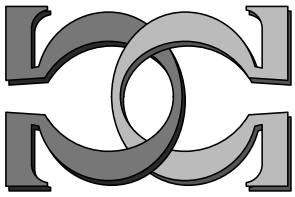
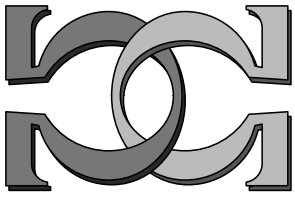
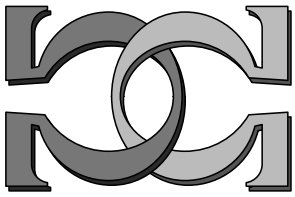
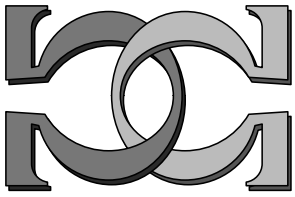


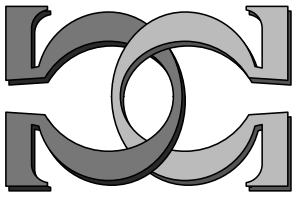




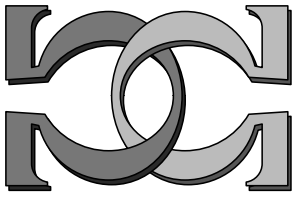
CDMTCS Research Report Series



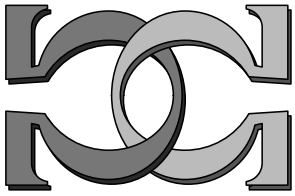
Evolution of Mutating Software



Gregory J. Chaitin
IBM Research, NY, USA



CDMTCS-337
October 2008



Centre for Discrete Mathematics and
Theoretical Computer Science

Evolution of Mutating Software*

Gregory Chaitin†

Abstract

We propose using random walks in software space as abstract formal models of biological evolution. The goal is to shed light on biological creativity using toy models of evolution that are simple enough to prove theorems about them. We consider two models: a single mutating piece of software, and a population of mutating software. The fitness function is taken from a well-known problem in computability theory that requires an unlimited amount of creativity, the Busy Beaver problem.

Key words: evolution, random walk, software space, Busy Beaver function

1 Introduction

This paper proposes modeling biological evolution as mutating software. However at the level of abstraction of this paper, biological evolution and mathematical creativity are not so different.

I have been interested in theoretical biology for a long time [1]. However I could not find a way to go forward. Here we propose an approach that I feel shows promise, but much remains to be done. We only present some preliminary results.

*Talk given Friday October 10, 2008 at the IBM Watson Research Center in Yorktown Heights, NY. The author wishes to thank his colleagues Charles Bennett and David DiVincenzo for their helpful comments.

†IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598, U.S.A., *chaitin@us.ibm.com.*

The immediate stimulus for this paper was Berlinski's entertaining polemic [2], in which he presents a number of criticisms of Darwinian evolution, a number of perplexing aspects of evolution that he feels are not yet well explained by Darwin's theory.

Here are five areas in which I believe that thinking of DNA as mutating software can be helpful. (Software mutations may be point mutations, such as changing or adding a single bit, or they may be high-level, such as copying a subroutine and then modifying it.)

1. **Why are there so many missing intermediate biological forms?** This is not a problem, because a small change in a program can produce an enormous change in its output. Change in output does not have to be continuous, it can be extremely discontinuous.

2. **How are sudden major steps in evolution possible, for example, the transition from single-celled organisms to multi-cellular organisms?** What is happening here is that the main program (a single cell) becomes a subroutine in a larger multi-cellular organism. Changing the main program into a subroutine that is called several times is not a big software change.

3. **Why is there so much fractal, hierarchical structure in organisms?** This is just an extension of the previous point. It's recursive, deeply-nested subroutine structure, which is a good way to build a large piece of software out of smaller pieces of software.

4. **Why does complexity increase?** An increase in the size of a software organism may be due to a genuine increase in function but may also be due to software bloat, the familiar fact that it is easier to add new code than to change existing code.

5. **Why do we need DNA? In other words, why is the genotype different from the phenotype?** Well, if we are mutating software written in a very redundant high-level language convenient for humans, the mutation distance between useful organisms will be much larger than if we are mutating a much more concise representation of these algorithms more similar to a binary machine language. So a way of obtaining redundant programs from compressed versions of them will probably evolve, since it greatly increases evolvability by decreasing the mutation distance between useful software.

> On the other hand, if we are mutating software written in the extremely compact kind of representations used in algorithmic information theory [3], then there is no need for DNA and the genotype can remain the same as the phenotype.

Based on these general considerations, we propose studying the evolution of mutating software. We already made this proposal in [4], but there we could not see where the dynamics comes from. We could not see how to get our software organisms to evolve.

Now I've found a way to do that. In this paper we describe some simple models where you can show that complexity increases due to increased function, not due to software bloat (point (4) above).

2 A problem requiring creativity: Naming big numbers

To get evolution, we must give our software organisms something challenging to do. Here is a simple mathematical problem requiring unlimited creativity, naming big numbers.

For example, 100^{100} is certainly a big number, but we can do much better than that.

Following Hodges [5], write $n \uparrow m$ instead of n^m. Then define $n \uparrow\uparrow m$ as follows:

- $n \uparrow\uparrow 0 = 1$,
- $n \uparrow\uparrow (m+1) = n \uparrow (n \uparrow\uparrow m)$.

Similarly, define $n \uparrow\uparrow\uparrow m$ as follows:

- $n \uparrow\uparrow\uparrow 0 = 1$,
- $n \uparrow\uparrow\uparrow (m+1) = n \uparrow\uparrow (n \uparrow\uparrow\uparrow m)$.

Continue in this fashion with four up arrows, five, etc.

Going beyond Hodges, define $n \Uparrow m$ to be $n \uparrow \cdots \uparrow n$ with m up arrows! What is the value of $999 \Uparrow 999$?

Unlimited creativity is required for this problem, because for any scheme for naming large numbers, one can come up with a better scheme, just like we improved on Hodges.

(See also Steinhaus [6], Knuth [7], Davis [8].)

Naming big numbers is closely related to a classical problem in computability theory, the Busy Beaver problem, and the Busy Beaver function. If you are naming numbers informally as we have just done, then a Busy Beaver function of N can be loosely defined as the biggest positive integer you can name in $\leq N$ symbols or characters of text.

If you are using a programming language to name positive integers, then the Busy Beaver function is the greatest positive integer that can be produced by a program of size $\leq N$ bits (programs that calculate a single integer and then halt). Busy Beaver functions grow faster than any computable function of N, which shows that naming big numbers requires an unlimited amount of creativity.

For more on this, see Aaronson [9] and Wikipedia [10].

3 The formal setting

In a nutshell:

- binary program = genotype = phenotype,
- positive integer output by program = fitness.

3.1 Software space = binary programs for calculating a positive integer

All finite bit strings are valid programs, but some may never produce any output.

We have to pick a universal Turing machine = a general-purpose programming language. I'll use the universal Turing machine that I'm most familiar with, the one U in [3], which has the property that for any other Turing machine C, there is always a prefix π_C such that the output $U(\pi_C\, p)$ produced when the concatenation of π_C with p is run on U, is the same as the output $C(p)$ produced when p is run on C. This shows that the programs for U are concise, or, more precisely, not much bigger than the programs for C.

However, we need to modify the universal machine U of [3] so that it runs forever without producing any output or else produces a single positive integer 1, 2, 3, as output. This is easy to do by filtering the output of U. In

addition, U requires programs to be self-delimiting, and not all bit strings are valid programs. Here we need all finite bit strings to be valid programs. This can be done by modifying U so that it ignores extra program bits and loops forever if it runs out of program bits.

Since U is universal, our software space includes all possible algorithms for calculating a positive integer.

3.2 Point mutations and mutation distance

Point mutations: change a bit, delete a bit, or insert a bit. (Some authors allow adjacent bits to be interchanged, but I'll omit that here.)

The most straight-forward way of defining a metric on software space is as the number of point mutations required to get from one organism to another.

That's the general idea, but sometimes we need a more subtle way to define mutation distance, as $-\log_2$ of the probability of getting from one organism to another via a single mutation. That is the right way to think about mutation distance if you can go from any organism to another in one mutation with small but non-zero probability, which we need to do in model 1 to avoid getting stuck on a local fitness maximum.

3.3 The fitness function: Naming big numbers

The bigger the positive integer that is produced by a program, the fitter it is. If it never produces any output because it never halts, then it is totally unfit.

3.4 We need an oracle for the halting problem

In our models we will need to use an oracle for the halting problem, because if a random mutation would give us a program that never produces any output, we want to be able to skip it. If it does produce output, we can run the program and see how fit it is.

4 Model 1: A random walk in software space

In this model we consider a single software organism, initially the empty program = zero-length bit string. At each step N, we pick a mutation at

random, and check if the resulting organism is more fit. If so, this organism becomes our new step $(N+1)$th organism. Otherwise we pick another mutation at random and continue as before.

The problem here is to get the random walk to cover the entire software space, i.e., be ergodic. If we pick a point mutation at random this will not happen. There is also the problem of being stuck in a local fitness maximum.

To avoid these problems, we can't just use point mutations, we need to do something more sophisticated. We need a small but non-zero probability of going from any software organism to any other.

One way to get this to work, which just happens to be the first way that I could think of, but which is no doubt only one of many possible ways to accomplish this, is as follows:

- To get each new organism, use a single point mutation with probability 1/2, use two point mutations with probability 1/4, three with probability 1/8, etc.

- Also, bias the point mutations to change the beginning of the program. This is a good strategy since our universal computer U reads a self-delimiting prefix from the beginning of the program string and then runs it (see [3]).

- The point mutation will delete, flip, or insert a bit at the first bit of the program with probability 1/2, it will make a change at the second bit with probability 1/4, at the third bit with probability 1/8, etc.

If the rules of the game are set up in this way, a single mutation consisting of many point mutations will eventually insert a prefix at the beginning of the software organism that computes an extremely large positive integer and that ignores the rest of the program string. The result is that one can show that with high probability the fitness of our software organism will grow faster than any computable function of the step N, which shows that genuine creativity is occurring.

I omit the detailed calculations and estimates.

5 Model 2: Evolution in parallel

In this model we consider a population of software organisms, not a single organism. We start as before with a trivial program, and at each stage add to

our population all the software organisms that are one point mutation away from the organisms in our current population. More precisely, each organism gives birth to all those organisms one point mutation away. So by stage N our population will include all N-bit software organisms, since we can add a bit at each stage.

Furthermore, fit organisms have many siblings. When an organism is added to the population, we check its fitness. If this is K, we add K additional copies of that organism to our population.

Programs that produce extremely large numbers K will quickly predominate. In fact, at stage N the organism with the most siblings will be the $\leq N$-bit program that calculates the biggest number K. This value of K is by definition the Busy Beaver function of N bits, the largest output that can be produced by a program $\leq N$ bits in size, and grows faster than any computable function of N. This value of K is also the greatest positive integer with program-size complexity (as defined in [3]) $\leq N$.

Model 2 is simpler than model 1, and it evolves more quickly, because it is evolving in parallel. Note that this is a deterministic model. However, the history leading from the initial organism to each individual organism at stage N, will still look like a random walk.

6 Model 3: The trivial model

Let me now criticize our two previous models. Here is another model of evolution.

At stage N look at all programs $\leq N$ bits in size and select the one that produces the biggest output, which is in fact the Busy Beaver function of N bits, the largest output that can be produced by a program $\leq N$ bits in size.

This is even simpler than our previous model, and does just as well. Why did we take the trouble to formulate models 1 and 2, if model 3 is simpler and does just as well?

The answer is that the problem with model 3 is that it is not at all biological in spirit. This is not how Nature searches through the space of all possible organisms.

7 Discussion

As the trivial model shows, if you have an oracle for the halting problem (and all three of our models do), it is easy to obtain concise names for extremely big numbers. It is the way the search through software space is done that is biological, not the organism that you come up with. (Although the way the search is done will influence the kinds of organisms that you are likely to get.)

I should also emphasize that at best our models are the Platonic ideal that biological evolution attempts to approach, they are not the real thing we see happening around us. Nature does not have an oracle for the halting problem, and the number of organisms cannot increase exponentially indefinitely.

Furthermore, different models may be required to shed light on each of the problems (1) through (5) discussed in Section 1. In this paper, we have only addressed point (4), why does complexity increase? We get complexity to provably increase by choosing a fitness function that rewards creativity and by making sure that our random walks in software space are ergodic, i.e., cover the entire space.

References

[1] G. J. Chaitin, "To a mathematical definition of 'life'," *ACM SICACT News,* No. 4 (January 1970), pp. 12–18.

[2] David Berlinski, *The Devil's Delusion,* Crown Forum, 2008, pp. 192–197.

[3] G. J. Chaitin, *Exploring Randomness,* Springer-Verlag, 2001.

[4] G. J. Chaitin, "Speculations on biology, information and complexity," *EATCS Bulletin,* Vol. 91 (February 2007), pp. 231–237. Reprinted in G. J. Chaitin, *Thinking about Gödel and Turing,* World Scientific, 2007.

[5] Andrew Hodges, *One to Nine,* Norton, 2008, pp. 246–249, *Alan Turing,* Simon and Schuster, 1983, p. 145.

[6] Hugo Steinhaus, *Mathematical Snapshots,* Oxford University Press, 1969, pp. 29–30.

[7] Donald E. Knuth, "Mathematics and computer science: Coping with finiteness," in D. E. Knuth, *Selected Papers on Computer Science,* CSLI Publications, 1996, pp. 31–33, 58.

[8] Martin Davis, *The Universal Computer,* Norton, 2000, pp. 169, 235.

[9] Scott Aaronson, `http://www.scottaaronson.com/writings/bignumbers.html`

[10] Wikipedia, `http://en.wikipedia.org/wiki/Busy_beaver`