Research Article

Active Information in Metabiology

Winston Ewert,[1*] William A. Dembski,[2] Robert J. Marks II[1]

[1]Electrical & Computer Engineering, Baylor University, Waco, Texas, USA

[2]Discovery Institute, Seattle, Washington, USA

Abstract

Metabiology is a fascinating intellectual romp in the surreal world of the mathematics of algorithmic information theory. In this world, halting oracles hunt for busy beaver numbers and busy beaver numbers unearth Chaitin's number, knowledge of which in turn allows resolution of numerous unsolved mathematical problems, many of whose solutions would earn large cash bounties. All this, despite the fact that halting oracles can't be implemented on a computer, a computer can never make a list of busy beaver numbers, and Chaitin's number, always a positive real number less than one, is proven to be unknowable. The fun of metabiology is the application of these ideas to illustrate Darwinian evolution. When metabiology's evolutionary process is stripped of the glitter of algorithmic information theory, however, what remains is a procedure similar to that used in other attempts to model Darwinian evolution, like the *ev* and AVIDA computer programs. Metabiology, like *ev* and AVIDA, succeeds because available sources of knowledge about the solution being sought can be mined. We show the mining of information from a halting oracle has striking similarities to mining information from a simple Hamming oracle. Unlike a halting oracle, however, Hamming oracles can be implemented on a computer. We demonstrate that for both oracles, information can be mined by search strategies that are analogous in some respects even though the methods differ; in both cases the search strategy used greatly influences the result. Because metabiology's process relies on unknowable numbers and infinite resources, its reported relative performance measures can only be expressed asymptotically. That is, the results of metabiology are only proven to be true on the largest possible scale. In fact, simple simulations using bounded resources suggest the asymptote is not always approached quickly, indicating that metabiology results may only hold for scales larger than any practical system.

Cite as: Ewert W, Dembski WA, Marks II RJ (2013) Active Information in Metabiology. BIO-Complexity 2013 (4):1–10. **doi:**10.5048/BIO-C.2013.4

Editor: Douglas Axe

Received: May 4, 2013; **Accepted:** June 17, 2013; **Published:** December 9, 2013

Notes: A *Critique* of this paper, when available, will be assigned **doi:**10.5048/BIO-C.2013.4.c.

* Email: evoinfo@winstonewert.com

1. INTRODUCTION

Gregory Chaitin is a pioneer in algorithmic information theory, which he developed independently from but in parallel with Kolmogorov and Solomonoff [1]. Algorithmic information theory [2] builds on the work of Gödel and Turing, and deals in part with mind-bending mathematics, such as proving there are unprovable propositions and knowing that there are things that can't be known.

Recently, Chaitin has embarked on the project of developing a field he dubs *metabiology* [3]. The underlying impetus for his project is to provide a solid mathematical underpinning for Darwinian evolution. As he states it, "The honor of mathematics requires us to come up with a mathematical theory of evolution and either prove that Darwin was wrong or right [3]." His approach is to model evolution in the abstract realm of computer programs run on Turing machines. By focusing on software, he hopes to learn something about the pure essence of the evolutionary process, which he claims is essentially about software, not hardware.

As in his many other books [2,4–9], Chaitin's description of metabiology [3] is casual, clear, compelling, and mind-bending. Yet in the end, although the mathematics is beautiful, our analysis shows that the metabiology model parallels other attempts to illustrate undirected Darwinian evolution using computer models [10–13]. All of these models depend on the principle of conservation of information [14–21], and all have been shown to incorporate knowledge about the search derived from their designers; this knowledge is measureable as active information [14,22–25]. In order for evolution to occur in these models, external knowledge must be imposed on the process to guide it. Metabiology thus appears to be another example where its designer makes an evolutionary model work.

2. ALGORITHMIC INFORMATION THEORY

Before undertaking a description of metabiology, some foundational ideas from algorithmic information theory need to be established. Outlined below are some of the concepts and terms Chaitin uses.

2.1 The Halting oracle.

Chaitin's model makes use of a hypothetical machine that can produce answers to the Turing Problem. Turing's halting problem [1,26] can be stated simply:

> Given an arbitrary computer program *X*, there is no meta-computer program, *Y*, able to analyze *X* that can announce whether or not, when run, *X* will stop or run forever. A hypothetical device capable of solving the halting problem is dubbed a *halting oracle.*

Turing showed that writing a halting program Y is not possible, using the following logic:

> There are a countably infinite number of Turing machine programs. Each possible program can therefore be indexed using a positive integer. We arrange all these programs in a list starting with the smallest. The p^{th} program is appropriately labeled as an integer p. Let $H(p,i)$ be a halting oracle program that decides if a program p with input i, written $p(i)$ halts or not. $H(p,i)$ outputs a 1 if the program $p(i)$ halts and 0 if it doesn't. Like programs, all possible inputs can be ordered and assigned an integer number, in this case i. Then, consider the program:

```
function N(p) {
    if (H(p,p) == 1) {
        while (1 == 1) {
        }
    }
    return 0;
}
```

> Given a program *p*, this program outputs a 0 when $p(p)$ doesn't halt, and runs forever in a while loop if the program $p(p)$ halts. What, then, of the program $N(N)$? In this case, the program is analyzing itself to see whether or not it will halt. The results are contradictory. If $H(N,N) = 1$ in the program, we get stuck in the while loop forever. But $H(N,N) = 1$ means the program $N(N)$ halts. This is a contradiction. Likewise, if $H(N,N) = 0$ in the program, a zero is printed and the program stops. But $H(N,N) = 0$ means the program $N(N)$ doesn't halt. Another contradiction. Thus, the assumption that there is a halting program $H(p,i)$ that works for all p and i has been proven false.

If the Church-Turing thesis [1] is true, halting oracles do not exist. Conversely, if halting oracles do exist, all open problems in math requiring a single counterexample to disprove them could be solved. A commonly used example is *Goldbach's conjecture*, which hypothesizes that all even numbers greater than two can be written as the sum of two primes. Instances include 10 = 7+3, 1028 = 1021+7, and 143,142 = 71,429+71,713. Goldbach's conjecture has eluded proof since its proposal in the 18th century.

Suppose a program *X* can be written to test each even number sequentially to see if it were the sum of two primes. If a counterexample is found, the program stops and declares "I have a counterexample!" Otherwise, the next even number is tested. If Goldbach's conjecture is true, the program will run forever.

If a halting oracle existed, we could feed it *X*. If the halting oracle says "this program halts," Goldbach's conjecture is disproved. A counterexample exists. If the halting oracle says, "This program never halts," then Goldbach's conjecture is proven! There exists no counterexample.

There are numerous other open problems in mathematics that could be proven or disproven if we had a halting oracle, including the question of the existence of an odd perfect number and the Riemann hypothesis. Many of these problems have substantive cash prizes for those able to solve them!

Metabiology uses the halting oracle in its modeling of Darwinian evolution. Chaitin accurately calls the halting oracle a "mathematical fantasy" [3]. A computer tool proven not to exist is admittedly at the outset an obvious major strike against a theory purporting to demonstrate reality.

2.2 Prefix-free programs[1]

In order to execute a program, it has to be written in a language or code. Within algorithmic information theory, these codes are typically assumed to be prefix-free. If a computer language uses prefix-free code, then no computer program can be a prefix of another program. For example, suppose that the bit stream B = 000 111 000 is a program in a prefix-free language. Then any other programs starting with 000111000 are not allowed. Thus, 000 111 000 **1** = *B* **1** and 000 111 000 **100 101** = *B* **100 101** cannot be computer programs in this language.

This is illustrated in Figure 1 where a binary tree is shown. The tree begins with a zero and a one and begins branching. Some of these branches end with blocks, called *leaves* of the tree. These leaves are binary strings corresponding to prefix-free computer codes. The example shown in Figure 1 is very simplistic. Trees of actual prefix-free computer codes will be enormous with leaves containing thousands of bits.

2.3 Busy beaver numbers

All evolutionary processes seek increased fitness. In metabiology, fitness is found through seeking busy beaver numbers. The meaning of "busy beaver number" is understood in the context of the following question: For a Turing machine with N states that utilizes only zeros and ones, what program will output the largest number? This program is dubbed the *busy beaver program* and the number output is the busy beaver number. As N increases, the *busy beaver number* cannot get smaller.

[1] Prefix-free programs are also called *self-delimiting programs* [1].

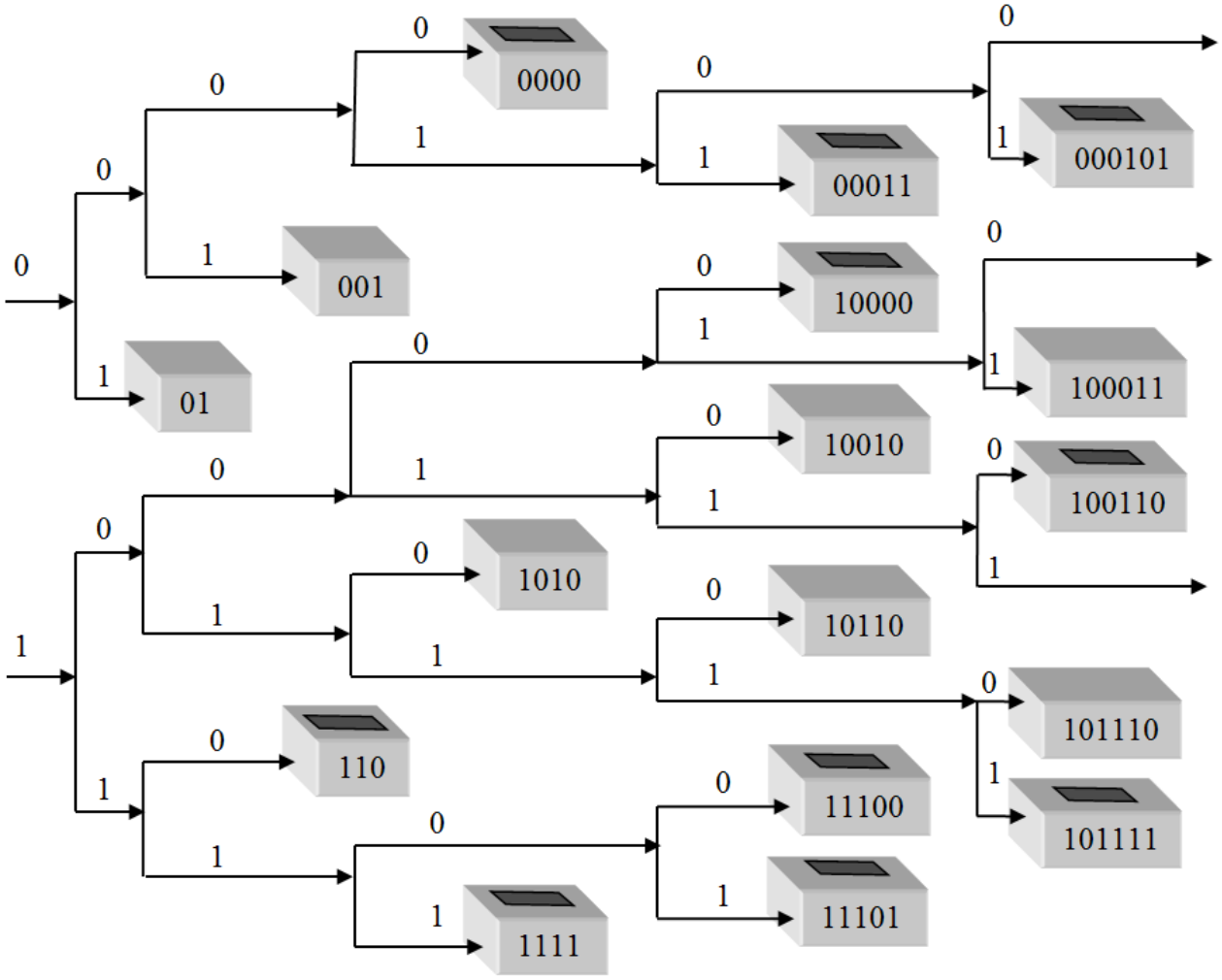

Figure 1: An illustration of a binary tree whose leaves correspond to prefix-free computer codes. The marked leaves correspond to programs that halt.
doi:10.5048/BIO-C.2013.4.f1

Generating larger and larger numbers is not difficult in general. For example, a program can be improved by construction of a new program that runs the original program and adds one to the result. This new program will be slightly longer than the old program. Without imposition of any stop criterion, the search for ever-increasing numbers as N increases requires unbounded computational resources. This increase is enormous in the search for busy beaver numbers.

Although not apparent on first viewing, there is a relationship between busy beaver numbers and the halting problem. Metabiology uses a variation of the busy beaver number, called $BB(K)$, the largest number able to be output by a program of length K bits before the program halts. To understand the relationship between $BB(K)$ and the halting problem, consider the following scenario: We have a program P and we want to determine whether or not it will halt. We can define another program, P', which executes P and then outputs how long it took to run P. If we knew $BB(K)$, where K is the length of P', we would know the longest running time P could take. If P took longer than that to run, we would know that P would never finish. Thus being able to determine the $BB(K)$ would be equivalent to having a halting oracle.

2.4 Chaitin's number

Some of the prefix-free computer programs in Figure 1 halt and some do not. We can assign a weight to each program, 2^{-l} for each program where l is the length of the program. Chaitin's number asks, what is the total weight of all programs that halt? [8]:

$$\Omega = \sum_{\text{all } p\text{'s that halt}} 2^{-\ell_p}. \qquad (1)$$

Due to Kraft's Inequality [1], we know this to be between zero and one. Chaitin's number has remarkable properties, prompting a leading textbook in information theory to dub it "Chaitin's mystical, magical number" [1]. Since Chaitin's number requires knowledge of whether or not a program halts, and halting oracles do not exist, Chaitin's number is unknowable even though it exists.

Metabiology is indirectly related to Chaitin's number and its use in finding busy beaver numbers. Let p be the index of a prefix-free code and let the length of the code be ℓ_p bits. If we run all programs of length L or less for L steps, some of the

programs will stop. We can look at all of these programs and compute:

$$\Omega_L = \sum_{\substack{\text{all } p\text{'s not more than } L \text{ bits that} \\ \text{halt in } L \text{ or less steps}}} 2^{-\ell_p}. \qquad (2)$$

Compare this with Chaitin's number in (1). We are summing over only short programs, each of which has been run for only L steps. Clearly, then, $\Omega_L < \Omega$. But, as $L \to \infty$, all programs will have been run for enough steps for them to halt and $\Omega_L \to \Omega$.

2.5 Big O notation

Big **O** notation describes the limiting behavior of a function when the argument grows unbounded. For example, if $y = 3e^x + x^3 + 4$, then $\mathbf{O}(y) = e^x$. Thus, for x large enough, the e^x term will eventually dwarf the other terms and, eventually, the other terms will become insignificant in their contributions to the numerical value of y.

3. ANALYSIS

In metabiology, in contrast with many other proposed models of evolution, there is no artificially imposed fitness function or artificially designed fitness landscape. Rather, metabiology's landscape flows from the mathematical structure of Turing machine programs. Using the mathematical construct of busy beaver programs, metabiology does not undertake to assist the evolutionary process directly, as many other evolutionary models have done. Rather, metabiology asks whether busy beaver programs can be found by evolving computer programs. While this may seem straightforward on the surface, we note that although mathematics is replete with other number-theoretic landscapes, *e.g.* prime numbers, perfect numbers and twin primes, we are aware of none as exotic as busy beaver numbers.

In order to seek the above-described *busy beaver numbers,* metabiology evolves programs that output very large numbers. These numbers grow faster than any computable function. Chaitin shows that, if they could be written, his programs would produce large numbers very quickly [1,3,27]. But in order to do so, Chaitin's model uses a halting oracle, busy beaver numbers and limitless resources, things that do not exist in reality, or are unknowable, or are unbounded. Because metabiology programs have unbounded length and can run for an unbounded amount of time, the unboundedness essentially undermines the creativity required to solve the large-number problem. With unbounded resources and unbounded time, one can do most anything. One can also quickly exceed the computational resources of the known universe [14,28].

Despite use of these extraordinary tools in its working, metabiology follows other models of evolution that we have analyzed. In particular, the halting oracle is a source of design knowledge. Chaitin rightly states, "[The halting oracle] is where all the creativity is really coming from in our program [3]."

Like gold ore from the side of a mountain, sources of information can be mined from oracles with varying degrees of efficiency. One can mine gold ore with a teaspoon, a shovel, a backhoe, or sophisticated machinery optimized for mining gold ore. Metabiology mines the halting oracle for information using three different methods.

For purposes of comparison and illustration, we'll start by analyzing a Hamming oracle, which is conceptually much simpler than the halting oracle, and show how it can be mined for information using different tools. Then we will show that the methods of mining both the halting and Hamming oracles have striking similarities.

3.1 Mining information from a Hamming oracle

The Hamming distance between two strings of bits is equal to the number of places the two strings differ, e.g., the Hamming distance between 000 111 000 and 110 110 000 is three. A Hamming oracle has a secret, unknown, binary string of L bits, and our job is to guess what it is. We present a string of bits to the Hamming oracle and, without specifying the location of differing bits, the oracle tells us the Hamming distance between our guess and the hidden number. The smaller the Hamming distance, the closer we are to knowing the hidden bit sequence.

The information to be mined is the bit sequence hidden in the Hamming oracle. There are a number of ways to repeatedly query the Hamming oracle to do this; below, we discuss three [29]. We evaluate each algorithm in terms of how many queries each takes to find the bit sequence. The smaller number of queries, the better the algorithm.

Blind search (poor). The Hamming oracle has the ability to specify whether or not a randomly chosen sequence is correct or not. We choose a bit sequence at random and see if the Hamming distance is zero. If it is, we have found the phrase. If it isn't, another phrase is randomly chosen. The process is repeated until there is a success. This is a repeated Bernoulli trial with probability of success 2^{-L}. The expected number of queries to achieve success is:

$$\overline{Q} = 2^L \qquad (3)$$

Evolutionary ratchet search or stochastic hill climbing (good). Here's a better approach that gets to the solution with fewer queries. We begin by choosing an arbitrary string of ones and zeros and present the string to a Hamming oracle. This is dubbed the candidate solution.

a. Choose a bit at random from the string and flip it. If the bit is a zero, we make it a one and, if a one, make it a zero.

b. If the Hamming distance is smaller than before, replace the candidate solution with the new string. Otherwise, retain the older candidate solution.

c. Go to step *a* and repeat until the Hamming distance is zero, in which case the candidate solution is the final solution.

Results	Hamming			Halting Oracle		
		Q	$I_{\boxplus}$		Q	$I_{\boxplus}$
Poor	Blind Search	2^L	$\frac{L}{2^L}$	"Exhaustive Search"	2^L	$\frac{L}{2^L}$
Good	Ratchet Search	LH_L	$\frac{1}{H_L}$	"Random Evolution"	$L^2 \leq Q \leq L^3$	$\frac{1}{L} > Q > \frac{1}{L^2}$
Better	Bit-Flip	L	1	"Intelligent Design"	L	1

Figure 2. Comparison of three ways each to use the Hamming and the halting oracles in a search for a binary string. See the text for definitions. The value given for metabiology's "random evolution" requires interpretation in big **O** notation, which is read "on the order of." **O**(L^2) means "on the order of L^2" (See Section 2.6). Values for Q and $I_{\boxplus}$ should be interpreted correspondingly. doi:10.5048/BIO-C.2013.4.f2

This algorithm will take fewer queries than the blind search. The expected number of queries to a success when we start with a Hamming distance of L is [14]:

$$\overline{Q} = LH_L \tag{4}$$

where the L^{th} harmonic number is:

$$H_L = \sum_{n=1}^{L} \frac{1}{n} \tag{5}$$

As L increases, $LH_L \rightarrow L \ln L$, so the query count in Equation 4 is significantly lower than for the blind search in Equation 3.

Sequential bit-flipping (better). The blind search described above makes no use of changes to the Hamming distance from query to query. The ratchet search, a form of evolutionary computing, does make use of changes in the Hamming distance, whether it is increasing or decreasing. We now show that sequential bit-flipping performs even better than the ratchet search.

The search begins with an initialization presented to the Hamming oracle. The Hamming distance is recorded. Bits are flipped sequentially one at a time starting with the leftmost bit and finishing with the rightmost bit.

a. If the Hamming distance is larger, the bit before the flip was the correct bit.

b. If the Hamming distance is smaller, then the flipped bit is the correct bit.

In either case, the identity of the bit is now known. We freeze this bit into place and proceed to a like examination of the next adjacent bit. The process is repeated until the Hamming distance goes to zero. Each query to the Hamming oracle thus reveals one bit of information about the unknown binary string. A total of L queries or less is required.

There exist even better ways to mine information from a Hamming oracle. For example, none of the searches here uses the actual numerical values provided by the Hamming oracle to choose the next query. The best possible search can be found by a search-for-the-search [30], which Ewert *et al.* [29] have done for a more general (non-binary) Hamming oracle.

The three different searches for mining information from a Hamming oracle described above are summarized in Figure 2. The efficiency of a search algorithm can be measured using *active information per mean query* [14]. The *endogenous information*, I_{Ω}, measures the difficulty of solving a problem when no information is known about the problem. Finding each element in a binary string of length L has an endogenous information of approximately $I_{\Omega} = L$ bits. The mean query count $\overline{Q}$ in Figure 2 thus allows immediate evaluation of the active information per mean query as:

$$I_{\boxplus} = \frac{I_{\omega}}{\overline{Q}} \tag{6}$$

3.2 Mining information using a halting oracle

Metabiology uses a *halting oracle* to mine information. Despite the fact that a halting oracle differs significantly from a Hamming oracle, both oracles can be mined in similar ways to seek identification of an unknown bit stream. Just as the Hamming oracle could be mined in three ways (described above), metabiology mines the halting oracle to search for busy beaver numbers in three ways. The different algorithms are dubbed:

- Exhaustive Search (poor)
- Random Evolution (good)
- Intelligent Design (better)

In metabiology, fitness is measured by comparing the number a program outputs to the busy beaver number $BB(K)$. We here compare three ways to mine information from the halting oracles and show that some are better than others.

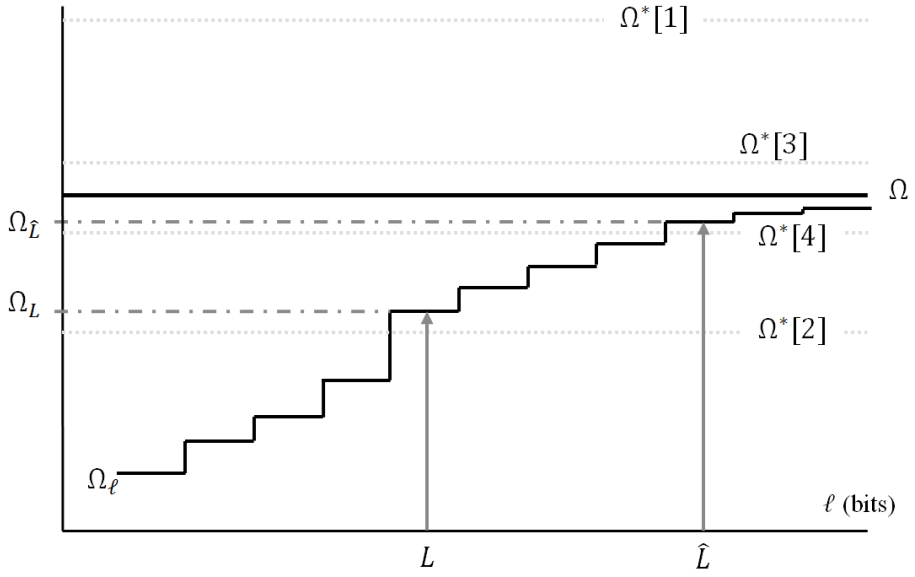

Figure 3. Interval halving in the "intelligent design" version of metabiology. doi:10.5048/BIO-C.2013.4.f3

Exhaustive search (poor). For exhaustive search, a computer program is chosen at random and its performance evaluated. The computer program is chosen by flipping a coin. With reference again to Figure 1, a fair coin is repeatedly flipped until we reach a leaf on the tree. With H = 1 for heads and T = 0 for tails, a flip sequence of TTTHTH gives the program 000101 which is the program in the upper right corner of Figure 1. The randomly chosen program is presented to the halting oracle. If the program doesn't halt, we start afresh flipping the coin to choose another program at random. If the program does halt, we run it and count the number of steps before the program halts. The coin flipping is repeated again and again. Each time we find a program that halts we run it and the number of steps is counted. When a program runs for more steps that have previously been tallied, the new program replaces the old as the best. The expected number of queries for the halting oracle to find the busy beaver number for program with a length of L bits is:

$$\overline{Q} = \mathbf{O}(2^L) \tag{7}$$

Metabiology's "random evolution" (good). Anyone who has written computer code knows that small changes in the code can disable function. If a computer program puts out a large number, then flipping a few bits in the code can result in drastically different performance. The fitness landscape of programs that produce large numbers by binary strings is therefore far from smooth. The specific form of the evolutionary ratchet search (stochastic hill climbing) and the bit-flipping searches we describe above for the Hamming oracle will therefore not be effective for searching among different computer programs. Chaitin, however, comes up with a clever alternative evolutionary ratchet search for extracting large number generators from the halting oracle.

For "random evolution" in metabiology, the random initialization identifies a program, p, that will halt and generate an output. The algorithm steps are:

a. Choose another computer program, M, by flipping a coin.

b. The randomly chosen program, M, will use as input the binary string of the p computer program. To find out whether the program M with input p does or not stop, we use the halting oracle.

- If the program doesn't halt, we go back to step a.
- If the program does halt, we run it and observe the output p'.

c. If the string of bits p' is determined by a halting oracle to be a program that stops, we run it. If the program outputs a longer number than p, we reassign $p = p'$. In all other cases, we keep p as is.

d. Go to step a for the next iteration.

Because of the ratchet feature, each query will result in equal or better fitness values. Eventually, busy beaver numbers will emerge. Remarkably, Chaitin shows that the expected number of queries, $\overline{Q}$, required by this algorithm for success obeys:

$$\mathbf{O}(L^2) \leq \overline{Q} \leq \mathbf{O}(L^3) \tag{8}$$

In the world of big **O** notation, this can be significantly less than the query count for exhaustive search in Equation 3.

Metabiology's "intelligent design" (better). In metabiology, the "intelligent design" search makes the most efficient use of the halting oracle in finding busy beaver numbers. Here is the procedure. We guess a number Ω* and ask whether Ω* > Ω or whether Ω* < Ω. (We are assured equality doesn't happen when Ω* is rational because Ω is irrational.) We write a program X to step through ever increasing values of L and keep track of Ω_L until $\Omega_L \geq \Omega^*$. If we have guessed Ω* > Ω, then X will never halt. We can find this out by submitting X to the halting oracle. If the halting oracle says "X doesn't halt," it is saying "Your guess of Ω* is too big. Guess a smaller value." A smaller

value is guessed and the process is repeated. If the halting oracle, on the other hand, says "X halts," it is telling you that there is an L such that $\Omega^* \leq \Omega_L \leq \Omega$.

We could start with one bit programs in the search, but this would be silly. So let's assume we know the shortest program requires m bits. So we:

1. Run all programs of m bits m steps and compute Ω_m,
2. Run all m and $m + 1$ bit programs for $m + 1$ steps and compute Ω_{m+1},
3. Run all m, $m + 1$ and $m + 2$ bit programs for $m + 2$ steps and compute Ω_{m+2},
4. etc.

We keep going until we get an L such that $\Omega_v \geq \Omega^*$. We know this will eventually happen because the halting oracle says it will. We can then choose a larger value of Ω^* and repeat the process.

An illustration is in Figure 3. A staircase of Ω_ℓ is shown as a function of ℓ that asymptotically approaches Ω as $\ell \to \infty$. Our first guess of Ω is $\Omega^*[1]$. A program X[1] is written to sequentially compute Ω_ℓ for ever increasing ℓ until it equals or exceeds $\Omega^*[1]$. The halting oracle says $X[1]$ will never stop. So the estimate needs to be reduced. We choose $\Omega^*[2]$ and submit $X[2]$ to the halting oracle, which says the program will eventually stop. So we run $X[2]$ until we get an $\ell = L$ such that $\Omega_L \geq \Omega^*[2]$.

Once Ω_L is found, we know the true value of Ω lies between Ω_L , which we know is too small, and $\Omega^*[1]$, which we know is too big. Using interval halving, we next test $\Omega^*[3]$, which lies between these two values. This is shown in Figure 3. The program $X[3]$ is presented to the halting oracle who responds "This program will never halt." In other words, $\Omega^*[3]$ is too big. Now we know Ω lies between Ω_L and $\Omega^*[3]$. The intermediate value of $\Omega^*[4]$ is chosen and $X[4]$ is presented to the halting oracle, which announces the program will halt. Thus, as before, we sequentially evaluate Ω_ℓ's until we find an $\ell=\widehat{L}$ where, for the first time, $\Omega_{\hat{L}} \geq \Omega^*[4]$. We now know that the true value of Ω lies between $\Omega_{\hat{L}}$ and $\Omega^*[3]$. This interval halving process is repeated to get estimates closer and closer to Ω.

Below is pseudocode that does what we just described. For a given value of Ω^*, we first find the smallest value of Ω_L for which $\Omega_L \leq \Omega^*$.

```
function OmegaL(Ω*) {
    L = m − 1
    Ω_L = 0,
    while (Ω<Ω*)    {
    L = L  + 1
       Run all programs of L bits or less for L steps,
       Compute Ω_L in (1)  using ℓ_p's for those
          programs that halted in L steps or less.

    }
    return Ω_L ;
}
```

If $\Omega^* > \Omega$, this program will run forever. Thus, before running it, we need to use the halting oracle to find whether or not OmegaL halts.

Recall, however, that the search is not for Chaitin's number Ω, but for busy beaver numbers. The interval-halving process allows us to find the latter. When all the programs ℓ bits long have been run for L steps ($\ell > L$), some of the programs have stopped and some have not. Of those that have halted, the one that ran the longest is a lower bound to $BB(\ell)$. As the interval-halving search progresses, more and more programs will halt giving better and better estimates of $BB(\ell)$. Eventually, the program that generates $BB(\ell)$ will halt and we will have our busy beaver number. We will never know when this occurs, but are guaranteed it will as the search continues endlessly into the future. Unbounded time is an assumption in metabiology.

The interval-halving procedure just described is dubbed "intelligent design" by Chaitin. Except for the first few choices of Ω^*, the search algorithm is deterministic as is the case with all interval-halving searches.

Summary. We compare the results obtained using the different search strategies and the Hamming oracle or halting oracle in Figure 2. For both exhaustive and blind search, both programs require 2^L queries on average. The introduction of either ratchet search or metabiology's "random evolution" greatly decreases the number of expected queries. A more finely tuned algorithm such as bit-flipping or Chaitin's intelligent design decreases the number of expected queries even more.

Metabiology mines information from a halting oracle, which Chaitin quite properly refers to as its source of creativity. While the same source is used in each case, the three different search strategies mine active information from that source in different ways and with different degrees of efficiency. The poor algorithm requires a large number of queries, and extracts a small amount of active information per query. The good algorithm, based on evolutionary search, takes fewer queries, and extracts more information per query. It makes better use of the available information source. However, the better algorithm, based on a carefully constructed algorithm to extract the information in the most efficient manner possible, uses still less queries. The evolutionary approach does comparatively poorly (see Figure 2). The same is the case for the search strategies using a Hamming oracle: the "evolutionary" ratchet search performs less well than an "intelligent" bit-flip search.

3.3 Simulation

Metabiology works in the theoretical realm of algorithmic information theory, where time and space limitations mean nothing, halting oracles can be used despite not existing, and all that matters is asymptotic behavior described by big **O** notation. A realistic model may behave quite differently. In order to test this, we have simulated a test of metabiology, using something similar to Chaitin's model, but with resource constraints.

To do this, we used the programming language P'' [31,32]. The language uses four symbols: R (which moves the Turing machine tape to the right); λ, which increments the current symbol and moves to the left; and parentheses, "(" and ")" to demarcate a loop: the instructions inside are repeated as long as the symbol under the head of the tape isn't the first symbol. For our purposes, our symbols are limited to R, λ, "(", and ")",

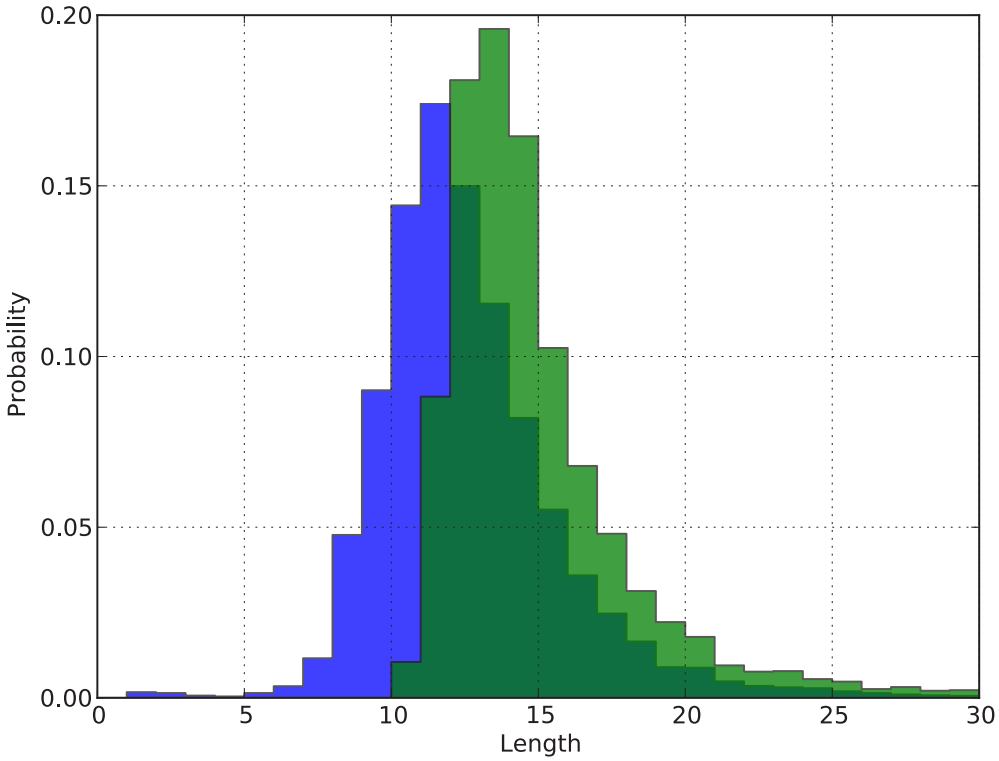

Figure 4: Comparison of random evolution and exhaustive search on metabiology. doi:10.5048/BIO-C.2013.4.f4

where R is the first symbol. The programs were executed using a tape with these four symbols so that the programs could naturally be taken as the input and output of other programs.

Since P'' is a *Turing complete language*, it is possible to create loops which will never terminate. Metabiology solves this problem by using a halting oracle, which we do not have. We approximate this by limiting execution time to 100,000 cycles. Any program executing for longer than that is assumed to be in an infinite loop, and is terminated.

We can generate random programs of various lengths using P''. One of the four symbols is repeatedly chosen uniformly. We require every right parenthesis ")" be balanced by a preceding left parenthesis "(" in order for the program to be valid. If ")" is chosen without a balancing "(", the random generation stops instead of adding the symbol. Should the program reach a length of 100,000 it is also stopped.

For random evolutionary search, we start filtering randomly generated programs until we find one that halts. For 10,000 iterations, the current program is fed as input to another randomly generated program. The output is tested, and if it produces a longer output tape than the original, it is accepted; otherwise the previous program is kept.

For exhaustive search, 10,000 random programs were generated and tested. The best of all these programs is reported as the result.

Figure 4 shows the results of the experiment run for 50,000 Monte Carlo iterations, using either the random evolution or the exhaustive search strategy. Contrary to metabiology's performance as suggested by big **O** notation, exhaustive search produces longer programs with higher probability than does random evolution, when the number of trials are limited. In other words, in this simulation exhaustive search is better than evolutionary search. How can this be? Metabiology's claim that evolution is better than random search only applies asymptotically, under conditions of unlimited resources. Our P'' model will eventually converge on metabiology's asymptotic result, given additional time. Thus, given enough generations and a large enough limit on execution time, metabiology evolution will beat randomness, but in the short term, metabiology does not demonstrate successful Darwinian evolution.

4. CONCLUSION

4.1 Resources and reality

Although elegant in conception, metabiology departs from reality because it pays no attention to resource limitations. Metabiology's math obscures the huge amounts of time required for the evolutionary process. The programs can run for any arbitrarily large number of steps. Additionally, programs can be of any length with no penalty imposed for longer programs.

As mentioned, big **O** notation only describes the limiting behavior of a function when the argument grows unbounded. The notation must be interpreted with care. If $y = 3e^t - e^{1000}$, then $\mathbf{O}(y) = e^t$ even though $3e^t$ will not overtake e^{1000} in all of recorded history with t measured in picoseconds. Furthermore, big **O** does not distinguish between $y = 3e^t$ and $y = 10^{100000}e^t$. This means that huge costs that are not apparent from results presented in big **O** notation are hidden in the analysis of metabiology. If we take these into account, we see that running a program for the number of steps indicated in Figure 3, requires

more computational resources than are universally available [14,28].

The typical definition of the busy beaver problem is the search for the program that produces the largest number amongst a certain class of possible Turing machines. When this class is finite, there is a busy beaver program. One of the programs under consideration will produce the largest number, even though it is in general impossible to mechanically determine the correct answer.

Metabiology considers the class of *all* programs, not merely those limited by a certain size. As a result, there is no program that is a busy beaver, only some programs with larger output then others. It is always possible to produce a longer Turing machine program, which produces a larger number. To solve the busy beaver problem requires creativity, because the program must make the most out of limited resources. Metabiology lacks this requirement for creativity.

Chaitin notes that as his metabiology is made more biologically realistic he will probably be unable to prove results and instead have to be content with simulation. It seems that the first step toward making metabiology realistic would be the introduction of resource limitations. However, it is the very absence of limitations that makes the proofs work. We suspect that any emerging version of metabiology will be akin to Hamming oracle mining, or any one of a number of other computer models purporting to demonstrate Darwinian evolution [10–14,22–25].

4.2 Metabiology landscapes

An interesting part of metabiology is the demonstration of evolution amongst Turing machine programs, which we would not suspect are suitable for the evolutionary process. We would think that changing a single bit of a Turing machine program could produce very large changes in the output. As any computer programmer will tell you, landscapes of computer program fitness are the opposite of smooth. We would therefore not expect Darwinian evolution to fare well. Chaitin notes this when he writes [3], "The fitness landscape has to be very special for Darwinian evolution to work."

The environment for evolution to occur, therefore, has to be carefully designed. Indeed, in the paradigm of conservation of information, smooth landscapes can be source of significant active information [14]. Metabiology's construction of smooth landscapes is accomplished by running all viable programs, a computationally expensive approach that is only possible because there are no resource limitations.

While metabiology is fascinating, it falls short of demonstrating the abilities of undirected Darwinian evolution. Other programs have claimed to simulate Darwinian evolution. These incluvde AVIDA [11,23] and *ev* [12,33]. Like AVIDA and *ev,* metabiology makes use of external information sources to assist in the search. Like the simple Hamming oracle, the halting oracle can be mined for information with various degrees of sophistication. Evolution thus requires external sources of knowledge to work. The degree to which this knowledge is used can be assessed using the idea of active information.

Chaitin states [3], "For many years I have thought that it is a mathematical scandal that we do not have proof that Darwinian evolution works." In fact, mathematics has consistently demonstrated that undirected Darwinian evolution *does not* work. The multiverse is neither large enough nor old enough [14,28]. Consistent with the laws of conservation of information, natural selection can only work using the guidance of active information, which can be provided only by a designer.

1. Cover TM, Thomas JA (2006) Elements of Information Theory. Second edition. Wiley-Interscience (Hoboken, NJ).
2. Chaitin GJ (1987) Algorithmic Information Theory (Cambridge Tracts in Theoretical Computer Science). Cambridge University Press (Cambridge).
3. Chaitin GJ (2013) Proving Darwin: Making Biology Mathematical. Vintage (New York).
4. Chaitin GJ (2002) The Limits of Mathematics: A course on Information Theory and the Limits of Formal Reasoning. Springer (New York).
5. Chaitin GJ (1999) The Unknowable. Springer (New York).
6. Chaitin GJ (2001) Exploring Randomness. Springer (New York).
7. Chaitin GJ (2002) Conversations with a Mathematician: Math, Art, Science and the Limits of Reason. Springer (New York).
8. Chaitin GJ (2006) Meta Math! The Quest for Omega. Vintage (New York).
9. Chaitin GJ (2007) Thinking About Gödel and Turing: Essays on Complexity, 1970-2007. World Scientific (Singapore).
10. Dawkins R (1996) The Blind Watchmaker: Why the Evidence of Evolution Reveals a Universe without Design. Norton (New York).
11. Lenski RE, Ofria C, Pennock RT, Adami C (2003) The evolutionary origin of complex features. Nature 423: 139–144. **doi:**10.1038/nature01568.
12. Schneider TD (2000) Evolution of biological information. Nucleic Acids Res 28: 2794–2799. **doi:**10.1093/nar/28.14.2794.
13. Thomas D (2010) War of the weasels: An evolutionary algorithm beats intelligent design. Skept Inq 43: 42–46.
14. Dembski WA, Marks II RJ (2009) Conservation of information in search: measuring the cost of success. IEEE T Syst Man Cy A 39: 1051–1061. **doi:**10.1109/TSMCA.2009.2025027.
15. Ho Y-CHY-C, Zhao Q-CZQ-C, Pepyne DL (2003) The no free lunch theorems: Complexity and security. IEEE T Automat Contr 48(5) 783-793. **doi:**10.1109/TAC.2003.811254.
16. Koppen M, Wolpert DH, Macready WG (2001) Remarks on a recent paper on the "no free lunch" theorems. IEEE T Evolut Comput 5: 295–296. **doi:**10.1109/4235.930318.
17. Mitchell TM (1990) The need for biases in learning generalizations. In: Shavlik JW, Dietterich TG, editors. Readings in Machine Learning. Morgan Kaufmann (San Mateo, CA). pp. 184–190.
18. Pepyne DL, Ho Y-C (2001) Simple explanation of the no free lunch theorem of optimization. IEEE Decis Contr P 5: 4409-4414. **doi:**10.1109/.2001.980896.
19. Schaffer C (1994) A conservation law for generalization performance. In: Cohen W, Willian H, editors. Proceedings of the Eleventh International Machine Learning Conference. pp. 259 – 265.
20. Wolpert DH, Macready WG (1997) No free lunch theorems for optimization. IEEE T Evolut Comput 1: 67–82. **doi:**10.1109/4235.585893.

21. Elsberry W, Shallit J (2009) Information theory, evolutionary computation, and Dembski's "complex specified information." Synthese 178: 237–270. **doi:**10.1007/s11229-009-9542-8.
22. Dembski WA, Marks II RJ (2009) Life's conservation law: Why Darwinian evolution cannot create biologica information. In: Gordon B, Dembski WA, editors. The Nature of Nature. ISI (Wilmington, DE).
23. Ewert W, Dembski WA, Marks II RJ (2009) Evolutionary synthesis of NAND logic: Dissecting a digital organism. IEEE Sys Man Cybern pp. 3047–3053. **doi:**10.1109/ICSMC.2009.5345941.
24. Ewert W, Dembski WA, Marks II RJ (2012) Climbing the Steiner tree–Sources of active information in a genetic algorithm for solving the Euclidean Steiner tree problem. BIO-Complexity 2012(1): 1-14. **doi:**10.5048/BIO-C.2012.1.
25. Dembski WA, Marks II RJ (2009) Bernoulli's principle of insufficient reason and conservation of information in computer search. IEEE Sys Man Cybern pp. 2647-2652. **doi:**10.1109/ICSMC.2009.5346119.
26. Turing AM (1936) On computable numbers, with an application to the Entscheidungsproblem. Proc London Math Soc 42: 230–265. **doi:**10.1112/plms/s2-42.1.23.
27. Rado T (1962) On non-computable functions. Bell Syst Tech J 41: 877–884.
28. Lloyd S (2002) Computational capacity of the universe. Phys Rev Lett 88: 237901. **doi:**10.1103/PhysRevLett.88.237901.
29. Ewert W, Montañez G, Dembski WA, Marks II RJ (2010) Efficient per query information extraction from a Hamming oracle. 42nd Southeastern Symposium on System Theory. IEEE. pp. 290–297. **doi:**10.1109/SSST.2010.5442816.
30. Dembski WA, Marks II RJ (2010) The search for a search: Measuring the information cost of higher level search. J Adv Comput Intell Intell Informatics 14: 475–486.
31. Böhm C (1964) On a family of Turing machines and the related programming language. ICC Bull 3: 185–194.
32. Böhm C, Jacopini G (1966) Flow diagrams, Turing machines and languages with only two formation rules. Commun ACM 9: 366-371.
33. Montañez G, Ewert W, Dembski WA, Marks II RJ (2010) A vivisection of the ev computer organism: Identifying sources of active information. BIO-Complexity 2010(3): 1-6. **doi:**10.5048/BIO-C.2010.3.