Supporting Online Material for
Organism Size Promotes the Evolution of Specialized Cells in Multicellular Digital Organisms

Martin Willensdorfer

Program for Evolutionary Dynamics,
Department of Molecular and Cellular Biology,
Harvard University, Cambridge, MA 02138, USA.
E-mail: ma.wi@gmx.at

Contents

1 Digital Self-Replicating Cellular Organisms (DISCOs)

This section provides details about DISCOs. Since I was motivated by Lenski *et. al.* (1), DISCOs are very similar to Avididans, that is, digital organisms developed by Christoph Adami (2). However, there are some important differences: (a) DISCOs have a cell cycle with a metabolic and a replication phase, (b) DISCOs have a minimal and disjunct set of instructions for each phase, (c) DISCOs can only copy their genome to a daughter strand and cannot modify their own genome, and (d) DISCOs can evolve multicellularity. Modifications (a)–(c) make it possible to directly identify the genomic basis of a phenotypic feature and modification (d) makes it possible to study the evolution of multicellularity.

Section 1.1 explains in detail how a single DISCO cell works. Section 1.2 describes how fitness is realized in DISCOs and how a DISCO can increase its fitness by computing logic functions. Finally, Section 1.3 explains aspects of multicellularity in DISCOs.

1.1 The Cell

A DISCO cell is a computing automaton with the ability to replicate. Each DISCO has a genome, that is, a sequence of instructions and modifiers. The instructions are designed to operate on the automaton and change its state in certain ways. This state change depends not only on the instruction but also on modifiers in the genome. For the following, however, I will use the term instruction to refer to an instruction and all the modifiers in the genome that affect the action of this instruction. Appendix A contains detailed information about how modifiers affect each instruction. In DISCOs modifiers are also used to encode multicellularity in the genome as we will see in Section 1.3.

Naturally, the execution of a sequence of instructions leads to a sequence of state changes of the automaton. The right sequence of instructions can cause state changes that result in the computation of logic functions, genome replication, and, finally, cell division. Which sequence of instructions a cell will actually execute is determined by its genome. Hence, the genome determines the ability of a cell to compute logic functions as well as speed and accuracy of replication, which constitute a cells phenotype.

The life of a DISCO cell is divided into two phases, a metabolic and a replication phase. At the onset of each phase the automaton is reset and starts to execute instructions from the beginning of the genome. During the metabolic phase a cell can read and manipulate data to compute logic functions. During the replication phase a cell can copy its genome and initiate cell division. Each phase has its own set of instructions. We have a set of instructions for the metabolic phase (`blank`, `io`, `nand`, `swap`, `push`, `pop`) and a set of instructions for the replication phase (`copy`, `search`, `mov-head`, `if-label`, `divide`).

A newly created cell enters first the metabolic phase. The transition from metabolic to replication phase happens after the execution of m metabolic instructions (m is a simulation parameter and equals 300 in all described simulations). The replication phase, on the other hand, ends at the successful or unsuccessful attempt to initiate cell division, that is, at the execution of a `divide` instruction.

During the metabolic phase a DISCO cell can create data in the form of logic compounds. To make the production of logic compounds possible the automaton is equipped with **three registers** (A, B, and C) and one **stack**. The registers are the operating platform. They are used to store and manipulate data. New logic compounds can only be generated by using the `nand` instruction which manipulates data stored in the registers. The most basic logic compound is a logic variable and every logic compound is composed of logic variables and logic operators that connect these variables. Logic variables are supplied by the `io` instruction and stored in one of the three registers. By loading a new variable to a register, the `io` instruction might have to replace a logic compound that is stored in this register. Whenever a logic compound is replaced, the `io` instruction checks whether this compound is equivalent to one of nine logic functions. If this is the case, then the DISCO will be rewarded (see next section). Data can also be copied from one of the registers to the stack with the `push` instruction and recovered with the `pop` instruction. The DISCO reads the metabolic instructions one by one from the genome. If the end of the genome is reached, the DISCO will continue from the beginning, that is, the genome is processed circular.

One might wonder how those activities relate to a biological metabolism. The analogy becomes quite obvious if one considers autotrophic processes like carbon fixation in plants. Plants convert carbon dioxide into organic compounds. The first stable intermediate is a 3-carbon compound. These trioses can be condensed into hexoses, sucrose, or cellulose. They can also be used to make amino acids and lipids. Hence, more complex organic compounds are assembled from simpler ones and all those compounds contribute to the fitness of the organism. The situation is analogous in DISCOs. However, a DISCO handles logic compounds instead of organic ones. Logic variables are assembled with the `nand` instruction to form more complex logic compounds and, as with real organisms, certain compounds can contribute considerably to the fitness of the organism.

The metabolic phase is followed by a replication phase. During the replication phase a DISCO can copy its genome and initiate cell division. The `copy` instruction instructs the automaton to copy a genome element to the daughter strand. Which element of the genome is copied depends on the position of the so called **read head**. Heads are pointers that the automaton can use to mark elements of the genome. Besides the read head the automaton has an **instruction head** and a **flow head**. The read head shows the automaton which element of the genome has to be copied next. It moves forward to the next element after each copy event. The instruction head shows the automaton which instruction has to be executed next and does also move forward after the instruction has been executed. The flow head is used to mark positions in the genome to which either the instruction head or the read had can be moved by the `mov-head` instruction. This together with the `if-label` instruction allows to encode jumps and loops in the genome. Cell division is initiated by the `divide` instruction. The execution of a `divde` instruction will always lead to a switch from the replication phase to the metabolic phase, regardless of whether it was successful or not.

Appendix A describes precisely how each instruction changes the state of the automaton and how modifiers can affect this state change. Appendix B contains some example genomes and illustrates how a sequence of instructions can encode logic functions and cell replication.

1.2 Environment, Fitness, Merit, and Logic Functions

DISCOs live in an unstructured environment that can accommodate a given number of organisms. In this environment DISCOs compete for the opportunity to execute instructions. For each iteration only one DISCO is selected and the selected DISCO can execute only one instruction. Most of the times the execution of an instruction will only affect the DISCO internally. At some point, however, a DISCO will produce offspring. The offspring is first exposed to replacement (copy), insertion, and deletion mutations and then placed at a randomly chosen position in the environment. In most cases this will result in the replacement (death) of another DISCO.

The frequency with which DISCOs are chosen to execute an instruction is proportional to their merit. Consequently, the more merit a DISCO has, the more frequently it will be chosen to execute an instruction. Obviously, if the number of instructions that have to be executed until a cell divides is the same for two DISCOs, then the one with the greater merit will have the higher fitness. On the other hand, if two DISCOs have the same merit, then the one that has to execute less instructions until cell division will have the higher fitness. We see that there are two components that determine the fitness of a DISCO: (a) its merit (relative to the merit of the other organisms) and (b) the speed of replication (i.e., the number of instructions that have to be executed until a cell divides).

As mentioned above, a DISCO can manipulate data to create logic compounds. By creating compounds that are equivalent to specific logic functions, a DISCO can increase its merit and consequently its fitness. A DISCO can only generate logic compounds by applying the `nand` instruction. The `nand` instruction connects two data elements (a data element is either a logic variable or a logic compound) with the NAND ("Not And") operation. The NAND operator (. | .) is defined as the negation (¬ .) of the conjunction (. ∧ .), that is, $(a|b) = \neg(a \wedge b)$. The resulting logic compound is false if and only if both a and b are true. Formal logic shows that every truth-functional compound can be expressed by using just the NAND operator (3). For example, we can write $\neg a = a|a$ and $a \wedge b = (a|b)|(a|b)$. Since a DISCO can apply the NAND operation, it can also produce a multitude of logic compounds. With the `io` instruction a DISCO can test whether a data element is equivalent to one of up to nine logic functions. If so, then the DISCO is rewarded with a merit increase. By how much the merit is increased depends on the complexity of the logic function (see below and Section 2). Hence, a DISCO receives a reward for the construction of meaningful logic compounds. Selection will therefore favor DISCOs that can construct such logic compounds.

For simplicity, the merit of an organism remains unchanged until it produces offspring. Each time a DISCO produces new offspring its merit is newly calculated according to the logic compounds it was able to create. One has to keep in mind that newly born DISCOs have not yet had the chance to construct any logic compounds. To avoid disadvantageous for newborns, they start off with the parental merit until they reached maturity and produce their own offspring.

The table below lists the nine logic operators that a DISCO can compute to change its merit. During a simulation logic operators are identified by calculating the truth value of the expression in the second column, where C is a logic compound that a DISCO generated. The truth value of the expression in column two is calculated by using randomly generated 64-bit integers as instances for the logic variables a and b. The NAND operation as well as the logic operations are applied bit-wise. An example is given below.

It is convenient to summarize which logic functions the genome of a DISCO can compute by using a nine digit binary code. For example a 000000000 tells us that the genome does not encode for any logic function; a 101101000 that the genome can compute NOT, AND, OR_N, and AND_N, and a 111111011 that the genome can compute all nine logic functions except for XOR.

logic operator	definition[1]	minimum number of NAND required
NOT	$C(a) \equiv \neg a$	1
NAND	$C(a,b) \equiv \neg(a \wedge b)$	1
AND	$C(a,b) \equiv a \wedge b$	2
OR_N	$\Big(C(a,b) \equiv a \vee \neg b\Big) \vee \Big(C(a,b) \equiv \neg a \vee b\Big)$	2
OR	$C(a,b) \equiv a \vee b$	3
AND_N	$\Big(C(a,b) \equiv a \wedge \neg b\Big) \vee \Big(C(a,b) \equiv \neg a \wedge b\Big)$	3
XOR	$C(a,b) \equiv (a \vee b) \wedge \neg(a \wedge b)$	4
NOR	$C(a,b) \equiv \neg(a \vee b)$	4
EQU	$C(a,b) \equiv (a \wedge b) \vee (\neg a \wedge \neg b)$	5

a	: `110101110001110011111001100010001000001000`...	64-bit integer
b	: `000100101101101001011100000111110000101000`...	64-bit integer
$C(a,b)$	: `000100100001100001011000000010000000001000`...	64-bit integer

We have $C(a,b) \equiv a \wedge b$ and conclude that compound $C(a,b)$ encodes for AND

1.3 Multicellularity

So far I have described how the genome of a DISCO determines the phenotype of a single cell. The genome can also encode information about the development of a multicellular organism.

[1] $\neg$, $\wedge$, $\vee$, and $\equiv$ symbolize the logical negation, conjunction (and), disjunction (or), and equivalence.

A DISCOs life starts always with a single cell, the default (D) cell. Each organism has exactly one D cell and the D cell is the only reproductive cell. After the first cell division the D cell has two options. It can either release the daughter cell into the environment as offspring and remain unicellular, or retain the cell as a first step towards the development of a multicellular organism. If, how many, and what kind of cells are retained is encoded in the genome.

To understand how multicellularity is encoded, we have to remind ourself that the genome is a sequence of instructions and modifiers. Among the set of instructions, the `divide` instruction is special because it is the only one that is not affected by modifiers. To keep things as simple as possible, I decided to exploit this feature of the `divide` instruction. In particular, the D cell will retain one cell for each `divide` instruction in the genome that is followed by a modifier. Depending on the kind of modifier the retained daughter cell is assigned to a somatic cell type. For example, a '`divide|A`' encodes for a X cell, whereas a '`divide|B`' and a '`divide|C`' encodes for a Y and a Z cell, respectively. (Z cells are not relevant for this work and just mentioned for completeness.) After a daughter cell has been retained for each such `divide` instruction, the D cell is able to release daughter cells as offspring into the environment (see Figure 1 for an example). Please note that the genome might not contain any `divide` instruction that is followed by a modifier and would therefore encode for a unicellular organism (see Appendix B for examples).

Most multicellular organisms have specialized cells. Even though the specialized cells of a multicellular organism contain in most cases the same genome as the replicative cells, they behave differently. Specialized cells in DISCOs work analogous. The genome of a cell might be able to compute all logic functions, but cells can only utilize logic functions according to their cell type. For this work, D and X cells can only benefit from the first six logic functions and Y cells only from the last three functions that might be encoded in the genome (see Section 2).

It is important to point out that, even though a DISCO can be composed of several cells, each cell is still an independent automaton that executes one instruction after another. In fact, whenever a multicellular DISCO is selected by the environment to execute an instruction, each cell of this organism will execute one instruction.

2 Details about the -X and +X simulations

This section describes the -X and the +X simulations in more detail. For computational reasons I limited the population size to 200 (uni- or multicellular) DISCOs. I conducted 500 runs for each type of simulation, which differed only with respect to the seed for the random number generator. Both types of simulations have two parts (a) an initial 10 000 generations (2×10^6 replication events) in which specialized cells are not beneficial and (b) a further 10 000 generations in which specialized cells are advantageous.

For the first 10 000 generations, each simulation was initiated with the same genome. This genome contains 141 '`blank`' instructions (essentially a place holder, see Appendix A) and a sequence of 9 instructions, '`search|copy|if-label|C|A|divide|mov-head|A|B`', which encodes cell replication (see Appendix B). Thus, the ancestral DISCO could not compute any logic function and was unicellular. The initial genome has length 150. During all simulations, genome length was restricted to $[145, 155]$, that is, offspring was nonviable if its genome was smaller than 145 or larger than 155. After the first 10 000 generations I determine the most recent common ancestor (MRCA) of the population. The genome of the MRCA is then used to seed the population for the next 10 000 generations. That is, the second part of a run is initialized with a genome that was produced during the first part of this run.

During the simulations, offspring was exposed to copy, insertion, and deletion mutations. For each offspring, the number of copy, insertion, and deletion mutations is chosen from a

Poisson distribution with mean 0.45, 0.025, and 0.025, respectively. An average mutation rate of 0.5 instructions per replication ($\approx 3.4 \times 10^{-3}$ mutations per instruction per replication) was chosen because it seemed to be optimal for the evolution of Y cell specific logic functions.

The merit of an organism is calculated based on the merit of its cells. The merit of a cell in turn is calculated based on the logic functions it can compute and utilize. A cell that cannot compute any logic function has merit 1. The second column of Table 1 shows how the merit of a cell changes if it is able to compute (and utilize) the corresponding logic function. The values were taken from Lenski et al. (1) and reflect the complexity of the respective function. In particular, the merit of a cell is multiplied by 2^n if the cell is able to compute a logic function of complexity n, where n gives the minimum number of NAND operations that are required to construct the logic function [see (1) and Section 1.2]. As mentioned before, not every cell is able to utilize every logic function. Which kind of cell can utilize which kind of functions during the first and the second part of the +X and the -X simulations is shown in Table 1.

Finally, the merit of the multicellular organism has to be calculated based on the merit of each cell. In short, the merit of an organism during the +X simulations is given by `SUM(X,D)*SET(Y)` and during the -X simulations by `SET(X,D)*SET(Y)`. The expression `SUM(X,D)` denotes the sum of merits of all D and X cells. Hence, during the +X simulations D and X cells contribute linearly to the merit of the organism. `SET(.)` is equal to the merit of a cell that can compute the set of functions that all the cells in the argument can compute. For example, `SET(X,D)` equals the merit of a cell that can compute the same set of functions that all X and D cells can compute. Since X and D cells can encode and utilize the same functions, `SET(X,D)` is essentially equal to the merit of one D cell. Hence, additional X cells cannot contribute to the merit of the organism. Similarly, `SET(Y)` equals the merit of a cell that can compute the set of functions that all Y cells can compute. Again, since one Y cell can compute the set of functions that all Y cells can compute, `SET(Y)` is equal to the merit of one Y cell.

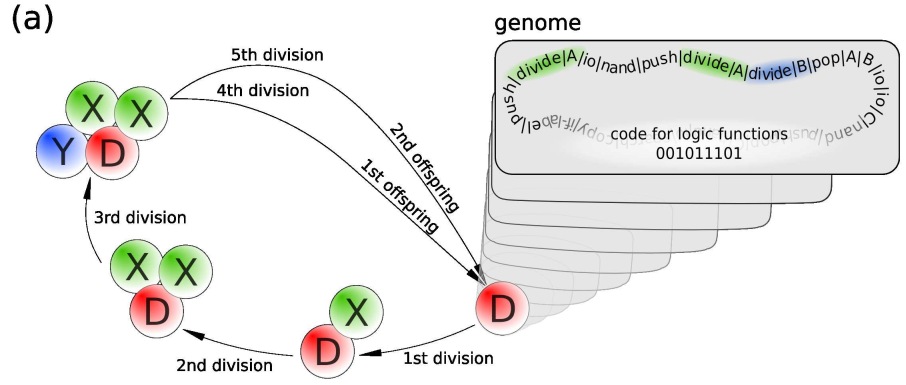
The use of `SUM(X,D)*SET(Y)` for the +X simulations was motivated by specialized cells in cyanobacteria. X and D cells are functionally equivalent and contribute additively to the merit of the organism: Two photosynthesizing vegetative cells can fix approximately twice as much carbon as one photosynthesizing cell. Y cells, however, are specialized cells that amplify the activity of X and D cells (by providing nitrogen, for example). This is reflected in the multiplicative contribution of Y cells to the merit of the organism. I use `SET(Y)` instead of `SUM(Y)` because DISCOs are thought to be small enough so that one Y cells can amplify the merit of X and D cells as well as two Y cells.

Figure 1 shows the first five cell divisions in the life of a multicellular DISCO composed of two X cells and one Y cell. It also shows what the merit of this organism would be during the first and the second part of the +X and -X simulations.

logic function	change in merit	cell types that can utilize the given function			
		first 10 000 generations		second 10 000 generations	
		+X	-X	+X	-X
NOT	$\times 2^1$	D,X	D	D,X	D
NAND	$\times 2^1$	D,X	D	D,X	D
AND	$\times 2^2$	D,X	D	D,X	D
OR_N	$\times 2^2$	D,X	D	D,X	D
OR	$\times 2^3$	D,X	D	D,X	D
AND_N	$\times 2^3$	D,X	D	D,X	D
XOR	$\times 2^4$	-	-	Y	Y
NOR	$\times 2^4$	-	-	Y	Y
EQU	$\times 2^5$	-	-	Y	Y

Table 1: Merit increase and cell type specifity during the +X and -X simulations. Please note that X cells can utilize functions during the -X simulations but are not able to contribute to the merit of the organism, since I am using `SET(D,X)*SET(Y)` instead of `SUM(D,X)*SET(Y)`.

(a)



(b)

		cell type specificity	utilized functions	merit of one cell	merit of the multicellular organism
1st 10000 generations	D	`111111000`	`001011000`	256	`+X: SUM(D,X)*SET(Y)= (256+256+256)*1=`**`768`**
	X	`111111000`	`001011000`	256	`-X: SET(D,X)*SET(Y)= 256*1=`**`256`**
	Y	`000000000`	`000000000`	1	
2nd 10000 generations	D	`111111000`	`001011000`	256	`+X: SUM(D,X)*SET(Y)= (256+256+256)*512=`**`393216`**
	X	`111111000`	`001011000`	256	`-X: SET(D,X)*SET(Y)= 256*512=`**`131072`**
	Y	`000000111`	`000000101`	512	

Figure 1: Multicellularity in DISCOs. **(a)** The first five cell division of a DISCO with a genome that encodes two X cells and one Y cell. As explained in Section 1.3, if the genome contains `divide` instructions that are followed by a modifier, then the D cell will retain daughter cells to build a multicellular organism. The genome of this DISCO contains two `divide|A` and one `divide|B` instructions. Consequently, the first three cell divisions are used to produce two X cells and one Y cell. Thereafter every further division results in cells that are released into the environment as offspring. **(b)** Calculating the merit of a multicellular organism. Let us assume that the genome of this DISCO encodes for AND, OR, AND_N, XOR, and EQU, that is, `001011101`. A logic function can increase a cells merit only if it is encoded in the genome and utilizable by the given cell type. Here, for example, the D cell can compute and utilize AND, OR, and AND_N (`001011000`) and has therefore a merit of $1 \times 2^2 \times 2^3 \times 2^3 = 2^8 = 256$. For the first 10 000 generations, Y cells cannot utilize any logic function and have therefore always merit 1. For the second part, however, Y cells can utilize the last three logic functions (`000000111`). Since the genome of this organism encodes XOR and EQU, Y cells receive a merit of $1 \times 2^4 \times 2^5 = 2^9 = 512$ for `000000101`. The -X simulations differ from the +X simulations in that X cells cannot contribute to the merit of the organism. I use `SET(D,X)` instead of `SUM(D,X)` and one D cell can compute the same set of functions as D and X cells together. We can clearly see that some somatic cells do not increase the merit of the DISCO (X cells during the -X simulations and Y cells during the first 10 000 generations). However, somatic cells constitute a cost, since they delay the time it takes to reach maturity and finally produce offspring. Hence, somatic cells that do not increase the merit of the organism are disadvantageous.

3 Developmental cost of one Y cell

I will calculate the fitness of an organism composed of $n+1$ cells relative to the fitness of an organism composed of n cells. Let us first consider $n = 1$. The unicellular organism produces offspring with every cell division at rate r_1, i.e., $\bigcirc \rightarrow \bigcirc + \bigcirc$. Its fitness is given by the rate of cell division. The bicellular organism produces offspring only after it reached maturity. In DISCOs (see Figure 1a) only one cell is able to divide. Hence, we have $\bigcirc \rightarrow \bigcirc\!\bigcirc \rightarrow \bigcirc\!\bigcirc + \bigcirc$. If x_1 and x_2 denote the frequency of the bicellular organisms in the unicellular and the bicellular stage of development, respectively, and r_2 the rate of cell division, then we can use the following differential equation to describe the population,

$$\begin{aligned} \dot{x}_1 &= -r_2 x_1 + r_2 x_2 - \Phi x_1 \\ \dot{x}_2 &= r_2 x_1 \qquad\qquad\quad - \Phi x_2 . \end{aligned} \tag{1}$$

The fitness of the bicellular organism is given by the average fitness Φ at equilibrium, which is given by the largest eigenvalue of

$$\begin{pmatrix} -r_2 & r_2 \\ r_2 & 0 \end{pmatrix} . \tag{2}$$

A short calculation shows, that the eigenvalues, λ, are given by the solutions of $\lambda(\lambda + r_2) - r_2^2$. Equivalently, we can solve $\lambda(\lambda + 1) - 1$ and multiply the solution with r_2. In any case, the fitness of the bicellular organism at equilibrium is given by $\Phi = (\sqrt{5} - 1)/2 \; r_2 \approx 0.62 \; r_2$. Hence the fitness of the bicellular organism relative to the fitness of the unicellular organism equals $r_2 0.62/r_1$. In DISCOs the rate of cell division is proportional to the merit of the organism and the efficiency of genome duplication. Since we are just interested in the effect of mutations that add an additional cell to an organism, we have $r_1 \approx r_2$. Hence, mutations that turn a unicellular into a bicellular organism decrease the relative fitness to 0.62.

The calculations for an organism of size n are similar. To calculate the fitness, Φ_n, of an organism of size n, we have to calculate the largest eigenvalue of the following $n \times n$ matrix,

$$\begin{pmatrix} -1 & 0 & \cdots & 0 & 1 \\ 1 & -1 & \ddots & & 0 \\ 0 & 1 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & -1 & 0 \\ 0 & \cdots & 0 & 1 & 0 \end{pmatrix}. \tag{3}$$

The eigenvalues λ are given by the roots of $\lambda(\lambda+1)^{n-1} - 1$. The fitness of an organism of size $n+1$ relative to and organism of size n is then given by Φ_{n+1}/Φ_n. For example, the cost of one additional, unused cell in an organism of size 10 is $\Phi_{11}/\Phi_{10} = 0.184/0.197 = 0.934$. The dotted line in Figure 2 shows Φ_n and the solid line Φ_{n+1}/Φ_n for a wide range of organism sizes.

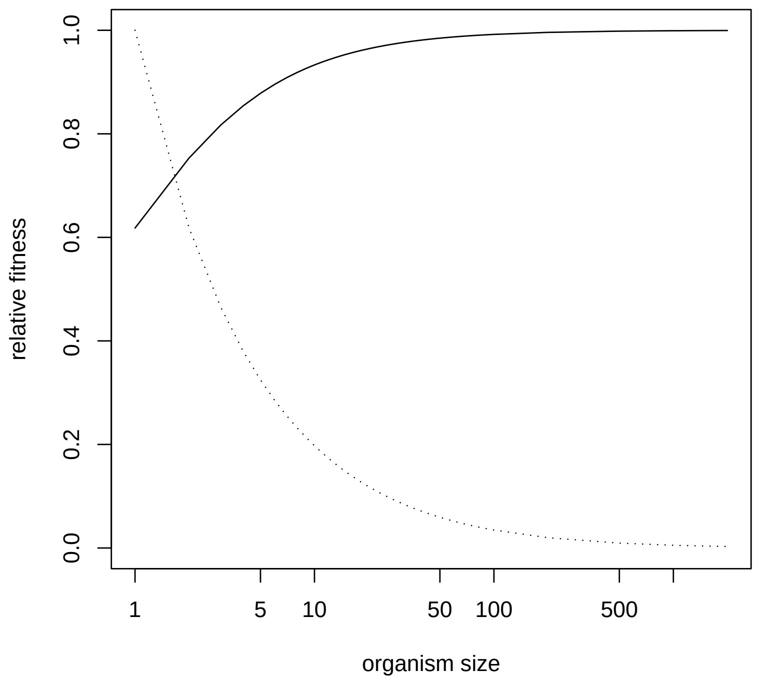


Figure 2: Developmental cost of an additional, unused cell. The dotted line shows the fitness of an organism of size n relative to the fitness of a unicellular organism. The solid line shows the fitness of an organism of size $n + 1$ relative to the fitness of an organism of size n. Hence, it shows how deleterious mutations are that add one, unused Y cell to an organism of size n.

References and Notes

1. Lenski R.E., Ofria C., Pennock R.T., & Adami C. (2003) *Nature* **423**, 139–44.

2. Ofria C. & Wilke C.O. (2004) *Artif Life* **10**, 191–229.

3. Goldfarb W. (2003) *Deductive Logic* (Hackett Publishing Company).

A List of Instructions

This section describes how instructions together with modifiers change the state of a DISCO cell. In the following `inst` will refer to any of the available instructions, `mod` will refer to a sequence of modifiers, and C(`mod`) to its complement. We have the three modifiers `A`, `B`, and `C` with their complements `B`, `C`, and `A`, respectively. The complement of a sequence of modifiers is given by the sequence of complements, for example, C(`CACABB`) $=$ `ABABCC`.

Instructions for the replication phase

The `copy` instruction: The `copy` instruction copies the instruction/modifiers to which the read head points to the daughter strand and moves the read head to the instruction/modifier following the just copied genome elements.

The `search` instruction: The `search` instruction repositions the flow head depending on modifier sequences in the genome. We have to distinguish the following cases.

- The `search` instruction is followed by another instruction, i.e., we have ... `search inst` ...: In this case the flow head is moved to point at `inst`.
- The `search` instruction is followed by a sequence of modifiers, i.e., we have ... `search mod` ...: In this case we look for the compliment of the modifier sequence C(`mod`) in the genome. If this sequence can be found, then the flow head is moved to point at the instruction following C(`mod`). If this sequence cannot be found, then the flow head is moved to point to the instruction following `mod`.

The `mov-head` instruction: The `mov-head` moves either the instruction head or the read head to the position of the flow head. The instruction head is moved to the flow head if the DISCO cell executes a `mov-head` that is followed by modifier `A`, i.e., `mov-head A`. If the `mov-head` is followed by modifier `B`, i.e., `mov-head B`, then the read head is moved to the flow head. In all other cases, the `mov-head` instruction does not affect the state of the automaton.

The `if-label` instruction: The `if-label` instruction can be used to skip instructions depending on the most recently copied genome element. Let us consider the sequence `if-label mod inst`. The `inst` instruction will be executed only if the most recently copied genome element is the complement of `mod`. If this is not the case, then `inst` will be ignored and the instruction head moved forward to the next instruction. In case of `if-label inst`, the automaton will execute `inst` if and only if the most recently copied genome element is an instruction.

The `divide` instruction: The `divide` instruction initiates cell division and terminates the replication phase. The `divide` instruction will successfully generate a new DISCO cell if the genome has the right length. For this work the genome cannot be smaller than 145 or larger than 155 modifiers and instructions long. The generated daughter cell is either used as a somatic cell or released into the environment as offspring.

Instructions for the metabolic phase

The state changes for the metabolic phase are more straight forward than the ones for the replication phase. They only affect the merit of the organism and data stored in the registers and the stack. In the following I will use s_i, a, b, and c to denote logic compounds and x to denote a newly generated logic variable. The first column of following tables shows the state of the automaton before the sequence in the second column is executed. The third column shows the state of the automaton after the execution of the sequence. Not shown is the advance of the instruction head to the next instruction.

The `blank` instruction: The `blank` instruction does not change the state of the automaton. It is essentially just used as a place holder in ancestral genomes. Usually, the `blank` instruction is not part of the pool of instructions from which instructions for the copy and insertion mutations are chosen. Consequently, `blank` instructions will eventually disappear from the genome.

The `io` instruction: The merit of the DISCO before the execution of the `io` instruction is given by m. The merit of the DISCO, after testing a, b, and c for logic functions is given by m_a, m_b, and m_c, respectively. A DISCO cell is rewarded only once for a given logic function during one metabolic phase.

before	instruction	after
merit:m A:a B:b C:c	`io A`	merit:m_a A:x B:b C:c
	`io B`	merit:m_b A:a B:x C:c
	`io C`	merit:m_c A:a B:b C:x
	`io inst`	merit:m_b A:a B:x C:c

The `nand` instruction: Please note that the `nand` instruction is the only instruction that can generate new logic compounds.

before	instruction	after
A:a B:b C:c	`nand A`	A:$a\|b$ B:b C:c
	`nand B`	A:a B:$b\|c$ C:c
	`nand C`	A:a B:b C:$c\|a$
	`nand inst`	A:a B:$b\|c$ C:c

The `swap` instruction:

before	instruction	after
A:a B:b C:c	`swap A`	A:b B:a C:c
	`swap B`	A:a B:c C:b
	`swap C`	A:c B:b C:a
	`swap inst`	A:a B:c C:b

The `push` instruction:

before	instruction	after
A:a B:b C:c stack: $s_1, s_2, \ldots$	`push A`	stack: $a, s_1, s_2, s_3, \ldots$
	`push B`	stack: $b, s_1, s_2, s_3, \ldots$
	`push C`	stack: $c, s_1, s_2, s_3, \ldots$
	`push inst`	stack: $b, s_1, s_2, s_3, \ldots$

The `pop` instruction:

before	instruction	after
	`pop A`	A:s_1 B:b C:c stack: $s_2, s_3, \ldots$
A:a B:b C:c	`pop B`	A:a B:s_1 C:c stack: $s_2, s_3, \ldots$
stack: $s_1, s_2, \ldots$	`pop C`	A:a B:b C:s_1 stack: $s_2, s_3, \ldots$
	`pop inst`	A:a B:s_1 C:c stack: $s_2, s_3, \ldots$

B Example Genomes

Encoding Replication

The following shows a sequence of the state changes that lead to genome replication. During the replication phase, state changes affect only the position of the read, flow, and instruction head. These positions are indicated by underlines, overlines, and bold font, respectively, i.e., $\underline{\text{read head}}$, $\overline{\text{flow head}}$, and **instruction head**.

- $\overline{\underline{\texttt{io}}}|\textbf{\texttt{search}}|\texttt{copy}|\texttt{if-label}|\texttt{C}|\texttt{A}|\texttt{divide}|\texttt{mov-head}|\texttt{A}|\texttt{B}$
  daughter strand:
  `search` moves the flow head to the `copy` instruction
- $\underline{\texttt{io}}|\texttt{search}|\overline{\textbf{\texttt{copy}}}|\texttt{if-label}|\texttt{C}|\texttt{A}|\texttt{divide}|\texttt{mov-head}|\texttt{A}|\texttt{B}$
  daughter strand:
  `copy` copies the `io` instruction to the daughter strand and moves the read head forward
- $\texttt{io}|\underline{\texttt{search}}|\overline{\texttt{copy}}|\textbf{\texttt{if-label}}|\texttt{C}|\texttt{A}|\texttt{divide}|\texttt{mov-head}|\texttt{A}|\texttt{B}$
  daughter strand: `io`
  The complement of `C|A` is given by `A|B`. Since the most recently copied element is `io` and not `A|B`, `if-label` ignores `divide` and advances the instruction head to the `mov-head` instruction.
- $\texttt{io}|\underline{\texttt{search}}|\overline{\texttt{copy}}|\texttt{if-label}|\texttt{C}|\texttt{A}|\texttt{divide}|\textbf{\texttt{mov-head}}|\texttt{A}|\texttt{B}$
  daughter strand: `io`
  `mov-head` moves the instruction head to the position of the flow head.
- $\texttt{io}|\underline{\texttt{search}}|\overline{\textbf{\texttt{copy}}}|\texttt{if-label}|\texttt{C}|\texttt{A}|\texttt{divide}|\texttt{mov-head}|\texttt{A}|\texttt{B}$
  daughter strand: `io`
  `copy` copies the `search` instruction to the daughter strand and moves the read head forward.
- $\texttt{io}|\texttt{search}|\overline{\underline{\texttt{copy}}}|\textbf{\texttt{if-label}}|\texttt{C}|\texttt{A}|\texttt{divide}|\texttt{mov-head}|\texttt{A}|\texttt{B}$
  daughter strand: `io|search`

  This loop continues until the automaton copies `A|B`, the last genome element.

- $\texttt{io}|\texttt{search}|\overline{\textbf{\texttt{copy}}}|\texttt{if-label}|\texttt{C}|\texttt{A}|\texttt{divide}|\texttt{mov-head}|\underline{\texttt{A}|\texttt{B}}$
  daughter strand: `io|search|copy|if-label|C|A|divide|mov-head`
  Copies `A|B` to the daughter strand and moves the read head forward.
- $\underline{\texttt{io}}|\texttt{search}|\overline{\texttt{copy}}|\textbf{\texttt{if-label}}|\texttt{C}|\texttt{A}|\texttt{divide}|\texttt{mov-head}|\texttt{A}|\texttt{B}$
  daughter strand: `io|search|copy|if-label|C|A|divide|mov-head|A|B`
  Now the most recently copied genome element is identical to the complement of `C|A` and `if-label` does not skip the `divide` instruction
- $\underline{\texttt{io}}|\texttt{search}|\overline{\texttt{copy}}|\texttt{if-label}|\texttt{C}|\texttt{A}|\textbf{\texttt{divide}}|\texttt{mov-head}|\texttt{A}|\texttt{B}$
  daughter strand: `io|search|copy|if-label|C|A|divide|mov-head|A|B`
  The `divide` instruction ends the replication phase and the daughter strand is used to build a new cell.

Encoding NAND and AND

This sequence creates two logic compounds. The first one $(a|b)|(a|b)$ is equivalent to AND and the second one $a|b$ is equivalent to NAND, where a and b are logic variables.

`**io**|io|C|nand|push|pop|C|nand|io|io|C`	A:	B:	C:	stack:	merit= 1
io|**io|C**|nand|push|pop|C|nand|io|io|C	A:	B:a	C:	stack:	merit= 1
io|io|C|**nand**|push|pop|C|nand|io|io|C	A:	B:a	C:b	stack:	merit= 1
io|io|C|nand|**push**|pop|C|nand|io|io|C	A:	B:$a\|b$	C:b	stack:	merit= 1
io|io|C|nand|push|**pop|C**|nand|io|io|C	A:	B:$a\|b$	C:b	stack:$a\|b$	merit= 1
io|io|C|nand|push|pop|C|**nand**|io|io|C	A:	B:$a\|b$	C:$a\|b$	stack:	merit= 1
io|io|C|nand|push|pop|C|nand|**io**|io|C	A:	B:$(a\|b)\|(a\|b)$	C:$a\|b$	stack:	merit= 1
io|io|C|nand|push|pop|C|nand|io|**io|C**	A:	B:c	C:$a\|b$	stack:	merit= 1×2^2
io|io|C|nand|push|pop|C|nand|io|io|C	A:	B:c	C:d	stack:	merit= $1 \times 2^2 \times 2^1$

Encoding NAND, AND, and Replication

This sequence is pieced together using the two sequences above. The first part encodes the two logic function NAND and AND, and the second part genome replication.

```
io|io|C|nand|push|pop|C|nand|io|io|C|search|copy|if-label|C|A|divide|
mov-head|A|B
```

Encoding NAND, AND, Replication, and One Y Cell

This sequence is a derivative of the previous sequence. It contains a `divide|B` and encodes therefore for a multicellular organism with one Y cell. Please note that the `if-label|C` before the `divide` is required to prevent a premature initiation of cell division.

io|io|C|nand|push|if-label|C|**divide|B**|pop|C|nand|io|io|C|search|copy|
if-label|C|A|divide|mov-head|A|B