Clever Creatures: Case Studies of Evolved Digital Organisms

Laura M. Grabowski[1], David M. Bryson[2], Fred C. Dyer[2], Robert T. Pennock[2] and Charles Ofria[2]

[1]University of Texas-Pan American, Edinburg, TX 78539
[2]BEACON Center for the Study of Evolution in Action
Michigan State University, East Lansing, MI 48823
grabowskilm@utpa.edu

Abstract

We present a "bestiary" of three digital organisms (self-replicating computer programs) that evolved in three different experimental environments in the Avida platform. The ancestral environments required the evolving organisms to use memory in different ways as they gathered information from the environment and made behavioral decisions. Each organism exhibited a behavior or algorithm of particular interest: 1) simple step-counting odometer; 2) clever low-level computation; and 3) pronounced modularity in both program structure and program functionality. We present descriptive in-depth analysis of the case study organisms, with a focus on the structure and operation of the evolved algorithms that produce the individuals' fitness enhancing behaviors.

Introduction

The multi-disciplinary nature of Artificial Life (Alife) makes for rich cross-fertilization between computer science and biology, with research that focuses on organizing principles of living systems (Bedau, 2007). In the broad context of evolving building blocks of simple intelligent behavior, we present experiments that address the interaction of memory, environment, and learning in an evolutionary context. Here, we recount the key points of work reported in Grabowski et al. (2010) and expand on that discussion. In the previous paper, we presented our experimental motivation and design, and gave a high-level view of the evolved behavior. In the current paper, we dissect the algorithms that lay beneath the evolved behaviors, exposing the low-level mechanisms that produce the fitness enhancing behaviors. We produced our analyses through instruction-by-instruction examination of execution traces of the evolved digital organisms.

An important aim of our approach is to inform inquiry in both computer science and biology. With that aim in mind, we selected three highly successful digital organisms that evolved in three different experimental environments. Each of the case study organisms has a salient feature or behavior that seems critical for the evolved solution to work. The first organism evolved a simple odometer that it uses to count its steps, turning immediately before it would have otherwise entered an environmental hazard. The second organism evolved a computational strategy that uses low-level bit operations to ensure correct behavioral responses to cues from the environment. This computational tactic is of special interest because it produces high-level behavior from low-level operations, and it also exemplifies a foundational principle of biology and psychology, that evolution tends to produce parsimonious solutions to behavioral problems ("Morgan's Canon," Morgan (1894)). The third organism evolved distinct functional and structural modularity; the role of modularity is a topic of great interest in a number of contexts. These digital organisms were products of an open-ended evolution system, and the system did not explicitly select for any of the solutions. We are exploring the range of unexpected solutions that can come out of such a system. The diversity of the evolved solutions is broad, even though we are dissecting only a handful of examples.

Methods

Avida: Overview

Digital evolution (Adami et al., 2000) is a type of evolutionary computation that places a population of self-replicating computer programs (digital organisms) in a computational environment, where the population evolves as the organisms replicate, mutate and compete for environmental resources. Digital evolution is a useful tool for understanding evolutionary processes in biology and for leveraging evolution to find solutions to computing and engineering problems. Avida (Lenski et al., 2003; Ofria et al., 2009) is a widely used software platform for digital evolution. Avida is an instance of evolution in its own right (Pennock, 2007), and provides a host of tools for experimental studies. In this section, we provide a brief summary of how Avida functions. For more detailed information, see Ofria et al. (2009).

The Avida world is a discrete two-dimensional grid of cells that holds the population of digital organisms. At most one organism (Avidian) may occupy a grid cell. The genome of an Avidian is a circular list of program instructions that resemble assembly language, that runs in a virtual central processing unit (CPU). The organism's CPU contains three

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2011/23/44/2022463/0262297140chap44.pdf by guest on 23 June 2024

registers (AX, BX, and CX), two stacks, and several heads (FLOW, used as a target for jumps; IP, an instruction pointer that denotes the next command to be executed; READ, for reading an instruction; WRITE, for writing an instruction). Execution of the instructions in the organism's genome act on the elements of the virtual CPU, incurring a cost measured in virtual machine cycles. An Avidian accomplishes all functions by executing the instructions in its genome, such as movement, gathering information from its environment, or replicating. The basic Avida instruction set is Turing-complete (Ofria et al., 2002), and is easily extended by adding new instructions to the system.

An Avidian replicates by copying its genome into a new block of memory. Mutations in Avida occur through errors in this copying process that produce differences between the genomes of parent and offspring. These differences may take the form of inserting or deleting an instruction, or changing one instruction to another, and occur at random with a user-defined probability. The Avida instruction set has the property of remaining syntactically correct in the presence of mutations, so a mutated genome will continue to execute, even if it performs no useful functions (Ofria et al., 2002).

Newly-produced offspring are placed in a randomly selected grid cell, overwriting any organism that was occupying the cell. This process gives a fitness enhancing advantage to an organism that can replicate faster than others in the population; organisms compete for the limited resource of grid space, and individuals that replicate sooner than others will have a higher proportion of descendants in future populations. Avidians may replicate sooner if they speed up their execution by accumulating metabolic rate bonuses as they evolve to perform user-specified tasks. Fitness in Avida is measured as the organism's metabolic rate divided by the number of cycles the organism requires to replicate.

Experiment Design

We placed each Avidian in an environment containing a path that it could follow to collect food and increase its metabolic rate. Our environments were inspired by maze-learning experiments with honey bees (Zhang et al., 2000). Organisms had to sense the cues that formed the path and react appropriately to them. In some cases, advantageous behavior involved the ability to store experience for later decision-making.

For these experiments, we added sensing and movement instructions to the basic Avida instruction set. The *sg-move* instruction allows an organism to move one cell in the direction of its current orientation (its facing). In this study, each digital organism had its own virtual grid, so organisms did not interact during movement. Two instructions accomplished orientation changes, *sg-rotate-right* for turning 45° to the right and *sg-rotate-left*, for turning 45° to the left.

We added a sensing instruction, *sg-sense*, that allowed the Avidian to get sensory information from its environment. When an Avidian executes the sensing instruction, the instruction places a predefined value in the executing Avidian's BX register, according to which cue is present in the grid cell at the organism's current location. These values are analogous to sensory input that the organism obtains from the environment, and are not directly used in calculations. The operation of this *sg-sense* instruction is important to the analyses of the evolved programs. The virtual grids for these experiments had a sensory cue in each cell of the grid. The environments contained some combination of the following cues (Grabowski et al., 2010):

1. **Nutrient:** A cue that indicates a cell is on the path, and provides "food" (*i.e.*, energy that adds to the organism's metabolic bonus). The nutrient cue has a sense value of 0 from the *sg-sense* instruction.

2. **Directional cue**: A cue indicating that a 45° turn to either the right or left is needed to remain on the path; the cell also contains nutrient. Right turns and left turns have different sense values from *sg-sense*, 2 for right and 4 for left.

3. **General turn cue:** A cue that indicates a turn but does not specify the direction, and contains nutrient. The return value for the general turn cue is 1.

4. **Empty:** A cue that indicates a cell that is not on the path. Movement into empty cells depletes energy gained by movement into cells that are on the path. The *sg-sense* instruction returns a sense value of -1 for empty cells.

We added two new comparison instructions to the Avida instruction set, *if-greater-than-X* (*if-grt-X*) and *if-equal-to-X* (*if-equ-X*), that supplemented existing comparison instructions. These instructions allow an organism to compare the value in its BX register to a predefined value. A *no-op* (NOP) label immediately following the comparison instruction determines the value to use in the comparison. We added the new comparison instructions because an Avida organism has to combine several different arithmetic instructions in order to compare a register value to any specific value. The new *if-equ-X* and *if-grt-X* instructions provided a shortcut and simplified comparisons for the Avidians, and also contributed to evolved genomes that were simpler to analyze. The details of these new instructions did not adversely affect the adequacy of our model, since our focus in the experiments was on memory; the mechanisms of constructing comparisons are not relevant to our questions of interest.

We constructed several environment types using the cues described above. The three organisms that we present in this paper evolved in three different environments.

- **Environment 1 and Environment 2: Evolving reflexes.** The first two environments contain paths with directional

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2011/23/44/2022463/0262297140chap44.pdf by guest on 23 June 2024

(right and/or left), nutrient, and empty cues. Paths in Environment 1 contained only one type of directional cue (right or left) in each path instance; they are "single-direction turn" environments (see Figure 1). Environment 2 paths contained both right and left turns in the same path, and so are "dual-turn" environments (see Figure 2a). With both of these environment types, we expected the organisms to evolve a reflexive reaction to the path cues that they sensed, since the sensory information did not have to be retained for decision-making in the individual's future.

- **Environment 3: Evolving volatile memory.** This environment type uses all four sensory cue types. The specific directional cue (right or left) is encountered when the turn is the first turn on the path or when the turn direction changes (*e.g.*, the organism has done one or more right turns and now needs to turn left). The general turn cue is encountered when the turn direction is to remain the same as the previous turn (*e.g.*, the organism executed a left turn at the previous turning and the current turn is also to the left) (see Figure 2b). This arrangement requires the organisms to evolve mechanisms for storing, using, and updating information about their experience on the path they are traversing, equating to a simple form of memory.

Organisms were presented with one of several different paths of the particular environment type (four different paths for Environments 1 and 3, and five different paths for Environment 2), chosen at random when the organism was born. Each individual experienced only one specific path in its lifetime, but all of the environments were experienced by multiple organisms during the course of evolution. In all experiments, organisms could raise their metabolic rate bonus through a path traversal task. The details of the task are presented in Grabowski et al. (2010). We ran 50 experimental replicates for each environment type, seeding each experiment with an organism with only the ability to replicate. All other functions had to evolve, using instructions entering the organism's genome through mutations. We used the default Avida mutation rates for all our experiments, a 0.085 genomic mutation rate for a length-100 organism (a 0.0075 copy-mutation probability per copied instruction, and insertion and deletion mutation probabilities of 0.05 per divide) (Ofria et al., 2009). Experiments ran for a median of approximately 33,000 generations (250,000 Avida updates). Our populations had a maximum of 3600 individuals.

Results and Discussion

Environment 1: Evolved Odometry

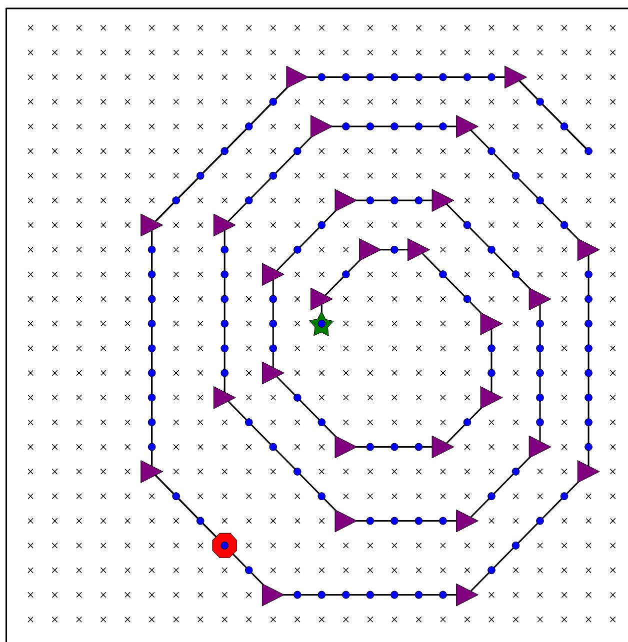
For Environment 1 (single-direction turn paths), we deliberately constructed simple paths with two regularities: each individual environment contained only right turns or only left turns, and the path progressed continuously outward from the starting position, giving the paths a spiral shape. There was also one unintentional regularity: the ancestral right-turn paths were the same except for the organism's starting position and the resulting distance to the first turn. The left-turn paths had more differences in the numbers of steps between turns. One population from this environment evolved a step-counting organism. This result is particularly exciting, since some animals use a mechanism analogous to step counting to determine the distance they have traveled on excursions away from their nests (Wittlinger et al., 2006). While odometry is considered a straightforward problem in robotics, it is by no means clear how it works in most animals, how it participates in higher-level processes such as path integration, and how it first evolved. Our approach may afford a way of exploring these problems.

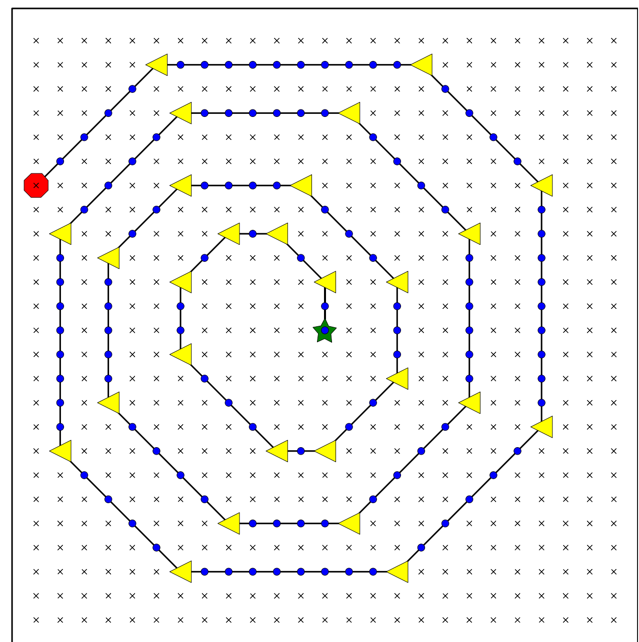
Figure 1 shows trajectories of the Environment 1 example organism (Org:StepCount) moving on a right-turn-only path (Figure 1a) and on a left-turn-only path (Figure 1b). Org:StepCount's evolved strategy performed well in both turn environments. Interestingly, Org:StepCount backtracked on the right-turn grid, *i.e.*, it turned around and retraced its steps on the path. This behavior did not reduce Org:StepCount's metabolic rate; the task quality calculation rewarded movement into unique path cells but did not penalize an organism for multiple movements into a path cell (Grabowski et al., 2010). Org:StepCount was able to navigate the entire right-turn path without entering any empty cells and also successfully followed the left-turn-only path, stopping after it encountered a single empty cell.

We analyzed an execution trace of Org:StepCount while it traversed each of these two paths, to uncover how its algorithm produces the observed behavior. Most—but not all—of the movement and replication code of Org:StepCount's program is organized into two sections. Some instructions for this behavior (*i.e.*, movement and replication) are scattered in other locations in the genome, so Org:StepCount is not completely modular. A distinctly modular organism evolved in Environment 3, discussed later. One of the code sections ("Section 1A") handles moving on a right-turn path, and the second ("Section 1B") focuses on left-turn paths. Section 1B also contains a nested copy loop that is used for replication. Both of these code sections execute, whether the organism is on a right-turn or left-turn path, but the resulting behavior differs according to the path type (*i.e.*, right or left). Section 1A is essentially a counting routine. When Org:StepCount is traversing a right-turn path, Section 1A counts its steps; for left-turn paths, Section 1A counts the number of 45° turns the organism executes. When on a left-turn path, Org:StepCount uses Section 1B to travel to the end of the path and then replicate. When Org:StepCount is on a right-turn path, Section 1B allows the organism to avoid stepping off the end of the path by retracing some of its steps, at the same time finishing its replication process.

The following is a pseudocode description of the functionality of Section 1A:

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2011/23/44/2022463/0262297140chap44.pdf by guest on 23 June 2024





(a) Right Turn Path

(b) Left Turn Path

Figure 1: Trajectories of Org:StepCount on ancestral paths.

```
DO
  rotate right
  IF (CX > 0) copy
  copy
  CX <- sense
  IF (CX equal nutrient) rotate left
  ELSE IF (CX equal right turn)
    CX <- 128
    move
  BX <- BX + 1
WHILE (BX not equal CX)
```

Org:StepCount's current environment (*i.e.*, left- or right-turn) determines how this code executes. When traversing a right-turn path, Org:StepCount uses this loop to count its steps to the end of the path. Setting the CX register to the value of 128 (by reading the current position of the Instruction Pointer (IP)) and incrementing the value in the BX register (which begins at a value of 0 at the first loop iteration) with every loop iteration sets up the exit condition for the loop: after Org:StepCount has taken 127 steps in the loop, the last increment of the BX register causes execution to exit the loop. When executing this loop on a left-turn path, the organism remains in the same spot and executes the loop four times, performing a one-eighth turn in each iteration. When the value of the BX counter reaches 4, Org:StepCount exits the loop, and is now facing in the "wrong" direction (*i.e.*, facing back the way it has already come). The section of code immediately following this section includes another set of four one-eighth turns, so Org:StepCount regains the facing it had upon entering Section 1A.

Section 1B operates as follows:

```
DO
  move
  BX <- sense
  IF (BX not equal nutrient)
      rotate left
  IF (BX equal empty)
    WHILE ( not end label) copy
  ELSE IF (not end label) copy
  IF (end label) divide
WHILE (BX not equal empty) AND
      (not end label)
```

When this algorithm is executed on a left-turn-only path, Org:StepCount moves along the path, eventually moving one step off the end of the path into an empty cell. At that point, Org:StepCount "stands still," and executes a tight copy loop to complete copying its genome to its offspring, at which time it divides. On a right-turn path, however, Org:StepCount never enters the tight copy loop; instead, it copies just one instruction for each iteration of Section 1B, while it retraces its steps along the path. This strategy produces the backtracking in the trajectory plot of Figure 1a. The organism retraces the path moving back toward its initial location, stopping part of the way through the path (red

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2011/23/44/2022463/0262297140chap44.pdf by guest on 23 June 2024

octagonal symbol). The number of instructions needed to produce an offspring remain similar on right- and left- turn paths (1779 instructions for the right-turn path shown in Figure 1a and 1780 instructions for the left-turn path shown in Figure 1b) since an extra instruction is copied with every iteration of Section 1A when Org:StepCount is moving on a right-turn path. Table 1 lists the Avida instructions for the two code sections described above.

Section 1A	Section 1B
sg-rotate-r	sg-move
if-grt-0	sg-sense
nop-C	nop-B
h-copy	if-n-equ
h-copy	sg-rotate-l
sg-sense	if-equ-X
nop-C	pop
jmp-head	if-less
sg-rotate-l	h-search
if-equ-X	if-label
get-head	nop-C
sg-move	h-divide
inc	h-copy
if-n-equ	mov-head
mov-head	

Table 1: Avida instructions for Org:StepCount.

Environment 2: Economical Code and Clever Math

Environment 2, the dual-turn environment, presents evolution with a slightly more complex version of the problem encountered in Environment 1, since evolution must always contend with both turn directions in every path. The evolved algorithm of the example organism from this environment (Org:BitOperator) is interesting because it evolved some remarkably clever math that helped it succeed in its environment, using simple, low-level computations to produce complex, high-level behavior.

Org:BitOperator successfully negotiated both ancestral paths and novel paths. Figure 2a shows Org:BitOperator's trajectory on a novel path. The dimensions of the grid containing the path are different from the dimensions of the grids in the ancestral environments: the novel path shown has dimensions of 20×20, as opposed to the 25×25 grids that were experienced during evolution. Since all environment grids are toroidal, the grid dimensions should make no difference to organisms, and organisms never have access to any global information. However, we included tests like these to provide additional evidence that the evolved algorithms do not work by finding and exploiting geometrical information, such as grid size, but instead function through gathering and using information from the environment.

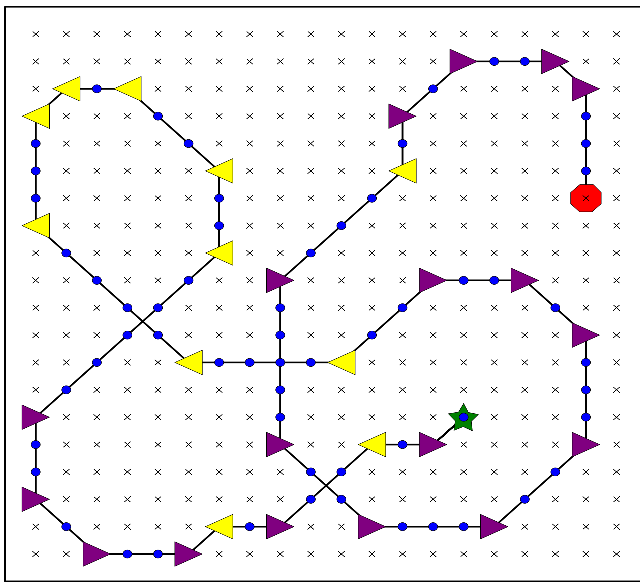
Org:BitOperator executes most of its movement with a concentrated movement loop. At a high level, the structure of the code is *move-sense-decide*. The decision concerns whether or not to turn, and if a turn is to be made, which direction to turn. Within the loop, conditional statements guard the turn directions to provide the correct execution flow for each environmental cue. In pseudocode, this movement loop functions as follows:

```
DO
  IF (BX > 1) rotate left
  copy
  move
  BX <- sense
  BX <- right-shift(BX)        #Line 1
  IF (BX equal 1) rotate right #Line 2
  ELSE IF (BX < CX)            #Line 3
    IF (BX > 0) CONTINUE       #Line 4
WHILE (BX > 0)
```

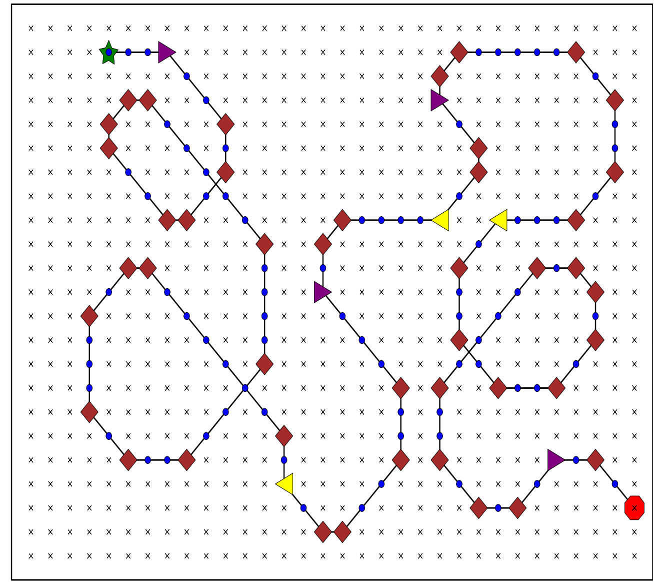
Org:BitOperator has a simple, but clever, mechanism for using the default behavior of the comparison instructions to select the correct action, based on the current sense information. Org:BitOperator manipulates the current sensed cue value so that the values match the comparisons as needed. The key detail of this loop's execution is how the right-shift operation (Line 1) prepares the sensed cue value for use with the unmodified comparison statements. Stepping through the algorithm, starting from the `BX<-sense` line, the current cell cue is sensed, and the value placed in BX. That value is then right-shifted, dividing most sense values by 2. Recall the return values from the *sg-sense* instruction. If the sensed cue is nutrient (return value = 0), BX is still 0; if the cue is right-turn (return value = 2), BX is now 1; if the cue is left-turn (return value = 4), BX is now 2; if the cue is empty (return = -1), BX is still -1 (since the operation is an arithmetic right-shift, the sign bit is preserved in the shift). This low-level manipulation of the sense value permits the algorithm to use the default behavior of the comparison instructions, thus avoiding the need for NOP modification of the instructions. This characteristic provides more robust performance for Org:BitOperator, since the comparison needs only one instruction to complete its action, not two. The first comparison (Line 2) is true when the last sensed cue is right-turn, so the right turn is executed. The next comparison (Line 3) is false for all cues except empty, so execution returns to the top of the loop as long as the organism encounters non-empty cells. Sensing an empty cell triggers loop exit.

This solution is simple and economical, accomplishing the job with few extraneous instructions. Org:BitOperator has evolved an equally frugal copy loop near the end of its genome. The copy loop performs the bulk of Org:BitOperator's replication, and begins execution only after the movement loop has terminated. Table 2 gives the Avida code for Org:BitOperator's movement loop. Not only is the elegance of these evolved solutions to be admired from

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2011/23/44/2022463/0262297140chap44.pdf by guest on 23 June 2024



(a) Sample trajectory, dual-turn novel path.



(b) Sample trajectory, irregular turn novel path.

Figure 2: Trajectory of the two example evolved organisms from Environments 2 and 3. Figure 2a shows the example organism, Org:BitOperator, from Environment 2 (dual-turn paths) traveling on a novel path. The grid containing the path has different dimensions (20×20) from those of the ancestral paths (25×25). Figure 2b shows the example organism, Org:Modular, from Environment 3 (irregular paths) traversing a novel path. This path grid also has dimensions (23×32) that differ from those of the ancestral environments (25×25).

Movement Loop
if-grt-X
sg-rotate-l
h-copy
sg-move
sg-sense
shift-r
if-equ-X
sg-rotate-r
if-less
if-grt-0
mov-head

Table 2: Avida instructions fused in our example dual-turn environment organism, Org:BitOperator.

a computational perspective, they also provide evidence of "Morgan's Canon," a parsimony principle that has guided a century of research in animal and human psychology (Morgan, 1894). By this principle, one should prefer hypotheses that invoke simpler rather than more complex mechanisms of information processing. Our results suggest that digital evolution could lead to empirical study of this principle.

Environment 3: Evolving Modularity

Environment 3 was the most complex environment in our study. To enhance fitness in this environment, organisms needed to make decisions based on their life experience, and update their memory of that experience at irregular intervals. The case study organism from this environment (Org:Modular, shown in Figure 2b traversing a novel path) evolved an algorithm with functional and structural modularity that provides appropriate behavioral responses to environmental conditions.

The execution of Org:Modular's genome is fairly complex, with a high degree of flexibility to handle conditions in its environment. In general, Org:Modular moves its execution to different parts of its genome depending on the sensed cue from the environment. Org:Modular has two loops for its path-following, one loop that navigates left-turn path segments, "Module 3A," and the other loop for traveling on right-turn path sections, "Module 3B." Org:Modular has well-defined functional and structural modularity in its genome for handling right-turn and left-turn path sections. Such refined modularity was not observed in other organisms that we analyzed. Module 3A appears first in Org:Modular's genome, before Module 3B. Module 3A can perform an arbitrary number of forward steps and consecutive left turns. This behavior in Module 3A is produced by a nested loop that results in straight-ahead movement on

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2011/23/44/2022463/0262297140chap44.pdf by guest on 23 June 2024

the path; iterations of this smaller loop continue until a non-zero cue is sensed. The smaller loop terminates when a left-turn or general turn is sensed, but execution remains within Module 3A. Sensing a right-turn or empty cue will exit both the smaller loop and Module 3A. Module 3B enables Org:Modular to negotiate right-turn path sections, accommodating any number of repeated right turns and forward steps. Execution exits Module 3B upon sensing a left turn cue, and jumps to the beginning of the organism's genome, thereby arriving again at Module 3A. Execution of Module 3B terminates if an empty cell is sensed, continuing with the instructions following the module. Org:Modular also has a modular copy loop near the end of its genome that manages the majority of the copying for the organism's replication.

Module 3A, for navigating on left-turn path sections, functions as follows:

```
DO
  DO
    move
    BX <- sense
    IF (BX < CX)
      swap (BX, CX)
    BX <- BX - 1     # Line 1
  WHILE (BX < CX)    # Line 2
    rotate left
WHILE (BX equal turn) OR (BX < CX)
```

The decrement of the value in BX following the sense instruction (Line 1) manipulates the value in the BX register such that execution remains in the nested loop as long as the organism is sensing nutrient cues (meaning that Org:Modular is moving straight on the path), but will exit the nested loop when any other cue is sensed. Whenever this module is executing, the value in CX is 0 at the top of the loop. Executing `BX <- BX-1` with the nutrient return value (0) places a value of -1 in BX, so execution does not exit the nested loop (Line 2). Decrementing the general turn cue return value (1) places a value of 0 in BX, causing execution to exit the nested loop and do the left turn. When the right-turn return value (2) is decremented, the value in BX becomes 1, and the nested loop is exited. Execution then exits Module 3A, after executing the left turn. The swap of values in BX and CX is executed only if an empty cell is sensed. The swap puts 0 in BX, and -1 in CX, so the nested loop is exited, and execution leaves Module 3A after the left turn, since BX is equal to CX after BX is decremented.

A pseudocode description of the functionality of Module 3B, for moving through right-turn path segments, is:

```
DO
  rotate right
  IF (BX < CX) BX <- sense
  move
  BX <- sense
  IF (BX equal turn) CONTINUE # Line 1
    ELSE IF (BX equal left turn)
      jump IP to 0
    BX <- BX + 1
    rotate left
 WHILE (BX not equal CX)
```

There is a section of instructions between the modules that has no move instructions, but has a single right-turn instruction that negates the last left turn performed before exiting Module 3A. An additional right-turn instruction executes before Module 3B entry, ensuring proper orientation for turning right, since Module 3B contains both right- and left-turn instructions that always execute. Correct orientation is maintained by selectively executing the left turn at the end of the module. When a general turn cue is sensed, execution in Module 3B skips the left turn (since BX=1), and returns directly to the top of the loop. When a nutrient is sensed, BX=0, so the increment of BX and the left turn are executed. When Org:Modular senses a left-turn cue, execution jumps out of Module 3B, returning to the beginning of the genome. As in Module 3A, the value in CX is 0 during execution of Module 3B. If an empty cell is sensed, incrementing the value places a value of 0 in BX, and execution exits the module. Once Org:Modular moves into an empty cell, execution moves to the copy loop, and Org:Modular completes its replication. Table 3 lists the Avida code for Org:Modular's path-following modules.

Two features of Org:Modular are particularly interesting. The first is the organization of the genome. The sections of the genome that do the bulk of the relevant behavior for Org:Modular—the two movement modules and the copy module—are functionally and spatially modular. For all three of these modules, very little happens within them apart from the main function of the module. The modules are also spatially modular, *i.e.*, located in different areas of the genome. Example organisms from the preceding experiments also demonstrate some structural modularity, but their functional modularity is less well-defined. The parallel with the structural and functional modularity seen in the neural control of animal behavior is striking (Bullmore and Sporns, 2009). The second feature of special interest is the flexibility of execution flow between code modules. The execution flow enables Org:Modular to cleverly handle all the contingencies of the environment. For example, even though Module 3A (left-turn module) is encountered first in the sequential execution of the genome, if a right turn is encountered first, the execution flow moves easily through Module 3A into Module 3B (right-turn module). The algorithm evolved to deftly maneuver along the paths, using the information of the cues from the environment to alter its execution.

We presented a "bestiary" of digital organisms, case studies of three evolved organisms that show the range of surprising solutions that can arise in open-ended evolving systems. Each example organism had a striking characteristic that highlights issues of interest in both computer sci-

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2011/23/44/2022463/0262297140chap44.pdf by guest on 23 June 2024

Module 3A	Module 3B
sg-move	sg-rotate-r
sg-sense	if-label
sub	nop-B
if-less	add
swap	nop-B
h-divide	if-less
dec	sg-sense
if-less	sub
mov-head	sg-move
push	nand
if-label	sg-sense
nop-C	if-equ-X
shift-r	mov-head
nop-A	
sg-rotate-l	
if-equ-X	
if-less	
mov-head	

Table 3: Avida instructions for example irregular path organism, Org:Modular.

ence and biology. Although it is premature to make broad generalizations based on our results, we can conclude that the Avidians evolved solutions that were well tailored to the task, neither more nor less complex than needed. The lesson of our results is that an evolutionary approach to higher levels of intelligence will require careful attention to both the computational resources available to the evolving system and to the complexity of the tasks presented by the environment.

The work that we report in this paper laid the foundation for several ongoing research projects. We are continuing our study of evolving navigation, including simple landmark navigation and vector navigation. The experiments discussed in this paper provide an excellent arena for investigating issues relating to historical contingency in evolution: how accidental changes to the genetics of a population shape the path of future evolution. Steps in evolution are thus dependent on prior history (Blount et al., 2008). We are exploring what factors determined what strategy evolved in our experiments.

Our results underscore how results from Artificial Life experiments can provide insight for different fields of study. The strategies shown by these case study organisms highlight the power of evolution to find surprising and clever solutions to problems. These solutions may guide the development of intelligent artificial agents, and also provide insights into the fundamental principles governing the early evolution of intelligent behavior in biological systems.

Acknowledgments

We thank Philip McKinley, Wesley Elsberry, Jeff Clune, Michael Vo, Erica Rettig, and other members of the MSU Digital Evolution Laboratory for valuable insights and feedback. This work was supported by a grant from the Cambridge Templeton Consortium,"Emerging Intelligence: Contingency, Convergence and Constraints in the Evolution of Intelligent Behavior," a grant from the DARPA FunBio program, and NSF grants CCF-0643952 and DBI-0939454.

References

Adami, C., Ofria, C. A., and Collier, T. C. (2000). Evolution of biological complexity. *Proceedings of the National Academy of Science*, 97:4463–4468.

Bedau, M. A. (2007). Artificial life. In Matthen, M. and Stephens, C., editors, *Handbook of the Philosophy of Science: Philosophy of Biology*, pages 595–613. Elsevier, Boston.

Blount, Z. D., Borland, C. Z., and Lenski, R. E. (2008). Historical contingency and the evolution of a key innovation in an experimental population of *Escherichia coli*. *Proceedings of the National Academy of Science*, 105(23):7899–7906.

Bullmore, E. and Sporns, O. (2009). Complex brain networks: graph theoretical analysis of structural and functional systems. *Nature Reviews Neuroscience*, 10(3):186–198.

Grabowski, L. M., Bryson, D. M., Dyer, F. C., Ofria, C., and Pennock, R. T. (2010). Early evolution of memory usage in digital organisms. In *Artificial Life XII: Proceedings of the Twelfth International Conference on the Synthesis and Simulation of Living Systems*, pages 224–231, Cambridge, MA. MIT Press.

Lenski, R. E., Ofria, C., Pennock, R. T., and Adami, C. (2003). The evolutionary origin of complex features. *Nature*, 423:139–144.

Morgan, C. L. (1894). *An introduction to comparative psychology*. Scott, London.

Ofria, C., Adami, C., and Collier, T. C. (2002). Design of evolvable computer languages. *IEEE Transactions in Evolutionary Computation*, 17:528–532.

Ofria, C., Bryson, D. M., and Wilke, C. O. (2009). Artificial life models in software. In Adamatzky, A. and Komosinski, M., editors, *Advances in Artificial Life*, chapter Avida: A Software Platform for Research in Computational Evolutionary Biology, pages 3–36. Springer-Verlag, Berlin, 2nd edition.

Pennock, R. T. (2007). Models, simulations, instantiations, and evidence: the case of digital evolution. *Journal of Experimental and Theoretical Artificial Intelligence*, 19(1):29–42.

Wittlinger, M., Wehner, R., and Wolf, H. (2006). The ant odometer: Stepping on stilts and stumps. *Science*, 312(5782):1965–1967.

Zhang, S. W., Mizutani, A., and Srinivasan, M. V. (2000). Maze navigation by honeybees: learning path regularity. *Learning and Memory*, 7:363–374.

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2011/23/44/2022463/0262297140chap44.pdf by guest on 23 June 2024