ELSEVIER

Physica D 80 (1995) 154–170

PHYSICA D

Learning and complexity in genetic auto-adaptive systems

Chris Adami

W.K. Kellogg Radiation Laboratory 106-38, California Institute of Technology, Pasadena, CA 91125, USA

Received 31 March 1994; revised 10 June 1994; accepted 21 June 1994
Communicated by K. Kaneko

Abstract

We describe and investigate the learning capabilities displayed by a population of self-replicating segments of computer code subject to random mutation: the tierra environment. We find that learning is achieved through phase transitions that adapt the population to the environment it encounters, at a rate characterized by external parameters such as mutation rate and population size. Our results suggest that most effective learning is achieved close to the transition to disorder, and that learning curves of evolutionary systems are fractal.

1. Introduction

Our concept of learning, in artificial as well as natural systems, despite a plethora of instances, applications, and model systems, has remained intuitive. Indeed, there is as yet no general theory of learning (except for very specific systems [1]) and this omission is apt to become more and more crucial as experiments in learning become more and more varied and diverse. One of the more elusive tasks associated with formulating a theory of learning is the isolation of *universal characteristics* of the learning process. In fact, the very existence of a universal learning process has yet to be established.

In this paper, we would like to shed some light on the learning process in a very specialized artificial system that nevertheless promises to exhibit universal features. This system offers the possibility to study learning from a biological, i.e. evolutionary, point of view. Evolution of DNA is perhaps the most dazzling instance of learning through adaptation of which we know. Yet, it seems to be of little use for machine learning applications for a very obvious reason: Learning through evolution is inherently slow. We hope nevertheless that by studying this immensely successful adaptive process, new insights can be gained which can be carried over to artificial learning systems.

In the next section, we point out the qualitative differences between evolutionary learning (as displayed by natural genetic systems) and a variety of popular adaptive schemes that are in use today from an abstract point of view. The classification of learning processes introduced there is important for those readers interested in the conceptual foundations of learning, but may be skipped by those only interested in the results. Section 3 introduces the tierra system that serves as a paradigm for auto-adaptive learning throughout this paper, while the fourth section rigorously defines observables in tierra and introduces the equations that describe population kinetics. Section 5 then describes universal characteristics of the tierra system emerging from extensive simulations. We describe a typical

SSDI 0167-2789(94)00157-L

tierra "experiment" in some detail and present results of an investigation of the learning rate as a function of the external mutation rate, i.e., the force that drives evolution. We offer conclusions in the last section.

2. Learning in adaptive systems

When investigating learning, we are interested in the macroscopic behaviour of a system in response to external stimuli. When specifying the macroscopic state of the system, we are faced with two possibilities: We may either specify the *space* of macroscopic states by enumeration (i.e., providing each state fully formed), or else provide a set of *microscopic* states together with a set of rules to construct the macroscopic ones. Either of these approaches has its advantages. The macroscopic implementation is well suited for complex tasks to be learned as each preprogrammed state can in principle be of arbitrary complexity. On the other hand, as will become clear later, flexibility is lost and the set of possible states is necessarily finite. The microscopic approach does not suffer from the latter problem because the microscopic rules can be combined in an infinite number of ways to produce a practically infinite set of macroscopic states. Finding a "microscopic alphabet" in which every macroscopic rule can be formulated, however, appears daunting most notably due to the hierarchy problem and the brittleness problem.

The hierarchy problem is most easily understood by considering its analogue in natural language: the parsing problem. In natural language, the meaning of a sentence can *not* be a universal function of the words, simply because words have no *intrinsic* meaning at all. Rather, the meaning of a word is given by all the possible ways it can be used in a meaningful sentence. Thus, there is no meaning on the microscopic level, whereas clearly there is a meaning on the macroscopic level. The mapping between the levels cannot be performed by a universal function because while words are universal (the same set of words are used to construct all sentences) the sentences are not (the meaning of sentences is context-specific). Thus, in natural language the hierarchy problem is to find a mapping from the microscopic level to the macroscopic one that is *not* a universal function. In learning systems, fitness replaces meaning, microscopic states (the alphabet) replace words, and macroscopic states (the rules) replace sentences. The alphabet must be devoid of intrinsic fitness in order to guarantee universality, i.e., the fitness of a certain arrangement of the microscopic states should not be a universal function of the fitness of each member of the alphabet, while we would like to see fitness emerge (on the macroscopic level) that is inherent to the context and thus *nonuniversal*, and only reflects the properties of the environment, i.e. the learning task at hand.

The brittleness problem is well-known: an arbitrary arrangement of microscopic rules leads to nonsensical macroscopic rules in almost all cases, and the space of macroscopic states turns out to be mostly empty. This problem most notably arises with computer-code for von Neumann machines: the ratio of possible programs to workable ones is almost zero, and any arbitrary mutation of a working program will most likely break it.

As a consequence of these problems, most approaches to the learning problem are based on the macroscopic implementation. Here, the major players in the field are Artificial Neural Networks [2], Genetic Algorithms [3,4] (including Expert Systems), and certainly Kauffman's NK-model [5]. All these are instances of "adaptive" systems, which learn by adapting to the fitness landscape dictated by the task to be learned. They share the ubiquitous feature that is the feedback mechanism: a process which modifies parameters that determine the response of the system to a certain input, according to the fitness, or success rate, of the previous set of parameters. In conventional adaptive systems, the mechanism to determine the fitness of a parameter set is extraneous to the system itself. This is of course a direct consequence of the inability to provide a problem-independent microscopic alphabet, as the parameter-string (or set of weights and thresholds) has *no significance* except when interpreted within the context of the fitness-function or error-function. Thus, the system can never learn anything outside the boundaries specified by this function: flexibility is lost. As it turns out, nature

seems to have found a solution to this problem, and we attempt to emulate this approach.

In almost all cases of learning in natural systems, the fitness of a certain configuration (or "hypothesis" [1]) is determined *within* the system. Thus, we strive for the fitness of a string (in the broad sense of NK-models) to emerge as a collective effect from the interaction of the environment (a "hard-coded" set of parameters) and the population. In a way, we would like the strings to *compute their own fitness*. We shall call systems that can perform this feat "auto-adaptive", to emphasize the fact that we *do not provide a fitness- or error-function*.

Fig. 1 is an attempt at schematizing adaptive and auto-adaptive systems. We assume that the information content of any learning system may be coded in bit strings. In adaptive systems (Fig. 1a), the bit strings are translated into macroscopic sets of rules [1]. This interpreter is necessarily problem-specific, and the construction of the rules (the action of the interpreter) is *fast* (on the time scale associated with the learning process). The fitness of this macroscopic rule-set is then computed via the external fitness-function, which is also problem-specific, and fast. The result of the fitness evaluation is used to select bit strings in the next generation. The bit strings of auto-adaptive systems (Fig. 1b) are first translated to a microscopic rule-set. This interpreter is quasi-universal: the same microscopic rule-set can in principle be used for any application, although it may in most cases turn out to be advantageous to adapt the interpreter to a specific *class* of problems. The action of this interpreter is fast. The promotion of microscopic rules to macroscopic ones proceeds via evolution, i.e., mutation and "natural" selection [2]. This process is universal, but slow on the time scale of generations. The fitness evaluation then does not require any more manipulation. Instead, the fitness emerges through the (social or non-social) interaction of the macroscopic rule-sets in the population simply by survival. Thus, fitness is the direct result of the actions and interactions of the members of the population, and is automatically the vehicle for selection of bit strings that survive in the next generation. Inevitably, for this to work the bit strings have to self-replicate.

The only (artificial) system, that (to our knowledge) is truly auto-adaptive was designed to mimic nature in a number of important aspects. The tierra environment [6], a software package created recently by Tom Ray, is one where a population of self-replicating segments of computer code (alternatively called "programs", "cells", or "creatures") thrives in an environment that is managed by the tierra program itself. The latter provides not only resources to the cells (CPU-time and memory space), but also oversees births, mutations, and deaths, along with providing the "shells" in which the creatures live: a virtual computer for each living cell in the population. Before we go on to describe the key aspects of the tierra system, we would like to clarify the recurrent use of metaphors culled from biology. In fact, tierra was designed around these metaphors, in the sense that certain devices of the computing environment were designed to *play the same role* as certain devices, in the broadest sense, occurring in nature. Thus, CPU-time is *analogous* to energy, memory allocation is *analogous* to birth, machine-language instructions (the microscopic rule-set) are *analogous* to the codons of DNA [3]. It turns out *a posteriori* that such a system of analogies and metaphors can, to the extent dictated by hardware limitations, emulate the evolution of simple proto-cellular systems to an astonishing degree [6].

The replication and mating operations, extraneous to the population of strings in for example Genetic

[1] We use the terms "rule-sets" and "states" synonymously, as each state of a system can in fact be viewed as a set of rules to handle input and output.

[2] We put "natural" in quotes since the selection of strings is necessarily dictated by the user-specified environment. However, we would still like to use the term "natural" to distinguish it from "artificial selection" based on the output of a fitness-function. More accurate terms would be "internal" as opposed to "external" selection.

[3] DNA is coded in base 4 deoxyribonucleotides, such that any sequence of 3 represents a codon that is translated into an amino acid (the microscopic rule-set of nature). Thus, 4^3 codons are translated into 20 amino acids, while 2^5 combinations of 1's and 0's are translated into 32 instructions in tierra, some of which turn out to be rarely used and could just as well be eliminated. In DNA, those amino acids that are used most frequently have the most representations in terms of codons. Such an approach could easily be implemented in tierra also.

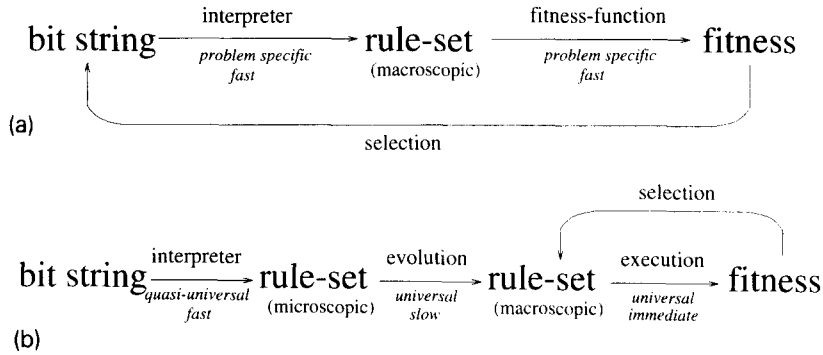

Fig. 1. (a) Feedback loop for adaptive genetic systems for selection of strings. (b) Selection process for auto-adaptive genetic systems.

Algorithms (GA's), is inherent to the tierra community of cells and as such the control of these activities is shared between the environment and the make-up of the population. Giving up control over key parameters has profound consequences for the macroscopic behaviour of the population. Loss of microscopic predictability increases the complexity of the system to such a degree that studies of the tierra system are in effect experiments with tierra. Concurrently, complexity ensures that the collective behaviour of the population is genuine and reproducible, and, in its general characteristics, universal.

3. The tierra system

The notion to evolve computer programs by means of random mutation appears doomed owing to the fact that the ratio of working programs to possible ones is very close to zero for most existing languages. In other words, any random mutation of a program is likely to break it. This has been known for some time as the problem of "brittleness". On the other hand, mutation does quite well in living systems, and according to Darwinian theory, is responsible for the emergence of complexity in natural living systems. Ray dissolved this dichotomy by designing an assembly language based on a number of instructions of the same order of magnitude as the number of amino acids in the genetic code. Specifically, he chose to code these instructions into five bits, such that the random mutation of any bit would be "contained" and lead to a different instruction of this family. This is the central idea for surmounting brittleness, and possibly the key to auto-adaptive systems in general. Another characteristic of the tierran instruction set garnered from nature is the use of templates (patterns of instructions) for addressing purposes rather than absolute addresses. From a computing point of view, the 32 instructions used by the tierran creatures are similar to machine language instructions; an extremely reduced instruction set running on the virtual computers provided by the tierra program. The virtual CPU is kept very simple using four registers, a stack, input/output buffers, and an instruction pointer. Table 1 shows the mapping from the tierran codons to instructions.

The intended analogy is for the strands of computer code to represent strands of DNA, while the tierra program fulfills the role that chemistry plays in nature. Specifically, it doles out CPU time-slices to the cells in the group (simulating parallel coexistence) and supervises the "aging" of the cells by arranging them in a "reaper queue", killing the oldest cells in the strip of memory reserved for the cells (the "soup") if there is not enough room to accommodate the new-born ones. Details of the operation of the queues and the observing software which is part of the tierra program can be found in [6] and in the documentation of the tierra software [8].

Evolution of the population is guaranteed by a rate of bit mutation that affects every cell in the soup to the same degree (this is the analog of cosmic rays). Mutations in the cells due to this phenomenon and to random copy-errors seems to be the key mechanism

Table 1
Mapping of 5-bit codons to instructions in the tierran instruction set used in the present simulations. A description of the commands can be found in the tierra manual [8].

00000 nop0	01000 pushdx	10000 dec	11000 jmp
00001 nop1	01001 popax	10001 add	11001 jmpb
00010 movdi	01010 popbx	10010 sub	11010 call
00011 movid	01011 popcx	10011 zero	11011 adr
00100 movii	01100 popdx	10100 shl	11100 adrb
00101 pushax	01101 put	10101 not0	11101 adrf
00110 pushbx	01110 get	10110 ifz	11110 mal
00111 pushcx	01111 inc	10111 iffl	11111 divide

that drives the emergence of complexity, learning, and diversity. The "splicing" mechanism of mating which is the corner stone of the evolution of GA's arises in tierra as a secondary effect, by copying an incomplete creature (incomplete due to a mistake in a cell's calculation of its own length as a result of mutation and flaws) into the space previously held by a defunct one, thus splicing these codes together and recycling the dead instructions. It turns out that this mechanism plays an important role in learning and the evolution of complexity on short time scales. Also, it is an example of an emergent characteristic; it was not anticipated by the designer [9].

A typical tierra experiment starts by inoculating empty memory by a self-replicating creature that is hand-written by the operator using any suitable instruction set. Throughout, we inoculate the soup with our equivalent of a program written and termed "the ancestor" by Ray[4]. The ancestor is a code consisting of 82 instructions that represent Ray's first attempt at writing a self-replicating program for this particular instruction set. As such, it turns out to be very inefficient and is easily improved by mutation. We use it as a progenitor for precisely this reason, since its inefficiency is due to the presence of redundancy in the code. Redundancy has emerged as a necessary requirement for successful evolution. Also, this progenitor possesses only the ability to replicate, and thus is not biased towards learning other tasks. After inoculation, the reserved space for the cells quickly fills up with offsprings of the ancestor, largely identical to it, with exceptions due to mutations. Once the space is filled up, the tierra program removes the oldest cells to provide room for the next generation. As mentioned, age is controlled by arranging the cells in a linear queue. New-born cells are entered at the bottom while the top creature is removed. From the moment of inoculation, the fate of the population is out of the hands of the operator, being entirely determined by the parameters of the tierra program and the physical environment (the "landscape") encountered by the cells (see below). Despite the evidently deterministic relationship between parameters and macroscopic behaviour, the system is complex enough to thwart any attempt at unraveling that connection.

4. Fitness and learning in tierra

As mentioned in the previous section, the fitness of a member of the tierran population is *not* determined by a fitness computation, but rather is a function of the cells genotype[5] *and* of the rest of the population. A universal measure of fitness in tierra, as well as possibly all auto-adaptive systems, artificial and (in a restricted sense) natural, is the number of off-spring ("daughters") of the organism i, d_i, in a suitably chosen time span. In tierra, we take this span to be the

[4] Our ancestor is not exactly identical to Ray's due to some slight changes in the instruction set that we deemed advantageous. The instruction set used in the simulations here is displayed in Table 1.

[5] The genotype of a cell is given by its specific arrangement of instructions. For programs of the same length, different genotypes are arbitrarily labelled by a three-letter code, in order of their appearance in the soup. Thus, the size-82 progenitor is labelled 82aaa, its first mutation or flawed off-spring of the same size with a different genotype is 82aab, and so forth.

lifetime of the organism, τ_i, measured in number of instructions executed. Very obviously, in the absence of a mechanism that allows organisms to kill each other, the genotype with the highest number of off-spring per lifetime will dominate the population. Naturally, this dominance can only be ephemeral as the successful creatures' off-spring will soon compete with it.

The number of daughters (during its lifetime) of organism i can be written as

$$d_i = \tau_i/(t_g)_i \,, \tag{1}$$

where $(t_g)_i$ is the time it takes organism i to gestate a single off-spring, and τ_i is the lifetime of this organism as defined earlier. Naturally, this must be an integer number for any individual cell; for any genotype, however, d_i represents the *average* number of off-spring of this particular genotype, which is in general non-integer.

In the emulation of parallel coexistence, the main program allocates slices of CPU time to each cell in a serial manner. Let $(t_a)_i$ be the time allocated to organism i (measured in number of instructions that this cell will be able to execute) in each sweep through the population. Then

$$\tau_i = \sum_{j=1}^{N_i} (t_a)_{i,j} \,, \tag{2}$$

where N_i is the number of sweeps that creature i obtains. Let us for simplicity also assume that the time allocated each sweep is roughly equal (or equivalently define $(t_a)_i$ to be the average allocated time per sweep). Then

$$\tau_i = N_i (t_a)_i \,. \tag{3}$$

In the following, we will drop the subscript i denoting the value of the respective quantity for organism i, while quantities averaged over the entire soup have angled brackets. I then follows that

$$d = N t_a/t_g \equiv N\alpha \,, \tag{4}$$

where we defined the *fitness fraction* α.

Indeed, this fraction is a function of the genotype of the organism only, and thus represents a good measure of *absolute fitness*. The total number of off-spring d can only be a measure of relative fitness as its value depends on the number of off-spring of other members of the population through its dependence on N. A good estimate for N is obtained by considering the movements in the reaper queue (RQ) due to new births only. As mentioned briefly earlier, every new-born cell is entered at the bottom of the queue, and reaches the top after n more births, where n is the total number of cells in the soup. The oldest cell in the soup is the one at the top of the queue, and suffers the action of the reaper. Since $\langle\alpha\rangle n$ is the average number of cells born each sweep, a constant population implies $N\langle\alpha\rangle n = n$ and thus

$$N = 1/\langle\alpha\rangle \,. \tag{5}$$

It then follows that

$$d = \alpha/\langle\alpha\rangle \,. \tag{6}$$

Trivially then, $\langle d\rangle = 1$, i.e. the average number of off-spring is unity, as must be the case for a constant number of cells. In tierra, however, there is also movement in the RQ which is not due to births and deaths alone. If a cell attempts an illicit operation, be it writing on write-protected memory space (for instance space owned by another creature), or attempting to allocate too much or too little memory [6], an error-flag is set, and the instruction is not executed. Anytime a cell commits such an illicit operation, its total number of error-flags n_e is compared to the number of error-flags generated by the cell just above it in the RQ, and switches places with it if that cell's error-count is larger. Thus, cells that commit more error-flags age faster. On the same token, a cell may be moved *down* the RQ if it accomplishes a task that the user feels worth rewarding. In the present implementation of tierra, a cell moves down one position in the RQ after a successful memory allocation instruction (`mal`), and after a successful `divide` instruction. The number k of downward moves per lifetime ($k = 2d$ in the task-neutral case) is at the discretion of the user and represents a means of rewarding or punishing cells

[6] The specific restrictions are set by parameters of the tierra software. See the documentation for the details.

according to whatever task is to be accomplished. Including these movements inside the RQ, we find the more general expression for the number of off-spring

$$d = \frac{\alpha + \alpha/n\left(\langle n_e\rangle - n_e\right)}{\langle\alpha\rangle + k/n\left[(\langle\alpha\rangle n_e - \alpha\langle n_e\rangle)/(n_e + \langle n_e\rangle)\right]} \,. \quad (7)$$

Note that the corrections to (6) are of the order $1/n$, and thus become more and more unimportant in simulations with large n. This is due to the fact that the reaper kills the oldest cells in the entire soup, while a more sophisticated model would consider removing the oldest cell in a specific neighbourhood of $\tilde{n}$ cells [7].

Another method of rewarding some actions and discouraging others is the distribution of *bonuses* in the form of extra time-slices. For an organism of length ℓ, tierra doles out slices of

$$t_a = (c + f)\ell^p + t_b \quad (8)$$

instructions per cell per sweep. Here, t_b is the average bonus received per sweep, p is a power that can be used to favour larger or smaller creatures (we set $p = 1$ for size neutrality throughout) and f is the "leanness" fraction of the cell, obtained by dividing the number of executable instructions of the cell by its length. This factor is introduced to discourage the development of unexecutable code (as occurs if e.g. a section of the code is jumped over by the instruction pointer. This would be advantageous since it reduces the gestation time as we shall see below). We have supplemented this fraction by a genotype-independent constant c ($c = 0.3$ throughout the simulations reported here). For comparison, the ancestor has $f = 0.54$, but evolution has been able to increase this fraction to close to the theoretical maximum of $f = 1$. Also, new genotypes are assigned $f = 0.5$ at birth until this fraction can be determined after gestation of the first off-spring.

Concurrently to increasing t_a, cells can decrease t_g in order to increase α. Let us divide a typical program into a "work" section of length ℓ_w and a copy loop of length ℓ_c, such that $\ell = \ell_w + \ell_c$ (with typically $\ell_c \ll \ell_w$). The copy loop consists of those instructions that have to be executed to copy instructions from mother to daughter. Thus, to copy ℓ instructions, a total number of $\ell\ell_c/m$ instructions have to be executed, where m is the number of instructions copied by executing the instructions in ℓ_c. In the ancestor $m = 1$; however, the cells quickly discover that increasing m reduces the gestation time. This technique of optimization is generally known as "unrolling the loop" and was observed to occur spontaneously in tierra by Ray [6]. To complete a gestation, the program also has to run through the remaining ℓ_w instructions, such that

$$t_g = \ell_w + \frac{\ell\ell_c}{m} = \ell\left(1 + \ell_c\left[\frac{1}{m} - \frac{1}{\ell}\right]\right) \quad (9)$$

and thus

$$\alpha = \frac{c + f + t_b/\ell}{1 + \ell_c(1/m - 1/\ell)} \,. \quad (10)$$

For $\ell_c \ll \ell$ and small m ($m \lesssim 3$) we find $\alpha \sim m$, i.e. unrolling the loop is an extremely beneficial operation. For larger m the lengthening of the copy loop cuts down this advantage. Likewise, skipping a large part of ℓ_w would turn out to increase α substantially. However, this is detrimental to learning as this is precisely the region where the cells are supposed to develop the code necessary to accomplish a task. For this reason, the leanness factor f was introduced in (8) above.

The mechanism that drives fitness-improvement in tierra is of course mutation. The soup is subject to independent, Poisson-random mutation (bit-flip events), such that the waiting times between mutations are distributed exponentially[7]. The mean time between mutations $\langle t_m\rangle$ is related to the mutation rate R (mutations per site per instruction executed) and the soup size s via

$$\langle t_m\rangle = R^{-1}/s \,, \quad (11)$$

while the probability that two mutation events are spaced by t_m is

$$p(t_m) = Rs\, e^{-Rs\, t_m} = \frac{1}{\langle t_m\rangle} e^{-t_m/\langle t_m\rangle} \,. \quad (12)$$

[7] This is an improvement over the univariate distribution in earlier versions of tierra.

We are now in a position to obtain a relationship between the fitness of a genotype i, α_i, and the mutation rate.

The number of cells of genotype i in the soup at time $t+1$, $n_i(t+1)$, is related to $n_i(t)$ via

$$n_i(t+1) = \left(1 + \frac{\alpha_i - \langle\alpha\rangle}{t_s} - R\ell_i\right) n_i(t) \ . \qquad (13)$$

Eq. (13) simply reflects that new cells of genotype i are born with a rate α_i/t_s (t_s is the time it takes to "sweep" through the soup once, i.e. to execute $\langle t_a\rangle_i$ instructions for each cell in the soup, $t_s = n\langle t_a\rangle$) while the fitness α_i is just the number of off-spring per sweep) and they die with a rate $\langle\alpha\rangle/t_s$ due to births by other genotypes, and with a rate $R\ell_i$ due to mutations. We can neglect here the rate of births of this genotype due to mutations affecting the rest of the soup, since this is infinitesimal in most situations[8]. For simplicity, we also neglect in this equation the effect of mutations due to copy-errors, which enters in the first term of (13). For a copy-error rate R_c (one out of R_c^{-1} instructions are not copied correctly) the term α_i/t_s in (13) should be multiplied by $(1 - R_c\ell_i)$. In the present paper we set $R_c = 1 \times 10^{-3}$ such that it can safely be ignored at medium and high background-mutation rates.

Solving (13) we find for the evolution of the population

$$n_i(t) = n_i(t_0)e^{\gamma_i t} \ , \qquad (14)$$

where $n_i(t_0)$ is some starting population (e.g. $n_i(t_0) = 1$) and (suppressing the genotype-index)

$$\gamma = \frac{\alpha - \langle\alpha\rangle}{t_s} - R\ell \ . \qquad (15)$$

Likewise, this allows us to derive a relation for the maximum mutation rate that a population of fitness α can sustain. The highest strain is put on a population at high mutation rate and the average fitness of the soup is driven close to zero, $\langle\alpha\rangle \to 0$. Then the soup can only survive if the best genotype has $\gamma \geq 0$ [see Eq. (14)], or (assuming $\langle t_g\rangle \ll t_g$)

$$\alpha/t_s \geq R\ell \ . \qquad (16)$$

In other words, there is a minimum fitness (i.e. minimum replication rate) required to survive under the hostile circumstances of a high mutation rate. This condition is similar to the error-threshold condition derived by Eigen et al. in the context of quasi-species in protein-space [10]. Assuming $t_a \approx \langle t_a\rangle$ (true during equilibrium) gives us a more intuitive understanding of the requirements for the survival of a population. Since $t_s = n\langle t_a\rangle$ we find

$$1/t_g \geq R\langle\ell\rangle n = 1/\langle t_m^\star\rangle \ , \qquad (17)$$

where $\langle t_m^\star\rangle$ is the average time between mutations affecting cells (as not all sites in the soup are actual living cell-sites), $t_m^\star = (s/n\langle\ell\rangle)t_m$.

The survival condition is thus a relationship between the two fundamental (small) time scales in the problem, the gestation time t_g and the average time between cell-mutations, $\langle t_m^\star\rangle$. Not surprisingly, we find that we must have

$$t_g \leq \langle t_m^\star\rangle \ , \qquad (18)$$

a relation that we expect to hold quite generally.

By the same token, Eq. (15) tells us how the mutation rate drives the fitness improvement. As equilibrium always drives any genotype towards $\gamma \approx 0$, we find

$$\Delta\alpha \equiv \alpha - \langle\alpha\rangle = R\ell t_s \ , \qquad (19)$$

i.e., the fitness gradient is proportional to the mutation rate. Of course, this equality is violated during the phase transitions that improve the fitness, i.e. during learning.

In order to gain some insight into how the mutation rate affects the *learning rate*, we need to perform actual experiments with tierra. We have seen that there is a maximum rate above which the soup cannot survive, while obviously there can be no learning at $R = 0$. We shall in fact see in the next section that, although a learning rate cannot unambiguously be linked to a mutation rate, they are in effect loosely correlated until near the transition to chaos, which effectively dissolves the population: a state where the error-threshold condition (18) is violated and self-replication stops.

[8] This term however *is* important in a consistent treatment of the statistical mechanics.

5. Characteristics of learning

In order to observe learning in tierra, we investigate a simple problem: learning to add two integer numbers. We choose this problem as a representative of a class of simple problems[9] that can be mastered by a tierran soup, while anticipating that more complex problems can be learned by combining such microscopic tasks. In addition, since the tierra system is a parallel one *in principle* (though not in practice), learning several tasks at once should not require the cumulative time of learning each of them.

As opposed to e.g. learning in Neural Networks, we do not "teach" the system using a certain set of data only to test it with a foreign one later on. Rather, we embed it in an environment that is biased towards a certain task, i.e., we *present it with the information that adding is advantageous*. Also, we provide numbers in the input buffer of each CPU that the tierran cells may choose to manipulate, but nothing more. While the cells eventually learn to add just these numbers, these may be exchanged with *any* other numbers at *any* given time. Thus, the cells truly learn the concept, not just an instance.

Our main tool to bias evolution towards accomplishing the chosen task is the distribution of bonuses in the form of extra time [cf. Eq. (8)]. We reward three accomplishments which are formulated in as general a manner as possible so as not to bias towards any particular solution to the problem. The first step consists in rewarding cells that develop the correct input/output structure for the problem at hand. Clearly, adding requires a minimum of two inputs and one output. As a consequence, any cell that develops a minimum of two `get` and a minimum of one `put` command receives a certain bonus at the time of gestation of an off-spring (Table 2 lists the specific bonuses used in the experiments presented here). The next step is "clearing the channels": we reward cells that manage to echo the values in the input buffers into the output buffers. Finally, any cell that writes a value into the output buffer that happens to be the sum of the two previously read values is rewarded with extra time at the time of executing the successful `put` command. Note that any such bonus increases the fitness of such a cell according to (10) resulting in more off-spring for that cell and a subsequent perpetuation of the discovery.

Table 2
Distribution of bonus for evolved features. A negative bonus indicates that this number of instructions is *subtracted* from the default allocated time-slice if this feature is *not* evolved.

feature	bonus
input/output	−50
echo	40
add	100

The rewards are of course available simultaneously and can in principal be discovered in any order. This reward-structure, "soft-coded" into the instruction set[10], constitutes the "fitness-landscape" with valleys, mountains, and ridges, that the soup has to adapt to in order to thrive.

In the simulations presented here the environment is extremely simple, with only three distinct explicit bonuses. However, they can be combined in different ways, and two of them can (in the present simulations) be repeated up to three times to gain additional bonus. Also, there is only a limited number of ways for a cell to reduce its gestation time (resulting in higher fitness). The introduction of the leanness factor f on the other hand already provides for a means to improve fitness in a quasi-continuous way (up to $f = 1$). Furthermore, cells can exploit the structure of the population itself to gain fitness, a feat most impressively demonstrated by the parasites (sections of code that cannot reproduce on their own, but rather use the copy-loop of a host cell to produce off-spring). In all these instances of fitness-improvement, information is "found" by a cell (through mutation) and used to gain an advantage. This information is then reflected in the genome of the adapted cell.

[9] An attempt at solving the XOR problem using tierra is described in Ref. [11].

[10] We distinguish between the "hard-coded" part of the instruction set, which is the same ("universal") for any problem (and could just as well be etched into silicon) and the "soft-coded" part, which is specific to the problem at hand, and thus represents part of the "physical" environment that the cells live in.

Clearly, the path that evolution takes in order to achieve a given task depends on the fitness landscape, and thus on the bonus structure. In the simple example presented here, the solution will undoubtedly reflect the particular bias structure we chose. However, this is not uncommon in natural systems, nor is it unnatural to reward tasks that are not directly associated with the final task achieved (such as in this case "echoing") [11]. In natural systems, the population takes advantage of every bit of information available, and of course adaptation is *not* active, but rather passive. It is beyond the scope of this article to investigate the dependence of the learning capabilities of the system on the bonus structure. Surely the paucity of rewards available guides evolution towards a certain (natural) algorithm, but it is "passive guiding", not "leading". It is uncertain whether our selection of the bonus structure precluded the discovery of a larger diversity of algorithms. Considering the simplicity of the problem, however, we suspect that an optimal algorithm was developed, even though its implementation varied tremendously from run to run. In the appendix, we present a random selection of algorithms that evolved in these simulations. From these it appears that while naturally there are similarities in the solutions, the variety is impressive and some solutions genuinely insane.

Even though the environment for the adding problem is extremely simple, the space of possible fitness improvements appears to be extremely large. Since every genotype has a specific fitness, we can think of the space of possible fitnesses pertaining to the problem as meta-stable states in a continuum of fitness states while transitions between these states are driven by mutations. Since the number of meta-stable states is already very large for this simple example (and should effectively be infinite in any realistic system) the tierran system will exhibit features of a self-organized critical (SOC) system [12]. As a consequence, the time between transitions (or "avalanches" in the language of SOC systems) is distributed according to a power law (as are the sizes of fitness-jumps) and thus an "average time between transitions" (which would allow a determination of the learning rate) cannot be defined. Moreover, this suggests that the learning curve (Fig. 2) of such a system would have a fractal appearance, i.e., would appear similar at all scales (Devil's staircase).

As is well-known [12], a power-law distribution of waiting-times is due to an absence of scales in the problem. This is true to a certain extent in tierra since there is no time scale of the order of the time scale of learning (or evolution), nor is there a scale setting the size of the avalanches [13]. There are however microscopic time scales [those of Eq. (18)] and these lead to a violation of power-law behaviour. It is precisely the presence of these scales that leads to a correlation between $\langle t_m \rangle$ (the average time between mutations) and the learning rate. The latter is still notoriously difficult to define. One approach would be to determine the average time taken to learn the specified task at a fixed mutation rate, yet the measured times scatter heavily around the average due to the stochastic nature of the learning process. In principal there is no guarantee that in any specific simulation the goal will be attained (i.e., the learning time is in principle infinite) while in practice the goal is (at large enough mutation rates) almost always attained (see Table 3). On the same token, it is impossible to predict the sequence of meta-stable states that the soup will traverse to reach the maximum fitness (pertaining to this problem). As a consequence, the end product (i.e., the most successful genotype) will very seldom look similar even for two runs with exactly the same starting conditions (except for the random seed) and thus exactly the same "environment". This is strong evidence for contingency in the learning process for auto-adaptive ge-

[11] A well-known paradigm is the evolution of the mammalian ear out of jaw bones. Obviously the capability to chew is unrelated to the capability to hear, but the reward to develop the former contributed to the development of the latter. Similarly, the particular bonus structure for the development of the ear clearly is reflected in its design.

[12] A full investigation of self-organized criticality in tierra is outside of the scope of this paper and will be reported elsewhere [13].

[13] This is only approximately true in the simulations presented here, as the paucity of rewards (see Table 2) does set a scale at large fitness-jumps.

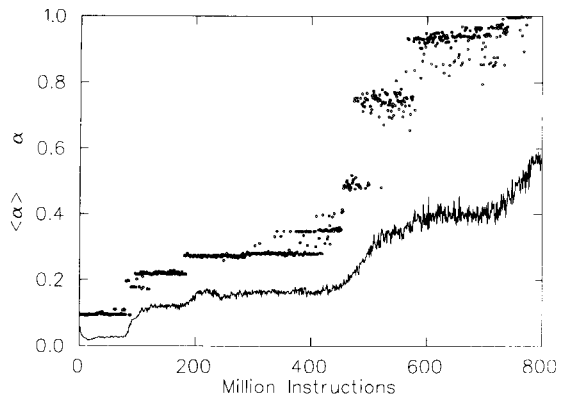

Fig. 2. Learning curve for a simulation with $R = 1.33 \times 10^{-8}$ mutations per site per instruction executed, in a soup of size 131072 instructions (a mutation probability of $Rs \approx 1.75 \times 10^{-3}$). The circles show the fitness α of the "best-of-population" while the lower curve shows the average fitness $\langle\alpha\rangle$ of the population.

netic systems, and possibly for evolutionary processes in general.

Fig. 2 shows the evolution of fitness (the learning curve) in a typical run at an intermediate mutation rate, specifically $R = 1.33 \times 10^{-8}$ with a soup-size $s = 131072$, which translates into an average time between mutations of $\langle t_m \rangle = 1/Rs \approx 572$ instructions. The upper circles denote the fitness α (defined in Section 4) of the "best-of population" (the genotype with the most living copies) every million instructions, which translates roughly into every three generations. The lower curve shows the average fitness of the population $\langle\alpha\rangle$. As expected, the fitness-of-the-best increases via jumps indicating transitions between meta-stable states. These are most likely first-order phase transitions as is evident from the coexistent phases. (A detailed investigation of the statistical mechanics of this system will appear elsewhere).

The first transition in Fig. 2 (at around $t = 80$ million executed instructions) is in fact due to the unrolling of the loop mentioned earlier, which literally halves the gestation time of the cell. Consequently, α jumps by roughly a factor 2. The transition at $t = 100$M involves a minor rearrangement of code, while at $t = 185$M the copy-loop is unrolled to $m = 3$. The input/output structure (first bonus in Table 2) is achieved around $t = 290$M but appears as only a small increase in fitness. This is due to the bonus being distributed over several sweeps, which entails that the average gain per gestation-period is rather small. Echoing is learned at 340M (this time is defined as the time when a cell that discovered echoing dominates the population for the first time). The transition at $t = 409$M simply makes echoing more efficient, and prepares the ground for the transition at $t = 453$M, when the best-of-population simultaneously triggers the bonus for adding *and* echoing. This is not a rare scenario, as cells often first develop the capacity to echo twice in a gestation period thus earning a bonus of 80, only to transform one of the echoing sections of code into an adding one. The later transitions simply accumulate echo's and add's (mainly by splicing together sections of code containing the pertinent sequence) so as to trigger the maximum bonus.

The complexity of the cell dominating the population at around $t = 800$M is intriguing. Not only has it evolved the capacity to successfully manipulate the numbers in the input buffers by adding them several times per gestation period, but it also optimized its reproduction loop to gestate off-spring three times faster than the ancestor[14]. While the cells will always attempt to do the latter, we could have chosen to reward an entirely different task, and consequently the final genotype would reflect that in its genome instead. In fact, after the cells learn to write the content of the input buffers into the output buffers, an inspection of the output buffers of all coexisting cells at that moment shows that all kinds of operations are performed on these numbers. The majority of the cells return the input-numbers untouched so as to trigger the "echoing" reward, some however subtract them, add all three, subtract the number 4, and so forth. The reward structure simply weeds out those cells with mutations that allow them to add two numbers out of the zoo of creatures that perform a litany of tasks, entirely accidentally. In this sense, the actual nature of the task is irrelevant for the general characteristics of learning in the tierra system.

[14] With our current version of tierra, 800M instructions are on average reached after 5 hours of CPU time on an HP 9000/750 workstation.

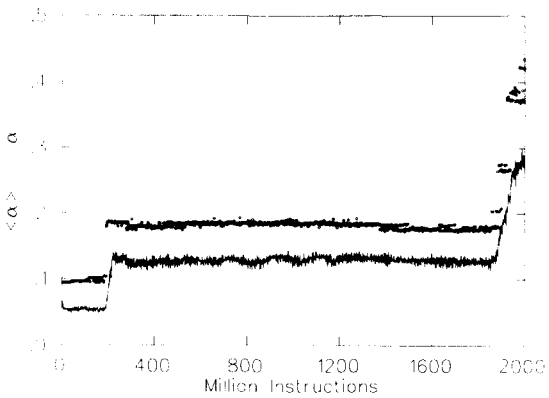

Fig. 3. Learning curve for a simulation with $R = 0.67 \times 10^{-8}$, same soup size as in Fig. 2. This run shows a long plateau at $\alpha \approx 0.2$ indicating trapping of the population in a meta-stable state. Note the difference in time and fitness scale as opposed to Fig. 1.

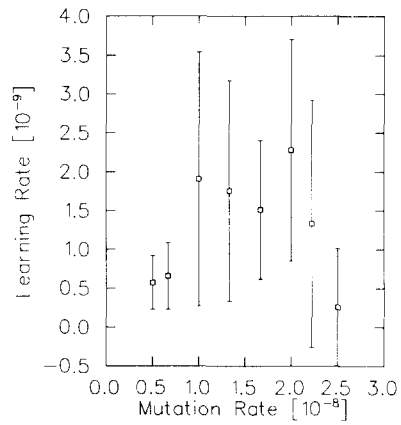

Fig. 4. Average learning rate as a statistical average of inverse learning times versus mutation rate. The error bars are 1 σ standard deviation.

We have performed this type of experiment ten times for each of eight different mutation rates, at a constant soup size. The mutation rates were chosen to range from very low, where adding is achieved only very late (if at all), to very high rates where the population effectively "melts" (ceases to reproduce). This happens at around the point where $t_g > \langle t_m^* \rangle$ as derived earlier, i.e., when on average a cell is hit by a mutation before it can generate its first off-spring. Clearly then, a cell cannot on average propagate its genome, and the information contained in it. For each mutation rate, the learning time fluctuates strongly due to the statistical nature of the learning process and to the presence of meta-stable states in the system that can trap the population. The time it takes for the population to escape such a trap then determines the learning time. In most such cases we were unable to wait long enough to see this happen. Fig. 3 shows a learning curve for half the mutation rate in Fig. 2, where the population was stuck in a meta-stable level of fitness $\alpha \approx 0.2$, before breaking out of it at around t = 1900M and learning to add almost instantly after. The time it takes to escape such a state should be considerably reduced by choosing a larger soup size, which would allow for a more heterogeneous population exploring different regions of the landscape at the same time. In tierra, the soup size (reserved memory space for cells) cannot easily be enlarged past a certain size, which entails that the population can equilibrate into a homogeneous phase rather easily. Consequently, it is important to investigate learning characteristics for different soup sizes.

Table 3 shows the result for all 80 runs, used in Fig. 4, at soup size 131072 instructions [15]. The number of cells in such a soup is variable, but is of the order of magnitude of 800 cells of length 80.

Each column in Table 3 contains the learning times (defined as the time a genotype that successfully adds first dominates the population, i.e., has the most living copies in the soup), or in case adding was not achieved, the time the simulation was interrupted, preceded by a "greater than" symbol. Each run was taken to a minimum of 2 billion instructions executed. An entry "e.c." implies that the run died soon after inoculation due to the error-catastrophe.

Averaging the inverse learning times (learning rates) from each column of Table 3 yields Fig. 4 [16]. As expected, the scattering of the data, implying large standard deviations, does not allow for definite conclusions on the behaviour of the learning rate. However, we can define a "learning fraction" by determining the fraction f_X of runs at a given mutation rate that achieved learning at a time $t < t_c = X$, where t_c is a cut-off that reflects the time-scale of learning in this environment for this task. As an example, the learning fraction with cut-off 1000 (million) at

[15] For technical reasons, the soup size has to be a power of 2.

[16] Runs that did not achieve learning were given an infinite learning time, i.e., a learning rate of zero.

Table 3
Learning times (in million instructions executed) for mutation rates from 0.5×10^{-8} to 2.5×10^{-8} for soup size $s = 131072$ instructions. An entry preceded by a "greater than" sign signifies that the task was not learned before that time and the run was interrupted. An entry "e.c." means that the population ceased to reproduce due to the "error catastrophe" as mentioned above.

R (10^{-8})	0.5	0.667	1.0	1.333	1.667	2.0	2.222	2.5
#1	1586	843	242	453	1052	836	e.c.	e.c.
#2	2802	1765	196	1601	606	1414	e.c.	397
#3	1220	696	995	263	>2002	240	293	e.c.
#4	1413	>3144	1201	>2026	586	596	>2022	e.c.
#5	1136	>2023	1041	407	406	507	343	e.c.
#6	>2025	1927	>2019	357	380	270	809	e.c.
#7	>2090	1922	330	520	625	327	225	e.c.
#8	901	1273	442	271	624	587	e.c.	e.c.
#9	1353	1374	>2251	>2094	>2117	e.c.	e.c.	e.c.
#10	2252	1233	581	>2087	406	222	803	e.c.

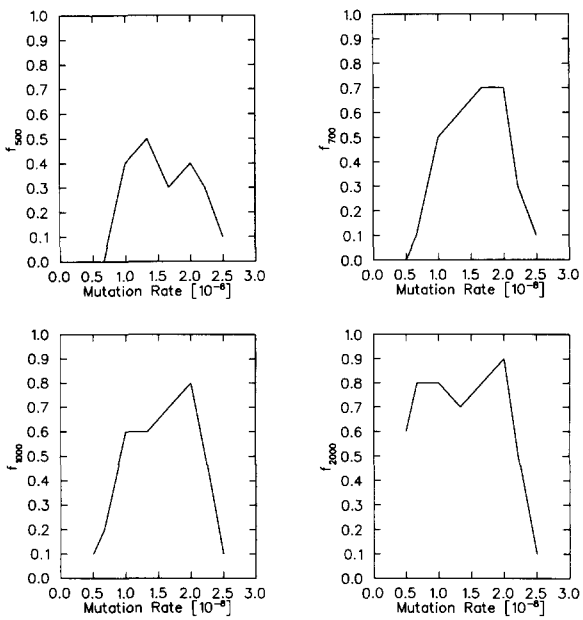

Fig. 5. Learning fractions f_X as defined in the text versus mutation rate for various cutoffs. (a) Cutoff $X = 500$ million instructions, (b) $X = 700$, (c) $X = 1000$, and (d) $X = 2000$.

mutation rate $R = 1.0 \times 10^{-8}$ is $f_{1000}(1.0) = 0.6$ i.e., six out of ten runs resulted in a population that successfully added before 1000 million instructions were executed. This procedure allows us to obtain the curves presented in Fig. 5.

Clearly, choosing the cut-off scale too low would not reflect the learning characteristics of the soup, and neither would a high choice. In fact, this procedure implicitly determines simultaneously the window in mutation rate when learning is most effective, *and* an estimate for the time-scale of learning for this particular system and task.

Choosing the cut-off time scale (in units of million instructions executed) to be $700 < t_c < 1000$, the behaviour of the learning fraction in Fig. 5 suggests that learning becomes more and more effective as the mutation rate is increased up to a point where the soup dissolves as a result of the error catastrophe. The time-scale for learning determined here, however, is certainly not universal but depends on soup-size, initial creature, and bonus structure. These dependencies will be investigated in the near future.

6. Conclusions

The evolutionary learning displayed by the artificial system presented here has a number of fascinating characteristics that it may share with the natural genetic system that gave rise to bacterial DNA. As a learning system for practical applications tierra falls short in many respects, as must any auto-adaptive system at this stage. More complicated tasks than the one analyzed here will require a more elaborate information environment. As in animal husbandry, the breeding of a specific trait may require a certain amount of experimentation with the reward structure, and success cannot be guaranteed. Clearly, those tasks that are easily transcribed into the instruction set used will

develop most easily. We hope to in the future attack problems that require a far more complex information landscape. We have little doubt that this complexity will ultimately be reflected in the genome of the adapted population. On the other hand, we fully expect limitations imposed by the instruction set, and the linear execution of code as opposed to the typically parallel approach of nature.

We have tried in this paper to extract universal characteristics of the auto-adaptive learning process, characteristics that should be reproducible by any software that incorporates the basic ingredients for auto-adaptive systems described earlier. Naturally, besides the universal characteristics there are features that must depend on the specifics of the implementation; it is the investigators task to isolate them. Particularly, the size and dimensionality of the tierran soup, i.e., the physical memory that the cells inhabit, has an influence on the global, and critical, behaviour of the population. In its present configuration, the tierran soup is multi-dimensional (even though in physical memory each cell has only two neighbours) because each cell can potentially interact with every other one via the reaper queue, making the system non-local. This may be the most important limitation of tierra as it affects growth, and information transfer through the population, which in turn determines equilibration times, diffusion coefficients, and self-organized behaviour. These aspects will be addressed for a manifestly two-dimensional genetic system with local interactions in the near future [7].

Acknowledgement

I would like to thank Steve Koonin for the initial suggestion that led to this work, as well as for continuing support and encouragement. This work was supported in part by NSF grant # PHY91-15574 and by a Caltech Divisional fellowship.

Appendix A

In this appendix we present a few selected genomes of evolved successful algorithms, and display the section of the genome that is related to the "phenotype" of interest: adding. Each such genome is identified by the code assigned to it by the genebanking system, which consists of a four digit number indicating the size of the genome, and a three letter code which is unique for every run, but not across runs. We give the mnemonic code of the instruction as well as the instruction number. We only display those instructions that are directly responsible for the input, output, and manipulation of numbers leading to the successful accomplishment of the task. Below is a description of the commands used by the algorithms. Note that the commands `put` and `jmp` differ from Ray's commands only in the selection of registers.

`get` reads a number in the input buffer and transfers it into the DX register.

`put` writes the contents of the CX register into output buffer and clears the CX register.

`add` adds contents of DX register to contents of CX register and writes result into CX register (CX + DX $\rightarrow$ CX).

`sub` CX - DX $\rightarrow$ CX.

`inc` increases CX by one.

`dec` decreases CX by one.

`pushx`X pushes contents of xX register onto stack.

`popx`X pops top of stack into xX register.

In Table A.1 we display the skeleton genome of a number of algorithms that we extracted from the genebank ("fossil record"). The provenance of the cell is denoted by the inverse mutation rate and run number (as defined in Table 3), with the time of origin of the genotype appended (in units of millions of executed instructions) in parentheses. Thus, 0118aay [1.33/1 (451)] denotes cell 0118aay of run number 1 of the batch run at mutation rate $R^{-1} = 1.33 \times 10^{-8}$, born at 451M. All cells displayed were the first "adders" to dominate the population in their respective runs, thus those that triggered a major phase transition. Of the 54 "first-adders", the following three were chosen at random.

Table A.1

0118aay [1.33/1(451)]	**0127ara** [0.5/2(2799)]	**0070chw** [2.0/1(832)]
003 add	002 inc	001 pushdx
004 get	004 dec	005 get
009 add	016 pushdx	007 popcx
013 dec	017 popcx	008 add
015 inc	018 get	009 put
016 put	022 add	047 get
024 get	025 sub	048 pushdx
026 pushdx	027 add	049 popdx
033 popcx	030 put	
034 put	101 get	
075 get		

Cell 0118aay is the first adding creature of the run displayed in Fig. 2, responsible for the phase transition at $t = 453$M. The first command 3 add adds DX to CX, which in case CX is empty is an effective way of transferring the contents of DX into CX. Since this command is at the beginning of the cell, this is usually the case. However, note that 16 put results in correct addition only when the program has run once, loading the DX register with 75 get. Otherwise (first run of the program) this sequence results in echoing. Note also that even though the sequence (13,15) is superfluous, it is most likely going to survive because single mutations will only destroy one of the instructions, resulting in incorrect addition. The following sequence (24,26,33,34) is a nearly perfect echoing sequence. Without 75 get, the algorithm displayed echoes twice per gestation period to record a bonus of 80. We suspect that 0118aay developed from such a cell simply by turning one of the neutral instructions just before the copy-loop into get. Unfortunately the "fossil record" is not extensive enough to verify this.

Cell 0127ara is a creature that developed much "later", from a run with lower mutation rate. Interestingly, a strategy similar to the one employed by 0118aay developed, where in the first run through the program only "echoing" is achieved, until the DX register is loaded with 101 get. The pair (2,4) again is superfluous, after which the contents of DX are shifted into CX by the pair (16,17). After a surprising add/sub/add the result is written into the output with 030, at which time the bonus is earned.

Cell 0070chw was formed in a run with high mutation rate. We leave it to the reader to discover its strategy.

Finally, we would like to display the skeleton genome of 0071bno[2.0/2(1754)]. Since its algorithm is more complicated than the ones encountered so far, we have to describe more commands.

`call` jump to the complementary pattern of the template immediately following the command and push the address of the instruction immediately following the template (not the complementary template) onto the stack. In case of absence of template, push address of instruction following command on stack, but do not jump.

`jmp` jump to the complementary template of the pattern just following the command, or, in the absence thereof, jump to the address stored in the BX register.

The commands `call` and `jmp` are used by this cell to create a devious adding-loop which is run through five times, performing the task each time. In Table A.2 we display the complete segment of genome from instruction 43 to 66.

Instructions 43–55 constitute the copy-loop. Earlier commands have placed the address of the mother in the BX register, the address of the daughter in AX, and the length of the mother in CX. `movii` writes the instruction at address BX + CX to location AX + CX. Thus, the mother is copied into the daughter starting with the last instruction of the mother, moving to the beginning of the mother with the sequence `dec movii`

Table A.2

0071bno [2.0/2(1754)]

43	nop1	56	add
44	nop0	57	pushbx
45	dec	58	divide
46	movii	59	get
47	ifz	60	add
48	jmp	61	nop1
49	nop1	62	popax
50	get	63	popbx
51	dec	64	popbx
52	movii	65	put
53	call	66	jmp
54	nop0		
55	nop1		

ifz. The latter of the commands checks whether CX is zero and executes the next command if it is, but skips it otherwise. This checks whether copying is complete and exits the loop in that case. Command 50 get serves no apparent purpose, but since it continually reads the input into DX has either once served a purpose, or can be used fruitfully in the future. Instructions 51, 52 accomplish the copying of two instructions per iteration of the copy-loop, the result of an "unrolling" operation as described in the text.

The intriguing command is 53 call. Indeed, this is used to return to the top of the copy-loop (instruction 43), but at the same time the address of the command following the template, 56 add, is pushed on the stack. Since this is executed 36 times during the copy-procedure, the stack (of size 10) is filled with the address of this instruction. Once the mother is fully copied into the daughter, 48 jmp is executed which jumps to instruction 55, the instruction following the template nop0, which is the complimentary template to nop1 which followed the jmp command (incomplete templates are recognised). Then, the contents of the DX register are transferred to CX via 56 add, as the CX register was empty since all the copying was finished. Following is 57 pushbx which pushes the mother's beginning address onto the stack (this will become important later on), while command 58 divide releases the daughter and terminates the mother's write access to the daughter's space. Instructions 59 get and 60 add place the sum of the two most recently read values into CX. Important is the following 64 popbx command, since it places the address of instruction 56 into BX (by popping it off the top of the stack). After 65 put, which triggers the adding-bonus, the subsequent 66 jmp command, since it is not followed by a template, jumps to the instruction whose location is in the BX register, i.e. 56 add! Since DX still contains the last read input value, it is transferred to CX via 56 add since again CX is empty (by virtue of the put command which clears CX). After get add, the stack is popped three times again, which together with 57 pushbx (which now also pushes the address of 56 add on the stack, unlike the first time it was invoked) makes a total of two pops per run through this unique adding-loop. As the stack is only of size 10, this loop can be run through five times until the stack *cycles* (the bottom of the stack becomes the top of the stack) and the top of the stack is the mother's *beginning address*, which was placed on the stack by the first execution of 57 pushbx. Thus, after 5 iterations of the adding-loop, 64 popbx places the mother's beginning address in BX, such that the following 66 jmp moves the instruction pointer to the beginning of the cell, to restart the whole process.

The evolution of such an elaborate scheme, while manifestly "insane", does encourage the belief that more complicated tasks can be bred using the auto-adaptive scheme.

References

[1] M. Anthony and N. Biggs, Computational Learning Theory (Cambridge University Press, Cambridge, 1992).

[2] D.E. Rumelhart, J.L. McClelland and the PDP Research Group, Parallel Distributed Processing, Vol. 1 (MIT Press, 1986).

[3] J. Holland, Adaptation in Natural and Artificial Systems, second Ed. (MIT Press, 1992).

[4] D.E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning (Addison-Wesley, 1989).

[5] S.A. Kaufman, The Origins of Order (Oxford Univ. Press, 1993).

[6] T.S. Ray, in: Artificial Life II: A Proceedings Volume in the Santa Fe Institute in the Sciences of Complexity, Vol. 10, C.G. Langton, J.D. Farmer, S. Rasmussen and C. Taylor, eds. (Addison-Wesley, Reading, MA, 1992) p. 371; Artif. Life 1 (1994) 195; Physica D 75 (1994) 239–263.

[7] C. Adami, C.T. Brown and C. Ofria, in preparation.

[8] Documentation of the tierra software (unpublished).
[9] T.S. Ray, private communication.
[10] M. Eigen, J. McCaskill and P. Schuster, Adv. in Chem. Phys. 75 (1989) 149.
[11] W.A. Tackett and J.-L. Gaudiot, in: Proc. of Int'l Conf. on Neural Networks (Beijing, 1992), in press.
[12] P. Bak, C. Tang and K. Wiesenfeld, Phys. Rev. Lett. 59 (1987) 381; Phys. Rev. A 38 (1988) 364.
[13] C. Adami, Self-organized criticality in living systems, KRL preprint MAP-167, Caltech (December 1993).
[14] C.G. Langton, J.D. Farmer, S. Rasmussen and C. Taylor, eds., Artificial Life II: A Proceedings Volume in the Santa Fe Institute in the Sciences of Complexity, Vol. 10 (Addison-Wesley, Reading, MA, 1992).