1

Avida: Evolution Experiments with Self-Replicating Computer Programs

Charles Ofria and Claus O. Wilke

Avida* is a software platform for experiments with self-replicating and evolving computer programs. It provides detailed control over experimental settings and protocols, a large array of measurement tools, and sophisticated methods to analyze and postprocess experimental data. This chapter explains the general principles on which Avida is built, its main components and their interactions, and gives an overview of some prior research with Avida.

1.1 Introduction to Avida

When studying biological evolution, we have to overcome a large obstacle: Evolution happens extremely slowly. Traditionally, evolution has therefore been a field dominated by observation and theory, even though some regard the domestication of plants and animals as early, unwitting evolution experiments. Realistically, we can carry out controlled evolution experiments only with organisms that have very short generation times, so that populations can undergo hundreds of generations within a timeframe of months to a few years. With the advances in microbiology, such experiments in evolution have become feasible with bacteria and viruses [14, 39]. However, even with microorganisms, evolution experiments still take a lot of time to complete and are often cumbersome to carry out. In particular, certain data can be difficult or impossible to obtain, and it is often impractical to carry out enough replicas for high statistical accuracy.

According to Daniel Dennett, "evolution will occur whenever and wherever three conditions are met: replication, variation (mutation), and differential fitness (competition)" [9]. It seems to be an obvious idea to set up these conditions in a computer and to study evolution *in silico* rather than *in vitro*.

* Parts of the material in this chapter previously appeared in Artificial Life 10:191-229 (2004) by the chapter authors, and in the Avida documentation, whose coauthors also include C. Adami, R. Lenski and K. Nanlohy.

In a computer, it is easy to measure any quantity of interest with arbitrary precision, and the time it takes to propagate organisms for several hundred generations is only limited by the processing power available. In fact, population geneticists have long been carrying out computer simulations of evolving loci, in order to test or augment their mathematical theories (see [17,23,31] for some recent examples). However, the assumptions put into these simulations typically mirror exactly the assumptions of the analytical calculations. Therefore, the simulations can be used only to test whether the analytic calculations are error-free, or whether stochastic effects cause a system to deviate from its deterministic description, but they cannot test the model assumptions on a more basic level.

An approach to studying evolution that lies somewhere in between evolution experiments with biochemical organisms and standard Monte Carlo simulations is the study of self-replicating and evolving computer programs (digital organisms). These digital organisms can be quite complex and interact in a multitude of different ways with their environment or each other, so that their study is not a simulation of a particular evolutionary theory but becomes an experimental study in its own right. In recent years, research with digital organisms has grown substantially ([2, 13, 21, 43, 45, 46]; see [41] for a recent review), and is being increasingly accepted by evolutionary biologists [30]. (However, as Barton and Zuidema [3] note, general acceptance will ultimately hinge on whether artificial life researchers embrace or ignore the large body of population-genetics literature.) Avida is arguably the most advanced software platform to study digital organisms to date and is certainly the one that has had the biggest impact in the biological literature so far. Having reached version 2.0, it now supports detailed control over experimental settings, a sophisticated system to design and execute experimental protocols, a multitude of possibilities for organisms to interact with their environment, including depletable resources and conversion from one resource into another, and a module to postprocess data from evolution experiments, including tools to find the line of descent from final organisms to their ultimate ancestor, to carry out knock-out studies with organisms, and to align and compare organisms' genomes.

1.1.1 History of Digital Life

The most well-known intersection of evolutionary biology with computer science is the genetic algorithm or its many variants (genetic programming, evolutionary strategies, and so on). All these variants boil down to the same basic recipe: (1) Create random potential solutions; (2) evaluate each solution assigning it a fitness value to represent its quality; (3) select a subset of solutions using fitness as a key criterion; (4) vary these solutions by making random changes or recombining portions of them; (5) repeat from step 2 until you find a solution that is sufficiently good.

This technique turns out to be an excellent method for solving problems, but it ignores many aspects of natural living systems. Most notably, natural organisms must replicate themselves, as there is no external force to do so; therefore, their ability to pass their genetic information on to the next generation is the final arbiter of their fitness. Furthermore, organisms in a natural system have the ability to interact with their environment and with each other in ways that are excluded from most algorithmic applications of evolution.

Work on more naturally evolving computational systems began in 1990, when Steen Rasmussen was inspired by the computer game "Core War" [10]. In this game, programs are written in a simplified assembly language and made to compete in the simulated core memory of a computer. The winning program is the one that manages to shut down all processes associated with its competitors. Rasmussen observed that the most successful of these programs were the ones that replicated themselves, so that if one copy were destroyed, others would still persist. In the original Core War game, the diversity of organisms could not increase, and hence no evolution was possible. Rasmussen then designed a system similar to Core War in which the command that copied instructions was flawed and would sometimes write a random instruction instead on the one intended [33]. This flawed copy command introduced *mutations* into the system, and thus the potential for evolution. Rasmussen dubbed his new program "Core World," created a simple self-replicating ancestor, and let it run.

Unfortunately, this first experiment failed. While the programs seemed to evolve initially, they soon started to copy code into each other, to the point where no proper self-replicators survived — the system collapsed into a nonliving state. Nevertheless, the dynamics of this system turned out to be intriguing, displaying the partial replication of fragments of code, and repeated occurrences of simple patterns.

The first successful experiment with evolving populations of self-replicating computer programs was performed the following year. Thomas Ray at the University of Delaware designed a program of his own with significant, biologically inspired modifications. The result was the Tierra system [34]. In Tierra, digital organisms must allocate memory before they have permission to write to it, which prevents stray copy commands from killing other organisms. Death only occurs when memory fills up, at which point the oldest programs are removed to make room for new ones to be born.

The first Tierra experiment was initialized with an ancestral program that was 80 lines long. It filled up the available memory with copies of itself, many of which had mutations that caused a loss of functionality. Yet other mutations were actually neutral and did not affect the organism's ability to replicate — and a few were even beneficial. In this initial experiment, the only selective pressure on the population was for the organisms to increase their rate of replication. Indeed, Ray witnessed that the organisms were slowly shrinking the length of their genomes, since a shorter genome meant that there was less genetic material to copy, and thus it could be copied more rapidly.

This result was interesting enough on its own. However, other forms of adaptation, some quite surprising, occurred as well. For example, some organisms were able to shrink further by removing critical portions of their genome and then use those same portions from more complete competitors, in a technique that Ray noted was a form of parasitism. Arms races transpired where hosts evolved methods of eluding the parasites, and they, in turn, evolved to get around these new defenses. Some would-be hosts, known as hyperparasites, even evolved mechanisms for tricking the parasites into aiding them in the copying of their own genomes. Evolution continued in all sorts of interesting manner, making Tierra seem like a choice system for experimental evolution work.

In 1992, Chris Adami began research on evolutionary adaptation with Ray's Tierra system. His intent was to get these digital organisms to evolve solutions to specific mathematical problems, without forcing them use a predefined approach. His core idea was the following: If he wanted a population of organisms to evolve, for example, the ability to add two numbers together, he would monitor organisms' input and output numbers. If an output ever was the sum of two inputs, the successful organisms would receive extra CPU cycles as a bonus. As long as the number of extra cycles was greater than the time it took the organism to perform the computation, the leftover cycles could be applied toward the replication process, providing a competitive advantage to the organism. Sure enough, Adami was able to get the organisms to evolve some simple tasks, but he faced many limitations in trying to use Tierra to study the evolutionary process.

In the summer of 1993, Charles Ofria and C. Titus Brown joined Adami to develop a new digital life software platform, the Avida system. Avida was designed to have detailed and versatile configuration capabilities, along with high-precision measurements to record all aspects of a population. Furthermore, whereas organisms are executed sequentially in Tierra, the Avida system simulates a parallel computer, wherein all organisms are executed effectively simultaneously. Since its inception, Avida has had many new features added to it, including a sophisticated environment with localized resources, an events system to schedule actions to occur over the course of an experiment, multiple types of CPUs to form the bodies of the digital organisms, and a sophisticated analysis mode to postprocess data from an Avida experiment. Avida is still under active development at both Michigan State University, led by Charles Ofria, and at the California Institute of Technology, led by Claus Wilke.

1.1.2 The Scientific Motivation for Avida

Intuitively, it seems that natural systems should be used to best understand how evolution produces complexity in nature, but this can be prohibitively difficult for many questions and does not provide enough detail. Using digital organisms in a system such as Avida can be justified on five grounds:

1. *Artificial life forms provide an opportunity to seek generalizations about self-replicating systems* beyond the organic forms that biologists have studied to date, all of which share a common ancestor and essentially the same chemistry of DNA, RNA, and proteins. As John Maynard Smith [22] made the case: "So far, we have been able to study only one evolving system and we cannot wait for interstellar flight to provide us with a second. If we want to discover generalizations about evolving systems, we will have to look at artificial ones." Of course, digital systems should always be studied in parallel with natural ones, but any differences we find between their evolutionary dynamics open up what is perhaps an even more interesting set of questions.

2. *Digital organisms enable us to address questions that are impossible to study with organic life forms.* For example, in one of our current experiments we are investigating the importance of genetic drift to the evolution of complexity by explicitly reverting all neutral mutations while leaving both beneficial and deleterious mutations unaffected. Such invasive micromanaging of a population is not possible in a natural system, especially without disturbing other aspects of the evolutionary process. In a digital evolving system, every bit of memory can be viewed without disrupting the system, and changes can be made at the precise points desired.

3. *Other questions can be addressed on a scale that is unattainable with natural organisms.* In an earlier experiment with digital organisms [20], we examined billions of genotypes to quantify the effects of mutations as well as the form and extent of their interactions. By contrast, an experiment with *E. coli* was necessarily confined to one level of genomic complexity. Digital organisms also have a speed advantage: A population with 10,000 organisms can have 20,000 generations processed per day on a modern desktop computer. A similar experiment with bacteria took over a decade [19].

4. *Digital organisms possess the ability to truly evolve, unlike mere numerical simulations.* Evolution is open-ended and the design of the evolved solutions is unpredictable. These properties arise because selection in digital organisms (as in real ones) occurs at the level of the whole-organism's phenotype; it depends on the rates at which organisms perform tasks that enable them to metabolize resources to convert them to energy, and the efficiency with which they use that energy for reproduction. Genome sizes are sufficiently large that evolving populations cannot test every possible genotype, so replicate populations always find different local optima. A genome typically consists of 50 to 1000 sequential instructions. With 26 possible instructions at each position, there are many more potential genome states than there are atoms in the universe.

5. *Digital organisms can be used to design solutions to computational problems* where it is difficult to write explicit programs that produce the desired behavior [15, 18]. Current evolutionary algorithm approaches are based on a simplistic view of evolution, leaving out many of the factors that are believed to make it such a powerful force. Thus there are new opportunities for biological concepts to have a large impact outside of biology, just as principles

of physics and mathematics are often used throughout other fields, including biology.

1.2 The Avida Software

The Avida software is composed of three main modules: The first is the *Avida core*, which maintains all of the key components needed for an experiment to run without user interaction, including a population of digital organisms (each with their own genomes, virtual hardware, etc.), an environment that determines the reactions and resources with which the organisms interact, a scheduler to allocate CPU cycles to the organisms, and various data collection objects. The second module is the *graphical user interface* (GUI) that the researcher can use to observe and interact with the rest of the Avida software, including a population monitor, graphing utilities, and other tools to measure or alter quantities in a population. The final component is a collection of *analysis and statistics* tools, including a test environment to study organisms outside the population, data manipulation tools to rebuild phylogenies and examine lines of descent, mutation and local fitness analysis tools, and many others, all bound together in a simple scripting language. A fourth module, an interactive help and documentation system, is currently under development.

In this section, we will discuss the core module of Avida, which is the only one needed to perform experiments with digital organisms. In the next section, we will go into the user tools to interact with an Avida population (the user interface) and postprocess the data that comes out of an experiment (the analyze mode).

1.2.1 Avida Organisms

In Avida, each digital organism is a self-contained computing automaton that has the ability to construct new automata. The organism is responsible for building the genome (computer program) that will control the offspring automaton and for transferring that genome to the Avida world. Avida will then construct virtual hardware for the genome to be run on, and determine how this new organism should be placed into the population. In a typical Avida experiment, a successful organism attempts to make an identical copy of its own genome, and Avida randomly places that copy into the population, typically by replacing another member of the population.

In principle, the only assumption made about these self-replicating automata in the core Avida software is that their initial state can be described by a string of symbols (their genome) and that it is possible through processing these symbols to autonomously produce offspring organisms. However, in practice, our work has focused on automata with a simple von Neumann architecture that operate on an assembly-like language inspired by the Tierra

system. Future research projects will likely have us implement additional organism instantiations to allow us to explore additional biological questions.

In the following sections, we describe the default hardware of our virtual computers and explain the principles of the language these machines work on.

Virtual Hardware

The structure of a virtual machine in Avida is depicted in Fig. 1.1. The core of the machine is the central processing unit (CPU), which processes each instruction in the genome and modifies the states of its components appropriately. Mathematical operations, comparisons, and so on can be done on three registers: `AX`, `BX`, and `CX`. These registers each store and manipulate data in in the form of a single, 32-bit number. The registers behave identically, but different instructions may act on different registers by default (see below). The CPU also has the ability to store data in two stacks. Only one of the two stacks is active at a time, but it is possible to switch the active stack, so that both stacks are accessible.

The program memory is initialized with the genome of the organism. Execution begins with the first instruction in memory and proceeds sequentially: Instructions are executed one after the other, unless an instruction (such as a jump) explicitly interrupts sequential execution. Technically, the memory space is organized in a circular fashion, such that after the CPU executes the last instruction in memory, it will loop back and continue execution with the first instruction again. However, at the same time the memory has a well-defined starting point, important for the creation and activation of offspring organisms.

Somewhat out of the ordinary in comparison to standard von Neumann architectures are the four CPU components labeled *heads*. Heads are essentially pointers to locations in the memory. They remove the need of absolute addressing of memory positions, which makes the evolution of programs more robust to size changes that would otherwise alter these absolute positions. Among the four heads, only one, the instruction head, has a counterpart in standard computer architectures. The instruction head corresponds to the instruction pointer in standard architectures and identifies the instruction currently being executed by the CPU. It moves one instruction forward whenever the execution of the previous instruction has been completed, unless that instruction specifically moved the instruction head elsewhere.

The other three heads (the read head, the write head, and the flow control head) are unique to the Avida virtual hardware. The read and write heads are used in the self-replication process. In order to generate a copy of its genome, an organism must have a means of reading instructions from memory and writing them back to a different location. The read head indicates the position in memory from which instructions are currently being read, and the write head likewise indicates the position to which instructions are currently being written. The positions of all four heads can be manipulated with special

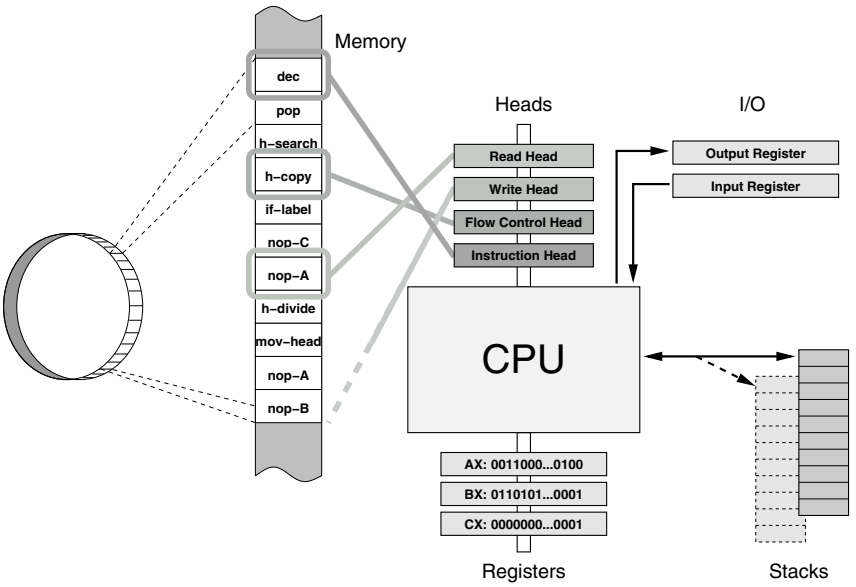

Fig. 1.1. The standard virtual hardware in Avida: CPU, registers, stacks, heads, memory, and I/O functionality.

commands. In that way a program can position the read and write heads appropriately in order to self-replicate.

The flow control head is used for jumps and loops. Several commands will reposition the flow control head, and other commands will move specific heads to the same position in memory as the flow control head.

Finally, the virtual machines have an input buffer and an output buffer, which they use to interact with their environment. The way in which this communication works is that the machines can read in one or several numbers from the input buffer, perform computations on these numbers with the help of the internal registers `AX`, `BX`, `CX`, and the stacks, and then write the results to the output buffer. This interaction with the environment plays a crucial role in the evolution of Avida organisms and will be explained in detail in Sec. 1.2.2.

Genetic Language

It is important to understand that there is not a single language that controls the virtual hardware of an Avida organism. Instead, we have a collection of different languages. The virtual hardware in its current form can execute hundreds of different instructions, but only a small fraction of them are used in a typical experiment. The instructions are organized into subsets of the full range of implemented instructions. We call these subsets *instruction sets*.

Each instruction set forms a logical unit and can be considered a complete genetic programming language.

Each instruction has a well-defined function in any context, that is, there are no syntactically incorrect programs. Instructions do not have arguments per se, but the behavior of certain instructions can be modified by succeeding instructions in memory. A genome is therefore nothing more than a sequence of symbols in an alphabet composed of the instruction set, similar to how DNA is a sequence made up of 4 nucleotides or proteins are sequences with an alphabet of 20 amino acids.

Here, we will give an overview of the default instruction set, which contains 26 instructions. This set is explained in greater detail in the Avida documentation, for those who wish to work with it.

Template Matching and Heads: One important ingredient of most Avida instruction sets is the concept of template matching. Template matching is a method of indirectly addressing a position in memory. This method is similar to the use of labels in many programming languages: Labels tag a position in the program, so that jumps and function calls always go to the correct place, even when other portions of the source code are edited. The same reasoning applies to Avida genomes, because mutations may cause insertions or deletions of instructions that shift the position of code segments and would otherwise jeopardize the positions referred to. Since there are no arguments to instructions, positions in memory are determined by series of subsequent instructions. We refer to a series of instructions that indicates a position in the genome as a *template*.

Template-based addressing works as follows. When an instruction is executed that must reference another position in memory, subsequent *nop* instructions (described ahead) are read in as the template. The CPU then searches linearly through the genome for the first occurrence of the complement to this template and uses the end of the complement as the position needed by the instruction. Both the direction of the search (forward or backward from the current instruction) and the behavior of the instruction if no complement is found are defined specifically for each instruction.

Avida templates are constructed out of no-operation (nop) instructions; that is, instructions that do not alter the state of either CPU or memory when they are directly executed. There are three template-forming NOPs, `nop-A`, `nop-B`, and `nop-C`. They are circularly complementary, i.e., the complement of `nop-A` is `nop-B`, the complement of `nop-B` is `nop-C`, and the complement of `nop-C` is `nop-A`. A template is composed of consecutive nops only. A template will end with the first non-nop instruction.

Nonlinear execution of code ("jumps") has to be implemented through clever manipulation of the different heads. This happens in two stages. First, the instruction `h-search` is used to position the flow control head at the desired position in memory. Then, the instruction head is moved to that position with the command `mov-head`. Figure 1.2 shows an example of this.

```
      ⋮
      ⋮                 Some code.
10    h-search          Prepare the jump by placing the
                        flow control head at the end of the
                        complement template in forward direction.
11    nop-A             This is the template. Let's call it α.
12    nop-B
13    mov-head          The actual jump. Move the flow control head
                        to the position of the instruction head.
14    pop               Some other code that is skipped.
      ⋮
      ⋮
18    nop-B             The complement template ᾱ.
19    nop-C
      ⋮
      ⋮                 The program continues ...
```

Fig. 1.2. Example code demonstrating flow control with heads-based instruction set.

Although this example looks somewhat awkward on first glance, evolution of control structures such as loops are actually facilitated by this mechanism. In order to loop over some piece of code, it is only necessary to position the flow control head correctly once and to have the command `mov-head` at the end of the block of code that should be looped over. Since there are several ways in which the flow control head can be positioned correctly, of which the above example is only a single one, there are many ways in which loops can be generated.

Nop's as Modifiers: The instructions in the Avida programming language do not have arguments in the usual sense. However, as we have seen above for the case of template matching, the effect of certain instructions can be modified if they are immediately followed by nop instructions. A similar concept exists for operations that access registers. The `inc` instruction, for example, increments a register by one. If `inc` is not followed by any nop, then by default it acts on the `BX` register. However, if a nop is present immediately after the `inc`, then the register on which `inc` acts is specified by the type of the nop. For example, `inc nop-A` increments the `AX` register and `inc nop-C` the `CX` register. Of course, `inc nop-B` increments the `BX` register, and hence works identical to a single `inc` command. Similar nop modifications exist for a range of instructions, such as those that perform arithmetic like `inc` or `dec`, stack operations such as `push` or `pop`, and comparisons such as `if-n-equ`. The details for specific instructions can be found in [29] or in the Avida documentation. For some instructions that work on two registers, in particular comparisons, the concept of the complement nop is important, because the two registers are specified in this way. Similarly to the nops in the template matching, registers are cyclically complementary to each other, i.e., `BX` is the complement to `AX`,

CX to BX, and AX to CX. The instruction if-n-equ, for example, acts on a register and its complement register. By default, if-n-equ compares whether the contents of the BX and CX registers are identical. However, if if-n-equ is followed by a nop-A, then it will compare AX and BX. Figure 1.3 shows a piece of example code that demonstrates the principles of nop modification and complement registers.

```
01  pop       We assume the stack is empty. In that case,
              the pop returns 0, which is stored in BX.
02  pop       Write 0 into the register AX as well.
03  nop-A
04  inc       Increment BX.
05  inc       Increment AX.
06  nop-A
07  inc       Increment AX a second time.
08  nop-A
09  swap      The swap command exchanges the contents
              of a register with that of its complement
              register. Followed by a nop-C, it exchanges
              the contents of AX and CX. Now, BX= 1, CX= 2,
              and AX is undefined.
10  nop-C
11  add       Now add BX and CX and store the result
              in AX.
12  nop-A     The program continues with BX= 1, CX= 2,
              and AX= 3.
    ⋮
```

Fig. 1.3. Example code demonstrating the principle of nop modification.

Nop modification is also necessary for the manipulation of heads. The instruction mov-head, for example, by default moves the instruction head to the position of the flow control head. However, if it is followed by either a nop-B or a nop-C, it moves the read head or the write head, respectively. A nop-A following a mov-head leaves the default behavior unaltered.

Memory Allocation and Division: When a new Avida organism is created, the CPUs memory is exactly the size of its genome, i.e., there is no additional space that the organism could use to store its offspring-to-be as it makes a copy of its program. Therefore, the first thing an organism has to do at the start of self-replication is to allocate new memory. In the default instruction set, memory allocation is done with the command h-alloc. This command extends the memory by the maximal size that a child organism is allowed to have. As we will discuss later, there are some restrictions on how large or small a child organism is allowed to be in comparison to the parent organism, and the restriction on the maximum size of a child organism determines the amount of

memory that `h-alloc` adds. The allocation happens always at a well-defined position in the memory. Although the memory is considered to be circular in the sense that the CPU will continue with the first instruction of the program once it has executed the last one, the virtual machine nevertheless keeps track of which instruction is the beginning of the program, and which is the end. By default, `h-alloc` (as well as all alternative memory allocation instructions, such as the old `allocate`) insert the new memory between the end and the beginning of the program. After the insertion, the new end is at the end of the inserted memory. The newly inserted memory is either initialized to a default instruction, typically `nop-A`, or to random code, depending on the choice of the experimenter.

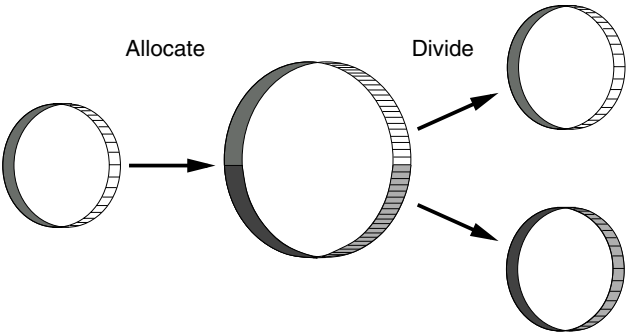

Fig. 1.4. The `h-alloc` command extends the memory, so that the program of the child organism can be stored. Later, on `h-divide`, the program is split into two parts, one of which turns into the child organism.

Once an organism has allocated memory, it can start to copy its program code into the newly available memory block. This copying is done with the help of the control structures we have already described, in conjunction with the instruction `h-copy`. This instruction copies the instruction at the position of the read head to the position of the write head and advances both heads. Therefore, for successful self-replication an organism mainly has to assure that initially, the read head is at the beginning of the memory, and the write head is at the beginning of the newly allocated memory, and then it has to call `h-copy` for the correct number of times.

After the self-replication has been completed, an organism issues the `h-divide` command, which splits off the instructions between the read head and the write head, and uses them as the genome of a new organism. The new organism is handed to the Avida world, which takes care of placing it into a suitable environment and so on. If there are instructions left between the write head and the end of the memory, these instructions are discarded, so that only the part of the memory from the beginning to the position of the read head remains after the divide.

In most natural asexual organisms, the process of division results in organisms literally splitting in half, effectively creating two offspring. As such, the default behavior of Avida is to reset the state of the parent's CPU after the divide, turning it back into the state it was in when it was first born. In other words, all registers and stacks are cleared, and all heads are positioned at the beginning of the memory. The allocation and division cycle is illustrated in Fig. 1.4.

Not all `h-divide` commands that an organism issues lead necessarily to the creation of an offspring organism. There are a number of conditions that have to be satisfied; otherwise the command will fail. Failure of a command means essentially that the command is ignored, while a counter keeping track of the number of failed commands in an organism is increased. It is possible to configure Avida to punish organisms with failed commands. The following conditions are in place: An `h-divide` fails if either the parent or the offspring would have less than 10 instructions, the parent has not allocated memory, less than half of the parent was executed, less than half of the offspring's memory was copied into, or the offspring would be too small or to large (as defined by the experimenter).

Mutations

So far, we have described all the elements that are necessary for self-replication. However, self-replication alone is not sufficient for evolution. There must be a source of variation in the population, which comes from random *mutations*.

The main form of mutations in Avida are so-called copy mutations, which arise through erroneously copied instructions. Such miscopies are a built-in property of the instruction `h-copy`. With a certain probability, chosen by the experimenter, the command `h-copy` does not properly copy the instruction at the location of the read head to the location of the write head, but instead writes a random instruction to the position of the write head. It is important to note that the instruction written will always be a legal one, in the sense that the CPU can execute it. However, the instruction may not be meaningful in the context in which it is placed in the genome, which in the worst case can render the offspring organism nonfunctional.

Another commonly used source of mutations are insertion and deletion mutations. These mutations are applied on `h-divide`. After an organism has successfully divided off a child organism, an instruction in the child's memory may by chance be deleted, or a random instruction may be inserted. The probabilities with which these events occur are again determined by the experimenter. Insertion and deletion mutations are useful in experiments in which frequent changes in genome size are desired. Two types of insertion/deletion mutations are available in the configuration files; they differ in that one is a genome-level rate and the other is a per-site rate.

Next, there are point (or cosmic ray) mutations. These mutations affect not only organisms as they are being created (like the other types described

above), but all living organisms. Point mutations are random changes in the memory of the virtual machines. One of the consequences of point mutations is that a program may change while it is being executed. In particular, the longer a program runs, the larger target it is for point mutations. This is in contrast to copy or insertion and deletion mutations, whose impact depends only on the length of the program, but not on the execution time.

Finally, it is important to note that organisms in Avida can also have *implicit* mutations. Implicit mutations are modifications in a child's program that are not directly caused by any of the external mutation mechanisms described above, but rather by an incorrect copy algorithm of the parent organism. For example, the copy algorithm might skip some instructions of the parent program, or copy a section of the program twice (effectively a gene duplication event). Another example is an incorrectly placed read head or write head on divide. Implicit mutations are the only ones that cannot easily be controlled by the experimenter. They can, however, be turned off completely by using the FAIL_IMPLICIT option in the configuration files, which gets rid of any offspring that will always contain a deterministic difference from its parent, as opposed to one that is associated with an explicit mutation.

Phenotype

Each organism in an Avida population has a phenotype associated with it. Phenotypes of Avida organisms are defined in the same way as they are defined for organisms in the natural world: The phenotype of an organism comprises all observable characteristics of that organism. As an organism in Avida goes through its life cycle, it will self-replicate and, at the same time, interact with the environment. The primary mode of environmental interaction is by inputting numbers from the environment, performing computations on those numbers, and outputting the results. The organisms receive a benefit for performing specific computations associated with resources as determined by the experimenter.

In addition to tracking computations, the phenotype also monitors several other aspects of the organism's behavior, such as the organism's gestation length (the number of instructions the organism executes to produce an offspring, often also called *gestation time*), its age, if it has been affected by any mutations, how it interacts with other organisms, and its overall fitness. These data are used to determine how many CPU cycles should be allocated to the organism and also for various statistical purposes.

Genotypes

In Avida, organisms can be classified into several taxonomic levels. The lowest, but most important taxonomic level is called *genotype*. All organisms that have exactly the same initial genomes are considered to have the same genotype. Certain statistical data are collected only at the genotype level. We pay special

attention to the most abundant genotype in the population — the *dominant* genotype — as a method of determining what the most successful organisms in the population are capable of. If a new genotype is truly more fit than than the dominant one, organisms with this higher fitness will rapidly take over the population.

We classify a genotype as *threshold* if there are three or more organisms that have ever existed of that genotype (again, the value 3 is not hard-coded, but configurable by the experimenter). Often, deleterious mutants appear in the population. These mutants are effectively dead and disappear again in short order. Since these mutants are not able to successfully self-replicate (or at least not well), there is a low probability of them reaching an abundance of three. As such, for any statistics we want to collect about the *living* portion of the population, we focus on those organisms whose genotype has the threshold characteristic.

1.2.2 The Avida World

In general, the Avida world has a fixed number N of positions or *cells*. Each cell can be occupied by exactly one organism, such that the maximum population size at any given time is N. Each of these organisms is being run on a virtual CPU, and some of them may be running faster than others. Avida has a scheduler (see below) that divides up time from the real CPU such that these virtual CPUs execute in a simulated parallel fashion.

While an Avida organism runs, it may interact with the environment or other organisms. When it finally reproduces, it hands its offspring organism to the Avida world, which places the newborn organism into either an empty or an occupied cell, according to rules we describe ahead. If the offspring organism is placed into an already occupied cell, the organism currently occupying that cell is killed and removed, irrespective of whether it has already reproduced or not.

Scheduling

In the simplest of Avida experiments, all virtual CPUs run at the same speed. This method of time sharing is simulated by executing one instruction on each of the N virtual CPUs in order, then starting over to execute a second instruction on each one, and so on. An *update* in Avida is defined as the point where the average organism has executed k instructions (where $k = 30$ by default). In this simple case, for one update we carry out k rounds of execution.

In more complex environments, however, the situation is not so trivial. Different organisms will have their virtual CPUs running at different speeds, and the scheduler must portion out cycles appropriately to simulate that all CPUs are running in parallel. Each organism has associated with it a value called *merit*. The merit indicates how fast the CPU should run. Merit is a

unitless quantity and is only meaningful when compared to the merits of other organisms. Thus, if organism A has twice the merit of organism B, then A should execute twice as many instructions in any given timeframe as B.

Avida handles this with two different schedulers. The first one is a perfectly integrated scheduler, which comes as close as possible to portioning out CPU cycles proportional to merit. Obviously, only whole time steps can be used; therefore, perfect proportionality is not possible in general for small timeframes. For timeframes long enough such that the granularity of individual time steps can be neglected, the difference between the number of cycles given to an organism and the number of cycles the organism should receive according to its merit is negligible.

The second scheduler is probabilistic. At each point in time, the next organism to be selected is chosen at random, with a probability of being chosen proportional to its merit. Thus, *on average* this scheduler is perfect, but there are no guarantees.

In practice, the perfectly integrated scheduler is faster, but occasionally can cause odd effects, because it is possible for the organisms to become synchronized, particularly at low mutation rates where a single genotype can represent a large portion of the population. The stochastic scheduler may be preferred for projects where this effect might be a problem. By default, Avida uses the perfectly integrated scheduler.

World Topologies and Birth Methods

The N cells of the Avida world can be assembled into different topologies that affect how offspring organisms are placed and how organisms interact (as described ahead). Currently, there are two world topologies: a 2D grid with Moore neighborhood (each cell has 8 neighbors) and a fully connected (sometimes called *well-stirred* or *mass action*) topology. In the latter, fully connected topology, each cell is a neighbor to every other cell. New topologies can easily be implemented by listing the neighbors associated with each cell (though more work might need to be done in the user interface to properly visualize the new topology).

When a new organism is about to be born, it will replace either the parent cell or another cell from the neighborhood. The specifics of this placement are set up by the experimenter. The two most commonly used methods are *replace random*, which chooses randomly from the neighborhood, or *replace oldest*, which picks the oldest organism from the neighborhood to replace (with a preference for empty cells if any exist).

Fully connected topologies are used in analogy to experiments with microbes in well-stirred flasks or chemostats. These setups allow for exponential growth of new genotypes with a competitive advantage, so that transitions in the state of the population can happen rapidly. Local neighborhoods, on the other hand, are more akin to a Petri dish, and the spatial separation between

different organisms puts limits on growth rates and allows for a slightly more diverse population [5].

In choosing which organism in a neighborhood to replace, a random placement matches up well with the behavior of a chemostat, where a random portion of the population is continuously drawn out to keep population size constant. Experiments have shown [1], however, that evolution occurs more rapidly when the oldest organism in a neighborhood is the first to be killed off. In such cases, all organisms are given approximately the same chance to prove their worth, whereas in random replacement, about half the organisms are killed before they have the opportunity to produce a single offspring. Interestingly, when replace oldest is used in 2D neighborhoods, 40% of the time it is the parent that is killed off. This observation makes sense, because the parent is definitely old enough to have produced at least one offspring.

Note that in the default setup of Avida, the only way for an organism to die is for it to be replaced by another organism being born. It is also possible to enable an independent death method that will kill off an organism after it has executed a specified number of instructions, which can be either a constant or proportional to the organism's genome length. In some cases a population without any form of death turned on can lose all ability to self-replicate, but persist since organisms have no way of being purged. This situation can lead to confusing results for the research if the cause is not identified.

Environment and Resources

All organisms in Avida are provided with the ability to absorb a default resource that gives them their base merit. An Avida environment can, however, contain other resources that the organisms can absorb to modify their merit. The organisms absorb a resource by carrying out the corresponding computation or *task*.

An Avida environment is described by a set of resources and a set of reactions that can be triggered to interact with those resources. A reaction is defined by a computation that the organism must perform to trigger it, a resource that is consumed by it, a merit effect on the organism (which can be proportional to the amount of resource absorbed or available), and a byproduct resource if one should be produced. Reactions can also have restrictions associated with them that limit when a trigger will be successful. For example, another reaction can be required to have been triggered first, or a limit can be placed on the number of times an organism can trigger a certain reaction.

A resource is described by an initial quantity (which can be infinite if a resource should not be depletable), an inflow rate (the amount of that resource that should come into the population per update), and an outflow rate (the fraction of the resource that should be removed each update). If resources are made to be depletable, then the more organisms trigger a reaction, the less of

that resource is available for each of them. This setup allows multiple, diverse subpopulations to stably coexist in an Avida world [4, 6].

The default Avida environment rewards nine boolean logic operations, each associated with a nondepletable resource, but organisms can receive only one reward per computation. Other prebuilt environments that come with Avida include one with 78 different logic operations rewarded, one similar to the default nine-resource environment, but with the resources set up to be depletable, with fixed inflow and outflow rates, and one with nine computations rewarded, and where only the resources associated with the simplest computations have an inflow into the system, and those for more complex operations are produced as byproducts, in sequence, from the reactions using up resources associated to simpler computations.

An important aspect of Avida is that the environment does not care *how* a computation is performed, only that the output of the organism being tested is correct given the inputs it took in. As a consequence, the organisms find a wide variety of ways of computing their outputs, some of which can be surprising to a human observer, seeming to be almost inspired.

Even though organisms can carry out tasks and collect rewards at any time in their gestation cycle, these rewards do not immediately affect the speed at which their CPU runs. The CPU speed (merit) is set only once, at the beginning of the gestation cycle, and then held constant until the organism divides. At that point, both the organism and its offspring get a new merit, which reflects the bonuses the organism collected during the gestation cycle it just completed. In a sense, the organisms collect rewards that go to their offspring rather than for themselves. The reason why we do not change an organism's merit during its gestation cycle is to level the playing field between old and young organisms. If organisms were always born with a low initial CPU speed, then they might never execute enough instructions to carry out tasks in the first place. At the same time, mutants specialized in carrying out tasks but not dividing could concentrate all CPU time on them, thus effectively shutting down replication in the population. It can be shown that the average fitness of a population in equilibrium is independent of whether organisms get the bonuses directly or collect them for their offspring [40].

Organism Interactions

As explained above, populations in Avida have a firm cap on their size, which makes space the fundamental resource that the organisms must compete for. In the simplest Avida experiments, the only interaction between organisms is that an organism is killed when another gives birth, in order to make room for the offspring. In slightly more complex experiments, the organisms are rewarded with a higher merit and hence a larger share of the CPU cycles for performing tasks. Since only a fixed number of CPU cycles is given out each update, the competition for them becomes a second level of indirect interactions among the organisms. As the environment becomes more complex

still, multiple resources take the place of fixed merit bonuses for performing tasks, and the organisms must now compete over each of these resources independently. In the end, however, all these interactions boil down to the indirect competition for space: More resources imply a higher merit, which in turn grants the organisms a larger share of the CPU cycles, allowing them to replicate more rapidly and claim more space for their genotype.

In most Avida experiments, indirect competition for space is the only level of interaction we allow; organisms are not allowed to directly write to or read from each other's genomes, so that Tierra-style parasites cannot form (although the configuration files do allow the experimenter to enable them). The more typical way of allowing parasites in Avida is to enable the `inject` command in the Avida instruction set. This command works similar to divide, except that instead of replacing an organism in a target cell, the would-be offspring is inserted into the memory of the organism occupying the target cell; the specific position in memory to which it is placed is determined by the template that follows the `inject`.

In Tierra, parasites can replicate more rapidly than nonparasites, but an individual parasite poses no direct harm to the host whose code it uses. These organisms could, therefore, be thought of more directly as cheaters in the classic biological sense, as they effectively take advantage of the population as a whole. In Avida, a parasite exists directly inside its host and makes use of the CPU cycles that would otherwise belong to the host, thereby slowing down the host's replication rate. Depending on the type of parasite, it can either take all of the host's CPU cycles (thereby killing the host) and use them for replicating and spreading the infection, or else spread more slowly by using only a portion of the hosts CPU cycles (sickening it), but reducing the probability of driving the hosts — and hence itself — into extinction.

In the future, we plan to implement two other forms of interactions. First, we plan to implement *sensors* with which organisms can detect the presence of resources, which would allow them to exchange chemical signals. Second, we are considering more direct *communication*, whereby the organisms can send numbers to each other, and possibly distribute computations among themselves to solve environmental challenges more rapidly.

1.2.3 Test Environments

Often when examining populations in Avida, the user will need to know the fitness or some other characteristic of an organism that has not yet gone through a full gestation cycle during the course of the experiment. For this reason, we have constructed a *test environment* for the organisms to be run in, without affecting the rest of the population. This test environment will run the organism for at least one generation and can be used either during a run or as part of post-processing.

When an organism is loaded into a test environment, its instructions are executed until it produces a viable offspring or until a timeout is reached.

Unfortunately, it is not possible to guarantee identification of nonreplicative organisms (this is known as the halting problem in computer science), so at some point we must give up on any program we are testing and assume it to be dead. If age-based death is turned on in the actual population, this becomes a good limit for how long a CPU in the test environment should be run.

The fact that we want to determine if an organism is viable can also cause some problems in a test environment. For example, we might determine that an organism does produce an offspring but that this offspring is not identical to itself. In this case, we take the next step of continuing to run the offspring in the test environment, and if necessary its offspring until we find either a self-replicator or a sustainable cycle. By default, we will only test three levels of offspring before we assume the original organism to be nonviable. Such cases happen very rarely, and not at all if implicit mutations are turned on.

Two final problems with the test environments include that they do not properly reflect the levels of limited resources (resource levels can be difficult to estimate, particularly if we are postprocessing) and that they do not handle any special interactions with other organisms since only one is being tested at a time. Both of these issues are currently being examined and we plan to have a much improved test environment in the future. Test environments do, however, work remarkably well in most circumstances.

In addition to reconstructing statistics about organisms as they existed in the population, it is also possible to determine how an organism would have fared in an alternate environment, or even to construct entirely new genomes to determine how they perform. This last approach includes techniques such as performing all single-point mutations on a genome and testing each result to determine what its local fitness landscape looks like, or to artificially crossover pairs of organisms to determine their viability. Test environments are most commonly used in the postprocessing of Avida data, as described in the next section.

1.3 Using Avida

The Avida software currently runs under all three major operating systems: Windows XP, Mac OS X, and Linux. The current version of Avida (including both stable releases and development versions) is available at `http://sourceforge.net/projects/avida/`.

Avida can be run either in an experimental mode, in which a population evolves under the experimental regimme designed by the user, or in an analyze mode in which the user can postprocess their data to a form more useful for them. Both of these modes are explained ahead.

1.3.1 Performing Avida Experiments

Currently there are two main methods of running Avida — either with the graphical user interface (GUI) or in primitive mode (which is faster, but the

user must pre-script the complete experimental protocol). Researchers will often use the GUI to get an intuitive feel of how an experiment works, but then they will shift to the primitive mode when they are ready to perform more exensive data collection.

The configuration of an Avida experiment requires either using the "Configuration Wizard" in the GUI, or manual editing of five different initialization files. The most important of these is the *genesis* file, which contains a list of variables that control all of the basic settings of a run, including the population size, the mutation rates, and the names of all of the other configuration files to use. Next, we have the *instruction set*, which describes the specific genetic language used in the experiment. Third is the *ancestral organism* that the population should be seeded with. Fourth, we have the *environment* file that describes which resources are available to the organisms and defines reactions by the tasks that trigger them, their value, the resource that they use, and any byproducts that they produce. The final configuration file is *events*, which is used to describe specific actions that should occur at designated time points during the experiment, including most data collection and any direct disruptions to the population. Each of these files is described in more detail in the Avida documentation.

Once Avida has been properly installed, and the configuration files set up, it can be started in primitve mode by going into the work/ directory and typing `primitive` on the command line (or else by clicking on the corresponding icon). Some basic information will scroll by on the screen (specifically, current update being processed, number of generations, average fitness, and current population size). When the experiment has completed, the process will terminate automatically, leaving a set of output files that describe the completed experiment. Each output file begins with an index describing the contents of that file.

Running the graphical version of Avida differs by platform but is well described in the documentation that is contained within the appropriate version. When Avida is started, it will give the option to use pre-existing configuration files (which can be set up in the same way as for the primitive mode), or else by running the configuration wizard, which will take the user step-by-step through all of the choices necessary to specifying an experimental protocol. The wizard can operate in two different modes, a basic mode, where only a few simple questions need to be answered, or a more advanced mode that provides access to all settings of Avida.

The first window that opens once Avida has started displays a view of the whole population, as shown in Fig. 1.5. This screen provides a pull-down menu that allows the user to choose what information should be displayed about each organism, such as its genotype, its fitness, or its age. Several other Avida viewers can also be launched from this screen. These viewers include graphs of data being collected, an instruction viewer to demonstrate how individual organisms function (as shown in Fig. 1.6), editors to control events

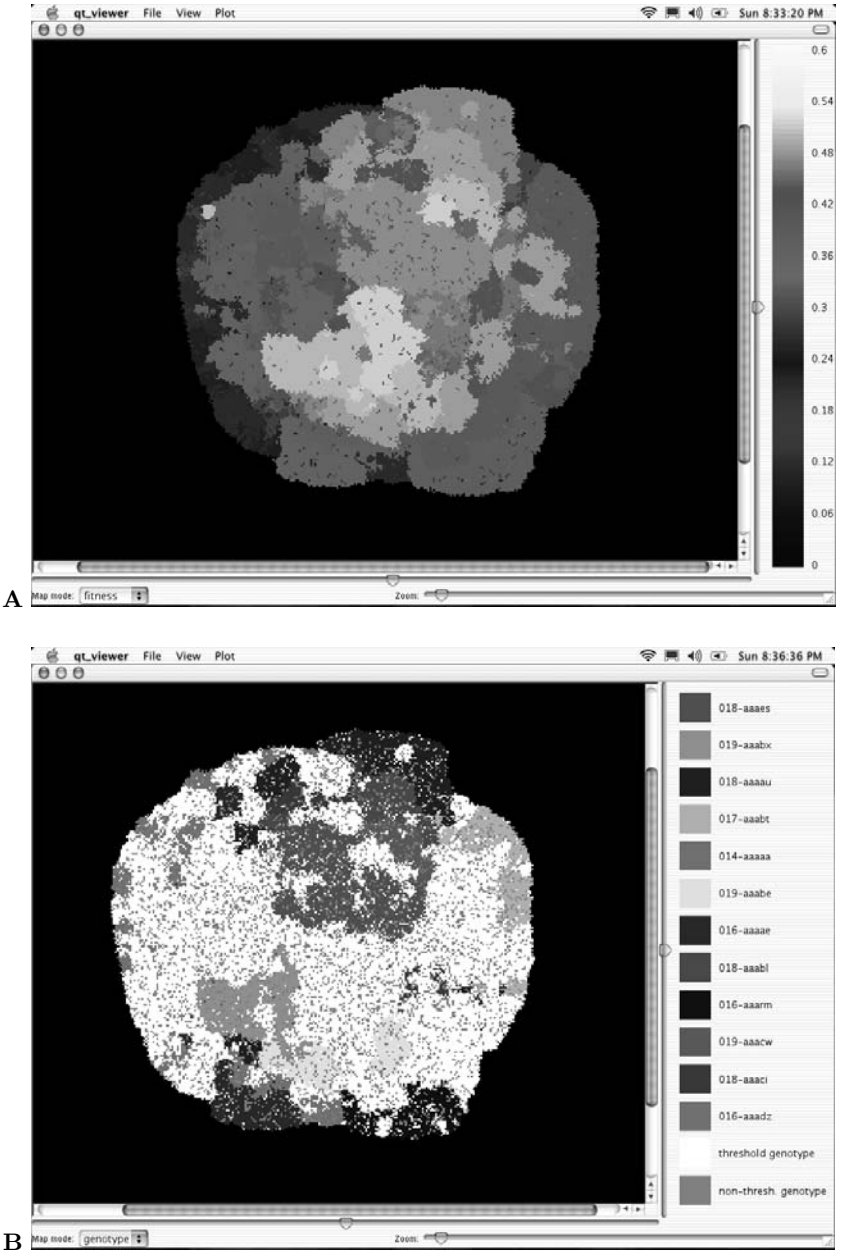

Fig. 1.5. The grid viewer from a typical Avida experiment. (A) Fitness map. (B) Genotype map.

or the environment, or even additonal map viewers (if multiple aspects of the population should be displayed at the same time).

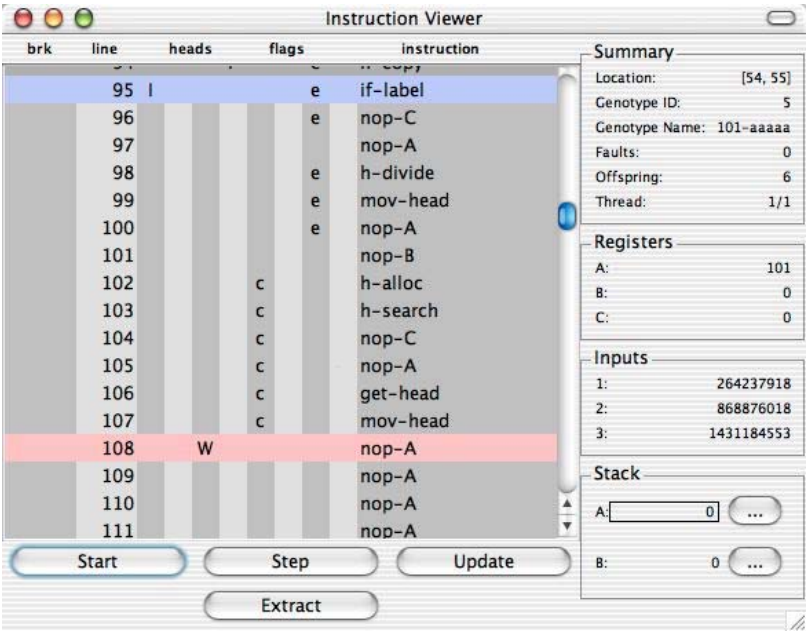

Fig. 1.6. A viewer window that allows the user to focus on a single Avida organism and monitor it as it executes its genome.

The graphical interface to Avida is currently under heavy development, with many new visualization tools expected to be introduced in the near future, as well as an extensive help and tutorial system, and an easy interface to the analysis tools described in the next section.

1.3.2 Analyze Mode

Avida has an analysis-only mode (short *analyze mode*), which allows for powerful postprocessing of data. Avida is brought into the analyze mode by the command-line parameter "-a". In the analyze model, Avida processes the analyze file specified in the genesis file ("analyze.cfg" by default). The analyze file contains a program written in a simple scripting language. The structure of the program involves loading in genotypes in one or more *batches*, and then either manipulating single batches, or doing comparisons between batches.

In the following paragraphs, we present a couple of example programs that will illustrate the basics of the analyze scripting language. A full list of commands available in analysis mode is given in the Avida documentation.

Testing a Genome Sequence

The following program will load in a genome sequence, run it in a test environment, and output the results of the tests in a couple of formats.

```
VERBOSE
LOAD_SEQUENCE rmzavcgmciqqptqpqcpctletncogcbeamqdtqcptipqfpg
RECALCULATE
DETAIL detail_test.dat fitness merit length viable sequence
TRACE
PRINT
```

The program starts off with the VERBOSE command, which causes Avida to print to screen all details of what is going on during the execution of the analyze script; the command is useful for debugging purposes. The program then uses the LOAD_SEQUENCE command to define a specific genome sequence in compressed format. (The compressed format is used by Avida in a number of output files. The mapping from instructions to letters is determined by the instruction set file and may change if the instruction set file is altered.)

The RECALCULATE command places the genome sequence into the test environment and determines the organism's fitness, merit, gestation time, and so on. The DETAIL command that follows prints this information into the file "detail_test.dat". (This filename is specified as the first argument of DETAIL.) The TRACE and PRINT commands will then print individual files with data on this genome, the first tracing the genome's execution line by line, and the second summarizing several test results and printing the genome line by line. Since no directory was specified for these commands, "genebank/" is assumed, and the filenames are "org-S1.trace" and "org-S1.gen". If a genotype has a name when it is loaded, then that name will be kept. Otherwise, it will be assigned a name starting with "org-S1", then "org-S2", and so on. The TRACE and PRINT commands add their own suffixes (".trace" and ".gen") to the genome's name to determine the filenames they will use.

Finding Lineages

The portion of an Avida run that we will often be most interested in is the lineage from a genotype (typically the final dominant genotype) back to the original ancestor. There are tools in the analyze mode to obtain this information, if the necessary population and ancestral dumps have been written out with the events `detail_pop` and `dump_historic_pop`. The following program demonstrates how to make use of these dump files.

```
FORRANGE i 100 199
  SET d /home/charles/dev/Avida/runs/evo-neut/evo_neut_$i
  PURGE_BATCH
  LOAD_DETAIL_DUMP $d/detail_pop.100000
```

```
  LOAD_DETAIL_DUMP $d/historic_dump.100000
  FIND_LINEAGE num_cpus
  RECALCULATE
  DETAIL lineage.$i.html depth parent_dist html.sequence
END
```

The FORRANGE command runs the contents of the loop once for each possible value in the range, setting the variable i to each of these values in turn. Thus the first time through the loop, 'i' will be equal to the value 100, then 101, 102, and so on, all the way up to 199. In this particular case, we have 100 runs (numbered 100 through 199) we want to work with.

The first thing we do once we are inside the loop is to set the value of the variable 'd' to be the name of the directory we are going to be working with. Since this directory name is long, we do not want to have to type it every time we need it. If we set it to the variable 'd', then all we need to do is to type "$d" in the future. Note that in this case we are setting a variable to a string instead of a number; that is fine, and Avida will figure out how to handle the contents of the variable properly. The directory we are working with changes each time the loop is executed, since the variable 'i' is part of the directory name.

We then use the command PURGE_BATCH to get rid of all genotypes from the last execution of the loop (lest we are not accumulating more and more genotypes in the current batch), and refill the batch by using LOAD_DETAIL_DUMP to load in all genotypes saved in the file "detail_pop.100000" within our chosen directory. A detail file contains all of the genotypes that were currently alive in the population at the time the detail file was printed, while a historic file (the next one loaded) contains all of the genotypes that are ancestors of those that are still alive. The combination of these two files gives us the lineages of the entire population back to the original ancestor. Since we are only interested in a single lineage, we next run the FIND_LINEAGE command to pick out a single genotype, and discard everything else except for its lineage. In this case, we pick the genotype with the highest abundance (i.e., the highest number of organisms, or virtual CPUs, associated with it) at the time of output.

As before, the RECALCULATE command gets us any additional information we may need about the genotypes, and then we print that information to a file using the DETAIL command. The filenames that we are using this time have the format "lineage.$i.html", that is, they are all being written to the current directory, with filenames that incorporate the run number. Also, because the filename ends in the suffix ".html", Avida prints the file in html format, rather than in plain text. Note that the specific values that we choose to print take advantage of the fact that we have a lineage (and hence have measured things like the genetic distance to the parent) and are in html mode (and thus can print the sequence using colors to specify where exactly mutations occurred).

These examples are only meant to present the reader with an idea of the types of analyses available in this built-in scripting language. Many more are possible, but a more exhaustive discussion of these possibilities is beyond the scope of this chapter.

1.4 A Summary of Avida Research

Avida has been used in several dozen peer-reviewed scientific publications, including some in *Nature* [20,21,43] and *Science* [4]. We describe a few of our more interesting efforts ahead.

1.4.1 The Evolution of Complex Features

When Darwin first proposed his theory of evolution by natural selection, he realized that it had a problem explaining the origins of the vertebrate eye [7]. Darwin noted that "In considering transitions of organs, it is so important to bear in mind the probability of conversion from one function to another." That is, populations do not evolve complex new features *de novo*, but instead modify existing, less complex features for use as building blocks of the new feature. Darwin further hypothesized that "Different kinds of modification would [...] serve for the same general purpose," noting that just because any one particular complex solution may be unlikely, there may be many other possible solutions, and we only witness the single one lying on the path evolution took. As long as the aggregate probability of all solutions is high enough, the individual probabilities of the possible solutions are almost irrelevant.

Substantial evidence now exists that supports Darwin's general model for the evolution of complexity (e.g., [8, 16, 25, 26, 44]), but it is still difficult to provide a complete account of the origin of any complex feature due to the extinction of the intermediate forms, imperfection of the fossil record, and incomplete knowledge of the genetic and developmental mechanisms that produce such features. Digital evolution allowed us to surmount these difficulties and track all genotypic and phenotypic changes during the evolution of a complex trait with enough replication to obtain statistically powerful results [21]. We isolated the computation EQU (logical equals) as a complex trait, and showed that at least 19 coordinated instructions are needed to perform this task. We then performed an experiment that consisted of 100 independent populations of digital organisms being evolved for approximately 17,000 generations. We evolved 50 of these populations in a control environment where EQU was the only task rewarded; we evolved the other 50 in a more complex environment where an assortment of 8 simpler tasks were rewarded as well, to test the importance of intermediates in the evolution of a complex feature.

Results: In 23 of the 50 experiments in the complex environment, the EQU task was evolved, whereas *none* of the 50 control populations evolved

EQU, illustrating the critical importance of features of intermediate complexity ($P \approx 4.3 \times 10^{-9}$, Fisher's exact test). Furthermore, all 23 implementations of the complex trait were unique, with many quite distinct from each other in their approach, indicating that, indeed, this trait had numerous solutions. This observation is not surprising, since even the shortest of the implementations found were extraordinarily unlikely (approximately 1 in 10^{27}). We further analyzed these results by tracing back the line of descent for each population to find the critical mutation that first produced the complex trait. In each case, these random mutations transformed a genotype unable to perform EQU into one that could, and even though these mutations typically affected only 1 to 2 positions in the genome, a median of 28 instructions were required to perform this complex task — a change in any of these instruction would cause the task to be lost, thus it was complex from the moment of its creation. It is noteworthy to mention that in 20 of the 23 cases the critical mutations would have been detrimental if EQU were not rewarded, and in three cases the prior mutation was actively detrimental (causing the replication rate for the organisms to drop by as much as half), yet turned out to be critical for the evolution of EQU; when we reverted these seemingly detrimental mutations, EQU was lost.

1.4.2 Survival of the Flattest

When organisms have to evolve under high mutation pressure, their evolutionary dynamics is substantially different from that of organisms evolving under low mutation pressure, and some of the high-mutation-rate effects can appear paradoxical at first glance. Most of population genetics theory has been developed under the assumption that mutation rates are fairly low, which is justified for the majority of DNA-based organisms. However, RNA viruses, the large class of viruses that cause diseases such as the common cold, influenza, HIV, SARS, or Ebola, tend to suffer high mutation rates, up to 10^{-4} substitutions per nucleotide and generation [12]. The theory describing the evolutionary dynamics at high mutation rates is called quasispecies theory [11].

The main prediction for the evolutionary process at high mutation rates is that selection acts on a cloud of mutants, rather than on individual sequences. We tested this hypothesis in Avida [43]. First, we let strains of digital organisms evolve to both a high-mutation-rate and a low-mutation-rate environment. The rationale behind this initial adaptation was that strains that evolved at a low mutation rate should adapt to ordinary individual-based selection, whereas strains that evolved at a high mutation rate should adapt to selection on mutant clouds, which means that these organisms should maximize the overall replication rate of their mutant clouds, rather than their individual replication rates. This adaptation to maximized overall replication rate under high mutation pressure takes place when organisms trade individual fitness for mutational robustness, so that their individual replication rate is reduced but in return the probability that mutations cause further

reduction in the replication rate is also reduced [42]. Specifically, we took 40 strains of already evolved digital organisms, and let each evolve for an additional 1000 generations in both a low-mutation-rate and a high-mutation-rate environment. As result, we ended up with 40 pairs of strains. The two strains of each pair were genetically and phenotypically similar, apart from the fact that one was adapted to a low and one to a high mutation rate. As expected, we found that in the majority of cases the strains evolved at a high mutation rate had a lower replication speed than the ones evolved at a low mutation rate.

Next, we let the two types of strains compete with each other, in a setup where both strains would suffer from the same mutation rate, which was either low, intermediate, or high. Not surprisingly, at a low mutation rate the strains adapted to that mutation rate consistently outcompeted the ones adapted to a high mutation rate, since after all the former ones had the higher replication rate (we excluded those cases in which the strain evolved at a low mutation rate had a lower or almost equal fitness to the strain evolved at a high mutation rate). However, without fail, the strain adapted to a high mutation rate could win the competition if the mutation rate during the competition was sufficiently high [43]. This result may sound surprising at first, but it has a very simple explanation. At a high mutation rate (1 mutation per genome per generation or higher), the majority of an organism's offspring differ genetically from their parent. Therefore, if the parent is genetically very brittle, so that most of these mutants have a low replication rate or are even lethal, then the overall replication rate of all the organism's offspring will be fairly moderate, even though the organism itself may produce offspring at a rapid pace. If a different organism produces offspring at a slower pace, but is more robust towards mutations, so that the majority of this organism's offspring have a replication rate similar to that of their parent, then the overall replication rate of this organism's offspring will be larger than the one of the first organism. Hence, this organism will win the competition, even though it is the slower replicator. We termed this effect the "survival of the flattest," because at a sufficiently high mutation rate a strain that is located on a low but flat fitness peak can outcompete one that is located on a high but steep fitness peak.

1.4.3 Evolution of Digital Ecosystems

The experiments discussed above both used single-niche Avida populations, but evolutionary design is more interesting (and more powerful) when we consider ecosystems. The selective pressures that cause the formation and diversity of ecosystems are still poorly understood [36, 38]. In part, the lack of progress is due to the difficulty of performing precise, replicated, and controlled experiments on whole ecosystems [24]. To study simple ecosystems in a laboratory microcosm (reviewed in [39]), biologists often use a chemostat, which slowly pumps resource rich media into a flask containing bacteria, while simultaneously draining the flask's contents to keep the volume constant. Un-

fortunately, even in these simple model systems, ecosystems can evolve to be more complex than is experimentally tractable, and understanding their formation remains difficult [27, 28, 32].

We set up Avida experiments based on this chemostat model [6] wherein 9 resources flow into the population, and 1% of unused resources flow out. We used populations with 2500 organisms, each of which absorbed a small portion of an available resource whenever they performed the corresponding task. If too many organisms focus on the same resource, it will no longer be plentiful enough to encourage additional use.

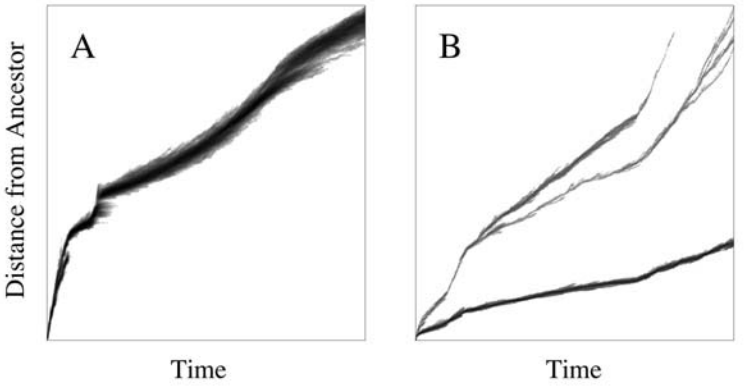

Fig. 1.7. Visualizations of phylogenies from the evolution of (A) a single niche population, and (B) a population with limited resources (and hence multiple niches). The x-axis represents time, while the y-axis is depth in the phylogeny (distance from the original ancestor). Intensity at each position indicates the number of organisms alive at the corresponding point in time and depth in the tree.

Theory predicts that an environment with either a single resource or with resources in unlimited quantities is capable of supporting only one species [37], and this is exactly what we see in the standard Avida experiments. It is the competition over multiple, limited resources that is believed to play a key role in the structuring of communities [35, 39]. In 30 trials under the chemostat regime in Avida, a variety of distinct community structures developed [6]. Some evolved nine stably coexisting specialists, one per resource, while others had just a couple of generalists that divided the resources between them. Others still mixed both generalists and specialists. In all cases, the ecosys-

tems proved to be stable because they persisted after all mutations were shut off in the system, and if any abundant phenotype were removed, it would consistently reinvade.

Phylogeny visualizations provide a striking demonstration of the differences between populations that evolved in a single niche and those from ecosystems, as displayed in Fig. 1.7. Single niche populations can have branching events that persist for a short time, but in the long term one species will out compete the others, or simply drift to dominance if the fitness values are truly identical. By contrast, in ecosystems with multiple resources, the branches that correspond to speciation events persist.

We also studied the number of stably coexisting species as a function of resource availability [4]. We varied the inflow rate of resources over six orders of magnitude and found that multispecies communities evolved at intermediate resource abundance, but not at very high or very low resource abundance. The reason for this observation is that if resources are too scarce, they cannot provide much value to the organisms and base merit dominates, while if resources are too abundant, then they are no longer a limiting factor, which means that space becomes the only limit. In both cases the system reduces down to only a single niche that the organisms can take advantage of.

1.5 Outlook

Digital organisms are a powerful research tool that has opened up methods to experimentally study evolution in ways that have never before been possible. We have explained the capabilities of the Avida system and detailed the methods by which researchers can make use of them. We must be careful, however, not to be lured into the trap of thinking that because these systems can be set up and examined so easily that any experiment will be possible. There are definite limits on the questions that can be answered.

Using digital organisms, we cannot learn anything about physical structures evolved in the natural world, nor the specifics of an evolutionary event in our own history; the questions we ask must be about *how* evolution works in general, and how we can harness it. Even for the latter type of questions, it is sometimes difficult to set up experiments in such a way that they give meaningful results. We must always remember that we are working with an arguably living system that will evolve to survive as best it can, not always in the direction that we intend. Avida has become, in many ways, its own bug tester. If we make a mistake, the organisms will exploit it. For example, we originally had only 16-bit inputs for the organisms to process; they quickly discovered that random guessing often took less time than actually performing the computation. In this case, the organisms indeed found the most efficient way to solve the problem we gave them, only that it wasn't the problem we had thought we were giving. This error happened to be easy to find and easy to fix — now all inputs are 32 bits long — but not all "cheating" will be so

simple to identify and prevent. When performing an Avida experiment, it is always important that we step through the population and try to understand how some of the organisms are functioning. More often than not they will surprise us with the cleverness of the survival strategies that they are using. And sometimes they will even make us step back to rethink our experiments.

Many possible future directions exist in the development of Avida. Ongoing efforts include (among others) the implementation of a new CPU model that is more powerful and realistic, an overhaul of the graphical user interface that will include more visualization tools and will be designed so that it can easily be used by those not familiar with the software, an expanded analyze mode based on the scripting language Python, and the move from asexual to sexual organisms. We hope for these additions to expand the user base of the software as well as the range of experiments possible.

Finally, we have an Avida Educational Initiative underway that is focusing on modifying the software so that it will be more conducive for use in a classroom. Our initial goal is for it to be used in introductory college biology courses to help elucidate simple evolutionary concepts. Eventually we plan to both simplify it further for a high school setting and to create a bridge to the research version of Avida so that it will be useful in more specialized biology courses as well.

References

1. Adami C, Brown CT, Haggerty MR (1995) Abundance distributions in artificial life and stochastic models: "Age and Area" revisited. Lect Notes AI 929: 503–514.
2. Adami C, Ofria C, Collier TC (2000) Evolution of biological complexity. Proc. Natl. Acad. Sci. U.S.A. 97:4463–4468.
3. Barton N, Zuidema W (2003) Evolution: The erratic path towards complexity. Curr Biol 13:R649–R651.
4. Chow SS, Wilke CO, Ofria C, Lenski RE, Adami C (2004) Adaptive radiation from resource competition in digital organisms. Science 305:84–86.
5. Chu J, Adami C (1997) Propagation of information in populations of self-replicating code. In: Langton CG, Shimohara T (eds.) Proc. Artificial Life V, pp. 462–469, MIT Press.
6. Cooper T, Ofria C (2002) Evolution of stable ecosystems in populations of digital organisms. In: Standish RK, Bedau MA, Abbass HA (eds.), Proc. Artificial Life VIII, pp. 227–232, MIT Press.
7. Darwin C (1859) On the Origin of Species by Means of Natural Selection. Murray.
8. Dawkins R (1986) The Blind Watchmaker. Norton.
9. Dennett D (2002) The New Replicators. In: Pagel M (ed.) Encyclopedia of Evolution, Oxford Univ. Press.
10. Dewdney AK (1984) In a game called core war hostile programs engage in a battle of bits. Scientific American, May issue, pp. 14–22.

11. Domingo E, Biebricher CK, Eigen M, Holland JJ (2001) Quasispecies and RNA Virus Evolution: Principles and Consequences. Landes Bioscience, Georgetown, TX, USA.
12. Drake JW, Holland JJ (1999) Mutation rates among RNA viruses. Proc Natl Acad Sci 96:13910–13913.
13. Egri-Nagy A, Nehaniv CL (2003) Evolvability of the genotype-phenotype relation in populations of self-replicating digital organisms in a Tierra-like system. Lect Notes Artif Int 2801:238–247.
14. Elena SF, Lenski RE (2003) Evolution experiments with microorganisms: The dynamics and genetic bases of adaptation. Nature Reviews Genetics 4:457–469.
15. Goldberg DE (2002) The Design of Innovation. Kluwer, Dordecht, Netherlands.
16. Jacob F (1977) Evolution and Tinkering. Science, 196:1161–1166.
17. Kim Y, Stephan W (2003) Selective sweeps in the presence of interference among partially linked loci. Genetics 164:389–398.
18. Koza J (ed.) (2003) Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Kluwer, Dordecht, Netherlands.
19. Lenski RE (2004) Phenotypic and genomic evolution during a 20,000-generation experiment with the bacterium, *Escherichia coli*. Plant Breeding Reviews 24:225–265.
20. Lenski RE, Ofria C, Collier TC Adami C (1999) Genome complexity, robustness and genetic interactions in digital organisms. Nature 400:661–664.
21. Lenski RE, Ofria C, Pennock RT, Adami C (2003) The evolutionary origin of complex features. Nature 423:129–144.
22. Maynard Smith J (1992) Byte-sized evolution. Nature 355:772–773.
23. McVean GAT, Charlesworth B (2000) The effects of Hill-Robertson interference between weakly selected mutations on patterns of molecular evolution and variation. Genetics 155:929–944.
24. Morin PJ (2002) Biodiversity's ups and downs, Nature 406:463–464.
25. Newcomb RD, Campbell PM, Ollis DL, Cheah E, Russell RJ, Oakeshott JG (1997) A single amino acid substitution converts a carboxylesterase to an organophosphorus hydrolase and confers insecticide resistance on a blowfly. Proc Natl Acad Sci 94:7464–7468.
26. Nilsson D-E and Pelger SA (1994) A pessimistic estimate of the time required for an eye to evolve. Proc R Soc Lond B 256:53–58.
27. Notley-McRobb L, Ferenci T (1999) Adaptive mgl-regulatory mutations and genetic diversity evolving in glucose limited *Escherichia coli* populations. Env Microbiol 1:33–43.
28. Notley-McRobb L, Ferenci T (1999) The generation of multiple co-existing mal-regulatory mutations through polygenic evolution in glucose-limited populations of *Escherichia coli*. Env Microbiol 1:45–52.
29. Ofria C, Wilke CO (2004) Avida: A software platform for research in computational evolutionary biology. Artificial Life 10:191–229.
30. O'Neill B (2003) Digital evolution. PLoS Biology 1:011–014.
31. Orr HA (2000) The rate of adaptation in asexuals. Genetics 155:961–968.
32. Rainey PB, Travisano M (1998) Adaptive radiation in an heterogeneous environment. Nature 394:69–72.
33. Rasmussen S, Knudsen C, Feldberg R, Hindsholm M (1990) The coreworld — Emergence and evolution of cooperative structures in a computational chemistry. Physica D 75:1–3.

34. Ray TS (1992) An approach to the synthesis of life. In: Langton CG, Taylor C, Farmer JD, Rasmussen S (eds.). Proc. of Artificial Life II, p. 371. Addison-Wesley.
35. Schluter D (1996) Ecological causes of adaptive radiation, Am Nat 148:s40–s64.
36. Schluter D (2001) Ecology and the origin of species, Trends Ecol Evol 16:372–379.
37. Tilman D (1982) Resource Competition and Community Structure, Princeton University Press.
38. Tilman D (2000) Causes, consequences and ethics of biodiversity, Nature 405:208–211.
39. Travisano M, Rainey PB (2000) Studies of adaptive radiation using model microbial systems. Am Nat 156:S35–S44.
40. Wilke CO (2002) Maternal effects in molecular evolution. Phys Rev Lett 88:078–101.
41. Wilke C, Adami C (2002) The biology of digital organisms. Trends Ecol Evol 17:528–532.
42. Wilke CO, Adami C (2003) Evolution of mutational robustness. Mutat Res 522:3–1.
43. Wilke CO, Wang JL, Ofria C, Lenski RE, Adami C (2001) Evolution of digital organisms at high mutation rates leads to survival of the flattest. Nature 412:331–333.
44. Wilkins AS (2002) The Evolution of Developmental Pathways, Sinauer.
45. Yedid G, Bell G (2001) Microevolution in an electronic microcosm. Am Nat 157:465–487.
46. Yedid G, Bell G (2002) Macroevolution simulated with autonomously replicating computer programs. Nature 420:810–812.