In *From Animals to Animats 12*, pages 230–239. Springer.

Hasegawa, T. and McMullin, B. (2013). Exploring the point-mutation space of a von Neumann self-reproducer within the Avida world. In *Advances in Artificial Life, ECAL*, volume 12, pages 316–323.

Lindenmayer, A. (1968). Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of theoretical biology*, 18(3):280–299.

McMullin, B. (2012). Architectures for self-reproduction: Abstractions, realisations and a research program. In *Artificial Life*, volume 13, pages 83–90.

Ofria, C. and Wilke, C. O. (2004). Avida: A software platform for research in computational evolutionary biology. *Artificial life*, 10(2):191–229.

Pigliucci, M. (2008). Is evolvability evolvable? *Nature Reviews Genetics*, 9(1):75–82.

Prusinkiewicz, P., Lindenmayer, A., and Hanan, J. (1990). The algorithmic beauty of plants. *The virtual laboratory (USA)*.

Ray, T. S. (1991). An approach to the synthesis of life.

Stanley, K. O. and Miikkulainen, R. (2003). A taxonomy for artificial embryogeny. *Artificial Life*, 9(2):93–130.

Von Neumann, J. and Burks, A. W. (1966). *Theory of self-reproducing automata*. University of Illinois Press, Urbana.

Wagner, G. P. and Altenberg, L. (1996). Complex adaptations and the evolution of evolvability. *Evolution*, 50(3):967–976.