Digital evolution is a form of evolutionary computation closely associated with artificial life in which a population of self-replicating computer programs that are subject to mutations and natural selection exists in a user-defined computational environment. Each organism comprises an evolved genome (program) of assembly-like instructions and a virtual CPU on which the genome executes. Since each genome is produced by a sequence of random mutations over evolutionary time, the encoding of behavior within the genome is often obscure, making manual analysis tedious and time-consuming. In this thesis, we present a methodology for analyzing digital organisms using program understanding and Bioinformatics techniques, including program slicing and sequence alignment, both of which we have adapted for the digital evolution context. We apply this approach to a set of evolved genomes and demonstrate how dependency analysis and slicing facilitate identification of known patterns and operations embedded within the genomes.
IDENTIFICATION OF CANDIDATE CONCEPTS IN A LEARNING-BASED APPROACH TO REVERSE ENGINEERING

A Thesis

Submitted to the
Faculty of Miami University
in partial fulfillment of
the requirements for the degree of
Master of Computer Science
Department of Computer Science and Software Engineering

by

Hanqing Hu
Miami University
Oxford, Ohio
2010

Advisor

Dr. Gerald Gannod

Committee Member

Dr. James Kiper

Committee Member

Dr. John Karro
Contents

1 Introduction 1
   1.1 Contributions ............................................. 4
   1.2 Organization ............................................. 4

2 Introducing Avida and Related Background 5
   2.1 Avida Core .................................................. 5
   2.2 Analysis and Statistics ................................... 13
   2.3 Program Slicing ........................................... 14
      2.3.1 Static Slicing ........................................ 14
      2.3.2 Dynamic Slicing ...................................... 15
   2.4 Alignment .................................................. 16
      2.4.1 Global Alignment .................................... 17
      2.4.2 Local Alignment ...................................... 17

3 Approach 18
   3.1 Flow Graphs ............................................... 20
      3.1.1 Sequential Execution .................................. 20
      3.1.2 Conditional Instructions .............................. 21
      3.1.3 Jump Instructions ..................................... 22
   3.2 Dead Code Graph ........................................... 23
   3.3 Data Dependency Graphs ................................... 24
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Results for Avida background</td>
<td>51</td>
</tr>
<tr>
<td>5.2</td>
<td>Results for program slicing questions</td>
<td>51</td>
</tr>
<tr>
<td>5.3</td>
<td>Result for evolution context analysis questions</td>
<td>52</td>
</tr>
<tr>
<td>5.4</td>
<td>Result for Bioinformatics and Computer Science background</td>
<td>53</td>
</tr>
<tr>
<td>5.5</td>
<td>Task result correctness percentage</td>
<td>53</td>
</tr>
<tr>
<td>5.6</td>
<td>Opinion for slicing software</td>
<td>54</td>
</tr>
<tr>
<td>5.7</td>
<td>Opinion for global alignment software</td>
<td>55</td>
</tr>
<tr>
<td>5.8</td>
<td>Opinion for local alignment software</td>
<td>55</td>
</tr>
</tbody>
</table>
List of Figures

2.1 A circular gene from Avida Organism ........................................... 6
2.2 Virtual hardware: genome, CPU, registers, stack, heads and IO .......... 7
2.3 The head of a template nop-A nop-B ............................................. 10
2.4 Two overlapping template ............................................................ 10
2.5 Static slice for a simple program .................................................. 15
2.6 Dynamic slice for a simple program .............................................. 16
2.7 An example of a alignment .......................................................... 16

3.1 Flow chart of proposed methodology .......................................... 20
3.2 Representation of sequential execution ........................................ 21
3.3 Conditional Instructions .............................................................. 21
3.4 Examples of Jump and Dead Code ............................................... 23
3.5 Example of a data dependency graph ............................................ 25
3.6 DDG and slices of output IO instruction from a simple Avida genome 27

4.1 Expanded form of the example genome ....................................... 33
4.2 Avida-ED visualizations .............................................................. 34
4.3 Genome visualizations ............................................................... 35
4.4 Static slice qgfcldqppq .............................................................. 36
4.5 Dynamic slice qgfcqppq .............................................................. 36
4.6 The data dependency graph of the ORN static slice ......................... 37
Alignment result for two short DNA sequence with Match score 2,
mismatch score -1 and gap score -1
Acknowledgements

I would like to thank my advisor, Dr. Gerald Gannod, for making this thesis possible through his guidance and dedication. Even before I was admitted to the program, he gave me advice and encouragement. I am grateful for his expertise and direction throughout this thesis.

I would also like to thank Dr. James Kiper and Dr. John Karro for serving on my committee. I appreciate the time and effort they put into finding areas of improvement for my thesis.

Thank you to the faculty of the Computer Science and Software Engineering department for their many contributions to my education and research. I can not imagine a better graduate school experience. Moreover, I am thankful for the funding during these two years.

Finally, thank you to the support from my family who support my work and life throughout my college years. Thank you to all my friend for helping me through harsh times.
Chapter 1

Introduction

Evolutionary computation (EC) methods codify the basic principles of genetic evolution in computer software. Traditionally, EC has been particularly effective in addressing problems with large, multidimensional search spaces whose fitness landscapes contain peaks and valleys [1]. The most well-known method is the genetic algorithm (GA) [2], an iterative search technique in which the individuals in the population are encodings of candidate solutions to an optimization problem. In each generation, the fitness of every individual is calculated, and a subset of individuals is selected, then recombined and/or mutated to form the next generation. GAs and related methods have successfully been used to solve complex problems, rivaling and even surpassing human designers in many application domains, including automotive and medical [3,4].

Digital evolution [5] is a form of evolutionary computation developed primarily as a tool for evolutionary biology. Instead of harnessing the evolutionary process strictly for search, digital evolution is intended to realize evolution in software in order to provide insight into the evolution of natural systems. In Avida [5,6], the most widely used digital evolution platform, “digital organisms” replicate asynchronously and interact with one another and their environment. Each organism comprises a circular genome of assembly-like instructions and a virtual CPU on which this program
executes. Organisms compete for space in a virtual environment and evolve over generations through random mutations that occur during replication. In Avida, whether a given organism’s genes survive to future generations depends on the organism’s ability to replicate faster and to live longer than other organisms, a simplification of nature’s process.

The Avida platform is not intended to simulate the physical and chemical processes found in natural organisms. Rather, conducting experiments in the “Avida world” enables the researcher to distill complex interactions into an abstract form that permits direct observation and analysis of the evolutionary process. Avida has been used to investigate several fundamental aspects of the evolutionary process, including the evolution of biocomplexity [7], kin selection [8], specialization and phenotypic plasticity [9,10], and genetic organization [11]. More recently, Avida has been applied to the development of distributed computing systems. Effectively, this platform provides the systems developer with a “cultivation laboratory” for constructing algorithms and protocols that are robust to highly dynamic and adverse environmental conditions [12–14].

Whether Avida is applied to biology or engineering, an understanding of the evolved behavior of genomes is essential. In the case of biology, this knowledge can give insight into how evolution constructs and combines simple “building blocks” to produce more complex behaviors [7]. In the case of engineering, Avida has the capacity to produce novel algorithms. For example, in a recent study an Avida population evolved a probabilistic consensus algorithm that is robust to message loss [14]. A detailed understanding of such an algorithm is needed to translate it into other programming languages and assure its correctness, for example, through model checking.

However, since each Avida genome is produced by a sequence of random mutations over evolutionary time, the encoding of behavior within the genome is often obscure, with code for different functions interwoven and mixed with “junk” code left
over from earlier generations, making manual analysis tedious and time-consuming. Moreover, many Avida instructions are context sensitive, with the behavior of code sequences dependent on surrounding code, the content of registers in the virtual CPU, and the current execution environment. Currently, Avida users set up waypoints in their experiments to observe the behavior of populations at a macro-level. However, this approach offers little explanation for how individual behaviors arose, or how the evolutionary process might have co-opted earlier functionality for more complex functions. There has been little work in the application of rigorous program analysis methods to help understand digital organisms and the populations they form.

In this thesis we present a methodology for understanding Avida organisms and population behaviors using program analysis tools [15] and sequence alignment algorithms [16, 17]. This methodology provides a means to quickly and easily interpret the semantics of Avida organisms and uncover hidden dependencies among program instructions in the genome. It also creates visualizations for putting a single organism in the perspective of a whole evolution process. Additional understanding comes from analyzing changes and dependencies within and among genomes over multiple generations. Then, it is compared to numerous other organisms in the population to understand how they evolve. Among numerous tools of program analysis, our research focuses primarily on program slicing [15]. In the context of digital organisms, program slicing identifies parts of the genome based on slicing criteria specified in terms of genome outputs. These parts correspond to specific functional regions that together implement the evolved traits of the organism. The sequence alignment algorithms used are standard Needleman-Wunsch and Smith-Waterman algorithm [16, 17].
1.1 Contributions

The contributions of this research facilitate the creation of a methodology for understanding Avida organisms and population behaviors using program analysis tools [4] and sequence alignment techniques [5]. Our method provides a way for quickly and easily interpreting the semantics of Avida organisms and subsequently explaining the reason behind their evolution traits. The research also provides a way of validating Avida so that researchers can claim that the platform does indeed model real-world evolutionary processes.

**Thesis Statement**  
Digital evolution platforms such as Avida produce organisms that are difficult to interpret. Through the use of program understanding and sequence alignment techniques, these organisms can be more easily analyzed and understood.

1.2 Organization

The remainder of this thesis is organized as follows. Chapter 2 describes the Avida platform, including features of the Avida instruction set that complicate genome analysis. Chapter 3 presents our approach to analyzing Avida organisms using program slicing and Bioinformatics sequence alignment. Chapter 4 applies the method to a non-trivial example. Chapter 5 describes how our method was evaluated using a pre-test post-test user study and the corresponding results.
Chapter 2

Introducing Avida and Related Background

Avida is a software platform for research in computational evolutionary biology [5], enabling researchers to perform experiments on self-replication and evolution[1]. Several protocols and configuration files are used to set up the experiment environment and exert control over the process. The Avida platform achieves the task with three main modules: Avida Core, Avida Analysis and Statistics and Graphical User Interface(GUI)[1].

2.1 Avida Core

The Avida Core is mainly composed of the Avida organism and the virtual hardware upon which Avida organisms run.

1. Avida Organism

An Avida organism is an individual, self-replicating computer program. An organism has its own genome, which is a program made of a sequence of assembly language-like instructions. These instructions are constructed into a circle in the virtual memory, giving the organism a circular gene (Figure 2.1). The Avida
organism is capable of self-replicating: building its offspring’s genome and then giving the genome to the Avida world. The world will then determine where to place the genome in the population [1]. Each organism takes up a cell space in the petri dish. The cells are numbered from 0 to the configurable maximum size of the population, usually from 3600 up to 10000, and are arranged in a 2-dimensional grid. The offspring will be placed in the adjacent cell of the parent organism. If all four of the adjacent cells of the parent organism are occupied by other organisms, then Avida will pick one to be replaced. The experimenter can either choose to replace the oldest organism or replace at random by changing the settings in the configuration file.

![Figure 2.1: A circular gene from Avida Organism](image)

2. Instruction Set

All the instructions that comprise the Avida genome are defined in an instruction set. A default instruction set is first defined to support the most primitive functionality of Avida, that is, run the organism and let them replicate and mutate. There are other instruction sets that serve specific purposes for the interest of different research groups using the Avida platform. For instance, there is a parasite instruction set for studying parasites. The alternative instruction sets are expansions or modifications of the default instruction set. This research concentrates on the default instruction set because once we understand the semantics of the default, the same principle can be applied easily to its expansions
and modifications.

3. Virtual Hardware

The virtual hardware Avida provides for its organisms is similar to the Von Neumann computer architecture. Figure 2.2 (replicated from [5]) demonstrates the standard virtual hardware structure. The CPU is capable of reading and executing instructions, performing IO and performing mathematical operations. Three registers $AX$, $BX$ and $CX$ are used to store the computation result or other information such as the input value after an IO call. Two stacks are also available to store program information from the CPU, but only one stack is active. The CPU is capable of switching between the two stacks, so it can access both of them.

![Virtual Hardware Diagram](image)

Figure 2.2: Virtual hardware: genome, CPU, registers, stack, heads and IO

The program memory is initialized with an organism’s genome. The hardware will then read the memory and start executing the genome at a well-defined
starting-point. Then it executes the instructions sequentially unless a jump instruction interrupts the order of execution. The memory is organized in a circular way, such that the program will be back to the beginning again after it reaches the end of the genome. The program continues to execute until the organism is dead or replaced by another organism.

Four heads, instruction head, read head, write head, and flow head control the program’s execution flow. Technically, the heads are memory pointers that dictate how the CPU will handle the genome. The instruction head functions as a program counter as in the Von Neumann computer architecture. It points to the instruction currently being executed, and will move to the next instruction once the current instruction is done, either sequentially or through jumping. The other three heads do not exist in the Von Neumann computer architecture and are unique to Avida. The read head and write head are used in self-replication. In order to make a copy of the genome, the CPU needs to read one instruction and write a copy of that instruction into new memory. The read head points to the instruction that is currently being copied and the write head specifies where the instruction will be copied to (with some possibility of mutation). The flow head are used for jumps. It points to the position where the other three heads will be moved when a reposition instruction is encountered. The Flow head itself can be repositioned by instructions \texttt{h-search} and \texttt{set-flow}. \texttt{h-search} finds a label within the genome and moves the flow head there. If the label is not found, then the flow head remains in the same position. \texttt{set-flow} will reposition the flow head to the location specified by the value in the CX register.

Each Avida organism has its own unique CPU instance. A CPU is created and assigned to an organism at the time the organism is created and put in the world. Programmers and researchers have the ability to access all the data regarding the CPU and the genome information for that specific organism. A
CPU is not a separate thread. The Avida world will schedule a single organism to run at a time, and all others have to wait in idle. The scheduling process is divided into a time unit called updates. During an update, a predetermined number of organisms are scheduled to run and after that the Avida world will consolidate the statistics for the population during that update.

4. Template Matching

Template Matching is a way of indirectly addressing a position in memory [5,6], equivalent to labels in other assembly language. When an instruction is referencing another position in the memory, a subsequent series of `nop` instructions will be read as the start of a template (Figure 2.3). Then the CPU searches through the genome linearly to find the position of the complement sequence. The end of the complement `nop` sequence will determine the jump-to position.

The three `nop` instructions, `nop-A`, `nop-B` and `nop-C` are cyclically complementary: `nop-A`’s complement is `nop-B`, `nop-B`’s complement is `nop-C` and `nop-C`’s complement is `nop-A`. So a template with sequence `nop-A, nop-B, nop-C` will have a complement with sequence `nop-B, nop-C, nop-A`. If the complement sequence is not found, then the CPU will not be able to use any position for the current instruction and therefore any jump attempt will fail to be executed. As a result, the program still executes sequentially. However, not every `nop` instruction sequence is considered a template, only those that immediately follow certain instructions. In this research, we only consider `nop` sequences immediately after the instruction `h-search` and the instruction `if-label` since these two instructions require a template to have functional meanings. The length of the template can be anything. By default, if a continuous `nop` sequence is present after `h-search` or `if-label`, then that entire sequence is considered one single template, no matter how long it is. Several templates might be overlapping with each other. Figure 2.4 shows an example.
Figure 2.3: The head of a template \texttt{nop-A nop-B}

Figure 2.4: Two overlapping template
For this example we have two templates. Template 1 has two \texttt{nop-A}s and template 2 has three \texttt{nop-A}s. On the third column, we have a sequence of three \texttt{nop-B}s, which is obviously the complement of template 2. However, the complement sequence also contains a sequence of two \texttt{nop-B}s, so the first two \texttt{nop-B}s are also considered complement for template 1. The two overlap with each other because one sequence happens to contain the other.

5. Nop as Modifiers

Sometimes changing the behavior of some instructions is necessary, but Avida does not support passing arguments into functions. Instead, Avida uses a subsequent \texttt{nop} instruction to specify behavior of certain instructions. Mostly, the \texttt{nop} instruction will affect which register the previous instruction will be reading or writing: \texttt{nop-A} representing \texttt{AX}, \texttt{nop-B} representing \texttt{BX} and \texttt{nop-C} representing \texttt{CX}. For instance, instruction \texttt{inc} increments the value in \texttt{BX} register by 1. If \texttt{inc} is followed by \texttt{nop-B}, then its behavior is not modified since the nop still tells it to change \texttt{BX}. However, if \texttt{nop-A} follows \texttt{inc}, the instruction will then increment \texttt{AX} instead by 1 and likewise, a subsequent \texttt{nop-C} will increment \texttt{CX}.

\texttt{Nop} instructions also affect some instructions that read or write two registers at the same time. For example, by default \texttt{swap} swaps \texttt{BX} and \texttt{CX}. A \texttt{nop-A} modifier will change the two registers into \texttt{AX}, specified by \texttt{nop-A}, and \texttt{BX}, which is \texttt{AX}'s complement. Similarly, \texttt{nop-C} will affect the registers \texttt{CX} and \texttt{AX} and \texttt{nop-B} will not change the default behavior.

\texttt{Nop} modifiers are also responsible for selecting heads for manipulation. One head manipulation instruction can only manipulate one head, but with \texttt{nop} modifier it can access all four heads. For example, \texttt{move-head} will move instruction head by default. A \texttt{nop-B} or a \texttt{nop-C} will change the head to read head or write head, respectively. A \texttt{nop-A} will not change the default \texttt{move-head} since it specifies the instruction head.
6. Memory Allocation

At the time an organism is assigned to a position in the Avida world, it has a memory size exactly the same with its genome. In order to replicate the genome and produce an offspring, additional space is needed to store the temporary unfinished offspring genome. The memory allocation is done by h-alloc in the default instruction set [5,6]. The h-alloc will insert the newly created memory between the end and the beginning of the genome, two special positions that Avida keeps track of on every organism. As a result, the parent’s genome start position meets the child’s start position and the parent’s end position joins with the child’s end. The newly allocated memory is initialized with nop-As, or random instruction, depending on the configuration settings.

After the replication process is complete, the child genome needs to be separated from the parent’s. The instruction h-divide will cut the child genome out and then restore both genomes into the circular shape. After the division, the parents CPU will be reset as though it were a new born organism. This action is to mimic the bacteria cell division in nature, where the parent cell splits in half, creating two new offspring.

Both instruction h-alloc and h-divide require a certain condition to be met for their successful execution. In other words, if the condition does not satisfy the requirements, then the two instructions will not be executed. For example, occasionally an organism will appear in the Avida world with a very long genome, about twice the size of the surrounding organisms. The failure of executing h-divide is one of the major causes of this condition. Unable to divide, the organism carries both its genome and the offspring genome.

7. Special Calculations

Not all Avidans get the same resources. Some are evolved to perform special calculations on the input, and the experimenter can determine whether to award
such behavior with more resources or not, as well as how much resources are awarded. These special calculations include: Not, Nand, OrN, And, Or, AndN, Nor, Xor, Equ. The input is a random number determined by the random seed that has been initialized when the Avida world was set up. An Avida organism first gets the input from the IO instruction. Then as the data propagates through the genome, calculations are done on the data by several mathematical or logical instructions (nand, inc, add, sub, etc). Then, another IO instruction outputs the end result to the environment and at the same time get a new input. Avida then compares the output with the original input to determine if the data has undergone those special calculations.

2.2 Analysis and Statistics

Avida analyses and statistics are mainly composed of an organism phenotype, organism genotype and population statistics.

**Phenotype** The phenotype of an Avida organism comprises all observable characteristics of that organism [5,6]. It has the same definition as in biological science. An organism will obtain certain merits and behaviors by interacting with the environment through input and output operations. Mostly these merits and behaviors are represented by the output number each organism provides after it has completed a required special computation on the input. Other information is also presented in the phenotype including fitness, gestation time (the number of instructions to be executed to create an offspring), age, interactions with surrounding organism and mutations [5,6]. The phenotype is the result of the organism genome’s execution, so analyzing the genome is an important way to gain some understanding of how certain phenotype comes about.
**Genotype**  Organisms are considered to have the same genotype if they have exactly the same initial genome. In an Avida population, there are potentially several genotypes that co-exist as the organisms keep dividing and mutating. In particular, the dominant genotype, the most abundant genotype in the population, is of special interest for population statistics because it represents the most successful population of organisms. For detailed information about genotype, refer to section 2.5.

**Population Statistics**  Population statistics are calculated from all phenotypes of all organisms in a population. Avida keeps track of several categories of values: the maximum (maximum fitness, maximum gestation time, etc), the average and the dominant. The dominate values are derived from the phenotype of the dominant genotype. This data is used in determining how well a population is doing, valuable information often used in comparing the impact of Avida environment settings on different populations with the same ancestor.

### 2.3 Program Slicing

Program slicing computes the set of program statements that may affect the values of interest at a given point in the program [15]. In this thesis, we combine the advantages of both static and dynamic slicing to overcome the difficulties of understanding Avida.

#### 2.3.1 Static Slicing

Static slicing uses only statically available information to show which statement impacts the value of a variable, called the slicing criteria [15]. It makes no assumptions on how the input will impact the program. Since no runtime information is ever used, the slice result contains all possible paths that could modify the slicing criteria. Figure 2.5 shows a program whose static slice for \( y \) is the whole program itself since no assumptions is made about the value of \( x \). Variable \( x \) is able to take on any value.
The two branches of the if statement both have a possibility impacting value \( y \), hence both branches have to be included in the slice.

In this thesis, the static slicing problem is defined as a reachability problem in the program dependency graph (PDG) [15]. When a vertex in the program dependency graph is chosen as the slicing criteria, the slice will contains all vertices that can reach that criteria. Because this definition is used, static slicing Avida genome requires generating the PDG, which contains control flow information and data dependency information.

### 2.3.2 Dynamic Slicing

Dynamic slicing uses information available from specific executions of a program [15]. Once an input is specified, dynamic slicing extract the set of instructions that impact the slicing criteria is from that particular input. Taking the static slicing example above but assuming the input is 3, then the dynamic slice outcome will only contain one of the if branches. Figure 2.6 illustrates the slice. In order to calculate dynamic slice, several hybrid approaches were invented. Choi et al. [8], Duesterwald et al. [9], and Kamkar [10] used static information to reduce amount of computation. Similar
to these, this thesis also uses a hybrid approach because of the limitation of Avida.

Figure 2.6: Dynamic slice for a simple program

2.4 Alignment

Alignment is used in Bioinformatics to study the differences and similarities between two sequences of DNA or protein [16, 17]. Figure 2.7 gives an example of a simple alignment result. A vertical line means two positions match while a dash means that sequence has a gap for that element. Since Avida organisms also have genomes, the alignment technique can be used here to compare and contrast sequences of Avida program. The most commonly used alignment algorithms are Needleman-Wunsch [16] and Smith-Waterman [17] algorithm. Both methods use the Dynamic Programming technique and a scoring system to generate the alignment with highest score.

Figure 2.7: An example of an alignment
2.4.1 Global Alignment

Global alignment is used to find correspondence between two sequences of characters. The Needleman-Wunsch algorithm solve the global alignment using dynamic programming technique [16]. It uses a scoring system and a score matrix to assess a current alignment score based on previous decision and the current options (a match, a mismatch or a gap). It then chooses the option that produces the maximum alignment score for the whole sequence. At the end, the algorithm produces a maximum alignment score. The alignment that maximize the score can be re-produced by tracing back the score matrix.

2.4.2 Local Alignment

Local alignment is different from global alignment in that it searches for an alignment between the subsequence of two genomes that produces the best score instead of the entire sequence [17]. The Smith-Waterman algorithm is an modification of the Needleman-Wunsch algorithm in that it also uses the score system and matrix as in the Needleman-Wunsch algorithm. The main differences lie in where the best score is found in the matrix and how the matrix is initialized. The algorithm produces results that only have part of the genomes aligned, as long as the alignment score is maximized.
Chapter 3

Approach

The primary barrier facing Avida users doing micro-level analysis of organisms is the complexity of the language and the manifestation of that complexity in the genomes. Modern features of programming languages meant to help with organization, such as begin-end blocks and explicit encapsulation constructs, do not exist in Avida. The closest analogy to common programming languages to Avida would be any variety of assembly language.

Our approach for analyzing Avida organisms has its foundations in dependency analysis and program slicing. In this paper, we describe two approaches that we have taken in order to facilitate the understanding of the structure and behavior of Avida organisms. First, we generate graphs that represent data and control dependencies within Avida programs. These graphs provide a visual representation to users in order to provide a sense of relationships between program instructions. Second, we apply slicing, both static and dynamic, to capture how different aspects of an Avida program impact outputs, and thus behavior, of the organisms.

The flow chart in Figure 3.1 shows the steps of the proposed methodology for analyzing Avida genomes. While obtaining information on any one individual organism is interesting, the capture of information on several generations of organisms enables the user to track the evolutionary process and identify building blocks upon which
more complex traits are based.

First, an Avida experiment is run to obtain data on organisms and populations (Process A). When the experiment is running, several organisms are extracted using the Avida Organism Freezer (Process B). The selection criteria depend on the interests of the particular user. However, to make later steps more meaningful in this research, organisms that complete multiple tasks are preferred. An Avida run generates a history file that records all organisms that appeared in the population throughout the experiment (Process C). Once the extracted organisms are ready and the history file is complete, a sequence alignment analysis [18] (Process D) is performed on the extracted organisms. Here, the history file is used to align the comparable portions of the organisms to facilitate analysis of the impact of evolution across multiple generations.

Then static slicing and dynamic slicing are used to analyze the genome (Processes E and F, respectively); this paper focuses on the slicing portion of the analysis. The results of these analyses are combined to explain how the organism’s instructions are executed and how data is processed (Process G). These results are then validated using Avida-ED, a graphical user interface for Avida [19]. Specifically, this step confirms that a given slice alone can complete a particular task (Process H). Such information, when combined with the alignment result (Process I), can show how these regions evolve over generations, thus providing insight into the evolutionary process for the organisms.

Information about an Avida organism comes in the form of dependencies among the instructions that comprise its genome. We have developed tools to generate graphical representations of the genome in order to visualize these relationships; specifically, control flow graphs (CFG), dead code graphs (DCG) and data dependency graphs (DDG). In the examples that follow, we demonstrate each of these using code snippets from genomes. We note that for the sake of completeness, Avida often uses a mnemonic representation of genomes, with a single alphabetic letter representing
each instruction. In our examples, we will list instructions side-by-side with their mnemonic representations.

### 3.1 Flow Graphs

#### 3.1.1 Sequential Execution

A control flow graph (CFG) shows the possible execution paths of a program [20]. Most instructions in the default instruction set will execute sequentially, with the program counter incremented after each instruction. In the corresponding CFG, the direction in which the program counter moves is shown as a arrow pointing from one instruction to the one directly below it (see Figure 3.2). Other statements that implement skips and jumps, such as `if-n-equ`, `if-less`, `if-label`, `jump-head` and `mov-head` are discussed below.
3.1.2 Conditional Instructions

The if-n-equ, if-less and if-label instructions are conditional branching instructions in the default Avida instruction set. For if-n-equ and if-less, Avida will decide whether or not to skip the next instruction based on comparing the values of two registers (as determined by nop modifiers). If a nop modifier is present immediately after the if statement, then the instruction immediately after the nop will be taken if the condition holds. For instance, Figure 3.3(a) depicts the if-less case, where the push is executed if the value of register A is less than the value of register C, and executes nand otherwise.

![Diagram of conditional instructions](image)

Figure 3.3: Conditional Instructions

The if-label instruction is important for copying instructions during replication.
The instruction interprets the sequence of \texttt{nop} instructions that immediately follow it as a \textit{label}. If the label is the complement of the most recently copied instructions, then the next instruction is executed, otherwise it is skipped. If an \texttt{if-label} does not have a following label, it skips the next instruction. As is shown in Figure 3.3(b), the node where a decision is made will have two emanating edges. One points from the node to the skipped instruction and the other points to where the next instruction is if a skip is going to occur. For instance, the \texttt{h-divide} in Figure 3.3(b) might be skipped after the \texttt{nop-C}, \texttt{nop-A} label.

3.1.3 Jump Instructions

The \texttt{jmp-head} and \texttt{mov-head} instructions affect the program counter by changing program flow based on a few different conditions. The \texttt{jmp-head} instruction will read the value of the \texttt{CX} register and advance a specific program control element (such as program counter as the flow control element) by that amount. The \texttt{mov-head} instruction will move a program control element to the position of a marker, usually specified by the \texttt{h-search} or, less commonly, by the \texttt{set-head} instruction. A modifying \texttt{nop} instruction will specify which control element will be moved; by default \texttt{nop-A} moves the program counter and the other two \texttt{nop}s move self-replication control elements.

If the \texttt{h-search} instruction is followed by a template, the virtual CPU will search the genome for its complement. If found, the CPU will mark the end of the complement template as the jump target position. If the \texttt{h-search} instruction is not followed by a template, the jump target position is set to the instruction immediately following the \texttt{h-search} instruction. Then, upon encountering a \texttt{mov-head} instruction, the jump target is set to the location that was determined by a \texttt{h-search}. For instance, consider Figure 3.4(a). The \texttt{h-search} is not followed by a template and thus the jump target for the \texttt{mov-head} is set to the \texttt{h-copy} that immediately follows the \texttt{h-search}. The \texttt{set-flow} instruction will set the control flow marker to the position specified by the value in the \texttt{CX} register. If the value is not valid, then the marker will remain
at its current position.

(a) Loop made possible by jump

(b) Dead code graph

Figure 3.4: Examples of Jump and Dead Code

3.2 Dead Code Graph

A dead code graph (DCG) is constructed using dead code analysis [21]. The approach for marking diagrams with dead code is based on a simple depth first search algorithm that colors nodes when visited. Figure 3.4(b) shows an example of dead code where the last three nodes have no incoming edge.
3.3 Data Dependency Graphs

A program dependency graph (PDG) shows all dependencies of a statement, including data dependency and control flow information. However, this research only concerns the data dependency of a PDG for reasons stated in section 3.11.

A PDG with only data dependency information is generated as follows:

1. For every statement, list the registers from which it reads and to which it writes.

2. Starting at the last statement \( i \) of the genome, traverse the genome backward, marking the current statement as visited.

3. If statement \( A \) depends on statement \( B \), then create a directed edge pointing from \( A \) to \( B \). The label on the edge identifies a register with which the dependency is associated. The graph edges are labeled according to the following scheme: \( AX = 1 \), \( BX = 2 \), \( CX = 3 \), \( Stack = 4 \).

4. For a statement that has a nop modifier, make the modifier a node and denote it as modifying the statement using a dotted arrow.

5. Stop traversing when either a visited statement or the beginning of the genome has been reached.

6. Repeat steps 2 to 5 for statement \( i-1, i-2, \ldots \)

Figure 3.5 shows all data dependencies for all statements in the genome represented by \( uqn11qn \), a short piece of a sample Avida genome.

3.4 Partial Static Slicing Analysis

As introduced above, we are using the reachability of the program dependency graph (PDG) as definition of our static slice. The construction of PDG requires the knowledge
of the program control information (a.k.a the order of execution) and data dependency. However, in this research we use partial static slicing because we ignore the control information, instead focusing only on the data dependency part of PDG, or a data dependency graph (DDG), for various reasons. First, due to the dynamic nature of Avida genome, extracting the control information is difficult in a static analysis, largely due to the presence of templates (a delimiter for loops, see section 1.3.1). There are only a few template defining instructions, but there is no restriction on where the compliment might appear. Often there are more than one compliments appearing in the same genome. It is extremely difficult tracking these compliment positions and deciding which one is actually used. Second, Avida organism’s behaviors is dependent on the dynamic environment. Seperating the environment to analyze Avida genome does not make sense in that if the environment changes, the behavior will vary significantly. For these reasons, the static slice analysis is a partial analysis tool that will help analyze an Avida genome dynamically where the control flow information and environmental factors are extracted at running time.
Theoretically, almost all information about the organism (except mutation) is stored inside its genome: information on its interaction with the environment is based on how the organism processes the input data; its life expectancy is determined by the semantically correctness of the genome; its reproduction information is encoded in the program loop, which copies the program itself at the end of the genome. Therefore, in order to understand the organism behavior, we need to first understand the program that represents the organism genome. In this thesis, we are most interested in how Avida organisms interact with the environment because this information not only heavily influences its life expectancy and its reproduction, but it is also closely related to the population behavior and evolution process. For instance, the rewards organism gained from processing input data will elongate its life expectancy. If the organism lives longer, then it can reproduce more and therefore the population will be more robust than others.

To understand how data is processed, we take the following program slicing approach. Instead of looking for inputs and their corresponding output, we look for each output data to see how it was modified in the past. The problem with looking for inputs is that we wont know at the beginning of the analysis if the input is actually used in producing any output data. Conversely, each output must have a data origin we can trace back to. Under this approach, program slicing with a program dependency graph seems to be a good choice of a tool to solve the problem. Program dependency graphs capture data modification and control flow relationships among instructions and program slicing will isolate the path of these modifications for each output data.

The program slices we produced this way will isolate a particular computational function from the rest of the genome for producing a specific output. To obtain such a slice, we will trace back to the origin of the data through the genome and record each instruction which changed the data. The trace-back stops at either the beginning of the genome or at an IO instruction, where a new number is brought in to replace
Figure 3.6: DDG and slices of output IO instruction from a simple Avida genome
the output one. It is obvious that if we can trace an output number, through all its changes, back to its original input, then this output is reachable from that input in the PDG, hence all recorded instructions should be included in the slice. For example, the DDG in Figure 3.6 represent the dependencies of an easy Avida program. One thing worth noting is that the IO instruction in Avida affects the value of input that subsequent IOs obtain. It does this not through affecting register value, but through change the environment value outside the organism. For accurate slice result, each IO was deemed dependent on previous IOs and therefore it must be included in the graph even though it does not have data dependency through registers.

For each Avida instruction, its definition in the instruction set specifies which registers it will read from or write to. For an output, it has to be stored in one of the three registers for an IO output to read. We can find the last previous instruction which wrote to that register (changed the data stored in the register) according to the instructions definition in the instruction set. By definition of PDG, the output is dependent upon that instruction and they are connected in the graph. From there we can find which register that instruction read from and connect the next dependency. If we keep tracing back the dependencies, then we can find a path where the output can be reached from an input. This path will be the static slice. The algorithm for finding the path in the PDG is described in the pseudo code below.

**Algorithm 1** Avida Genome Slicing

**Ensure:** A list of genome slices for each IO statement

```
List L
for each IO i in genome do
    List L_i
    List R ← the registers which i reads from
    index ← i’s program counter
    L_i.add(i)
    for each r in R do
        Traceback(r, L_i, index)
    end for
    L.add(L_i)
end for
```

---

28
Algorithm 2 Traceback(Register R, List $L_i$, int index)

Require: $R \geq 0 \lor L_i$ not NULL \ \forall \ index \geq 0

Ensure: Fill $L_i$ with sliced out statements one at a time

\[
\text{readPos} \leftarrow \text{index}
\]

for int $i \leftarrow \text{index - 1 to 0}$ do

if readPos == index then

if Instruction $q_i$ at $i$ writes to $r$ then
    \[
    \text{readPos} \leftarrow i
    \]
end if
end if
end for

if readPos == index then
    return
end if

$L_i$.add(Instruction $q_{\text{readPos}}$ at readPos)

List NewR ← Registers $q_{\text{readPos}}$ reads from

for each nr in NewR do
    Traceback(nr,$L_i$,readPos)
end for

For Avida organisms, we are interested in the slice of genomes that perform the special operations such as AND, XOR, etc since there are the operations that can gain merits for the organism, which determines the organism behavior. The IO statement outputs these operation result to Avida and at the same time obtains a new value. If we investigate each IO statement, then we are analyzing all possible positions in the genome where the special operation can bring impact upon the organism, thus finding out program slice. However, there is no way to confirm which IO statement will produce a result at run time because dynamic data is not used in static analysis. Therefore all IO must be included in our analysis.

The limitation of this approach is the lack of control flow information in the final slice result. Avida genome execution order, or control flow, is determined at runtime. The static slicing analysis is unable to predict the value stored in the register at runtime. As a result, the slicing algorithm cannot tell the order of execution just based upon the genome code. For example, the if-n-equ statement compares two registered value and determines whether to skip the next instruction. Potentially
these skipped instructions will end up in the slice, which will skew the slicing result. Also, the aforementioned template in the genome will sometimes dictate the genome execution order at runtime. There could be potential jumps or loops when the genome is executing. Such skips, jumps and loops might be critical for the functional region in the genome but they will not be reflected in the result of static program slicing. This limitation is overcome in dynamic slicing analysis.

3.5 Dynamic Slicing Analysis

The limitation of partial static slice analysis is the lack of control flow information in the final slicing result. A genome execution sequence, or control flow, is determined at run time and as such, the partial static slicing approach is unable to predict the value stored in registers at runtime. For example, the `if-n-eq` statement compares two register values to determine whether to skip the next instruction. Potentially these skipped instructions will end up in the slice, which will skew the slicing result by making the graphs larger than would actually occur in the execution. Also, the aforementioned templates in the genome will sometimes dictate the genome execution order at runtime. Consequently, there could be potential jumps or loops when the genome is executing. Such skips, jumps and loops might be critical for the functional region in the genome but they will not be reflected in the result of static program slicing [22]. To address these issues, dynamic slicing is used [23, 24].

Although dynamic slicing analysis is crucial for understanding the organism, it is very difficult to modify the Avida platform to allow us to insert dynamic analysis functions into it. It turns out that any major modification risks altering how the Avida platform is executed. We take a hybrid approach for dynamic analysis [25]. The known limitation to the static analysis is its lack of control flow information from the genome code. To obtain this information, we use the Avida-Ed [19] environment to “run” organisms; that is, to run an evolutionary experiment within the environment.
As the genome is running, the currently executing instructions are recorded into a new genome. In this way, the skipped instructions will not appear in the new genome and the repeated ones will appear multiple times at their corresponding location in the genome. The new genome contains all information the original genome has and also the control flow information which the original lacks. We then apply the static program slice algorithm on the new genome.

To analyze Avida genome, we need to understand the slices in the genome that perform special operations such as AND, XOR, etc, since these operations enable organisms to increase their merit and execute faster. The IO statement outputs these operation results to the Avida virtual machine and at the same time the IO obtains a new value to replace the old one. Since the IOs are the only instruction an organism can use to exchange values with the Avida virtual hardware, all special calculations must be outputted by an IO instruction in the genome. Therefore tracking down each IO and performing slicing on it will give us the slice we want for the calculations. However, there is no way to statically confirm which IO statement will output a meaningful calculation result. Therefore, all IO instructions must be included in our slices. It is necessary to check the slices for its validity and its actually output. Avida-ED conveniently provides a nice visual tool. The slices will be put in as an organism into Avida-ED and the resulting special calculation, or lack thereof, can be visually checked.
Chapter 4

Sample Analysis

In this section we describe an example that demonstrates the proposed approach for analyzing Avida genomes. The genome that we present is shown in Figure [4.1] and is represented by the mnemonic string: rucavcqgcelpcppqpyuugjcoicfdicscjqcznp-bruutycvsvab

The first step is to visualize the genome using our control flow graph, dead code graph and data dependency graph. The control flow and dead code graph presents an understanding of the order in which the genome will most likely be executed. Again, the static data has its limitations. In the control flow graph it also shows a possible template that is being used for jumping. In the data dependency graph, it shows how each instruction is dependent on the register value from a previous instruction. The number of the arc shows which register is used: 1, 2 and 3 being register A, B and C; 4 being the stack.

To aid in our analysis, we use the environment known as Avida-ED [19] to visualize and capture information about genomes. The environment animates movement through a program counter in a circular representation of the genome. In addition, Avida-ED indicates on a timeline when different tasks (functions) have been completed by the organism, along with counters indicating how many times each task was completed. As mentioned earlier, when an organism completes tasks it increases
1. h-alloc  27. nop-C
2. h-search  28. pop
3. nop-C  29. if-n-equ
4. nop-A  30. swap
5. mov-head  31. nop-C
6. nop-C  32. h-divide
7. IO  33. nop-C
8. push  34. IO
9. pop  35. shift-r
10. nop-C  36. nop-C
11. if-less  37. set-flow
12. inc  38. h-divide
13. nop-C  39. add
14. IO  40. nand
15. nand  41. nop-B
16. nand  42. h-alloc
17. IO  43. h-search
18. nand  44. h-search
19. if-label  45. h-copy
20. h-search  46. if-label
21. h-search  47. nop-C
22. push  48. mov-head
23. shift-r  49. h-divide
24. nop-C  50. mov-head
25. sub  51. nop-A
26. swap  52. nop-B

Figure 4.1: Expanded form of the example genome

its merit and executes faster relative to other organisms.

Figure [4.2(a)] depicts the memory representation of Avida-ED the genome in Figure [4.1]. The representation begins at the 3 o’clock position and the program counter proceeds in the clockwise direction. The dark circle indicates the current position of the program counter within the genome. Figure [4.2(b)] shows the function counter and indicates that the genome in question computes an ORN function. For this particular example, we have chosen a genome that is simple enough to demonstrate how the approach works. Accordingly, it only generates one of the desired functions, but enables us to construct a few simple static and dynamic slices.
4.1 Graph Generation

In order to help us visualize the hidden dependency in the genome, the control flow graph, data dependency graph and dead code graph were generated. Even though the information provided by the graphs is static, it still provide a preview on how the genome would be executed by highlighting potential template and dead code. The graph will also help to predict the result of the program slice by providing a initial view to the data and control dependency. The control flow graph (CFG) is shown in Figure 4.3(a), the dead code graph (DCG) is shown in Figure 4.3(b) while the data dependency graph (DDG) is shown in Figure 4.3(c).

4.2 Slicing

As mentioned in the Approach Section, our approach involves executing the genome in order to determine the function computed by the organism. We then generate the static slices for the genome. Once the slices are identified we execute each within Avida-ED to determine whether the slices compute any of the desired functions.

Using the slicing algorithm, the following static slices are generated: qqqugfq, qgfclcqppq, qq, ucq. The genome in bold is expanded in Figure 4.4(a) Figure 4.4(b).
Figure 4.3: Genome visualizations
shows the Avida-ED view of the static slice $qgfclcqqpq$.

As noted earlier, the static slicing approach may be unable to detect the functions computed by the genome. To determine whether they do or not, each is executed within Avida-ED. In the case of the static slices found using our algorithm, execution in Avida-ED reveals that they do not compute the functions of interest, including the one shown in Figure 4.4(b).

To cope with the shortcoming in the identification of the computed functions in the static slices, we generated the dynamic slices by using information found in the execution traces of the original genome. The generated slices are $qqqugf$, $qgfcqppq$, $qq$, $ucq$.

Figure 4.4: Static slice $qgfclcqqpq$

Figure 4.5: Dynamic slice $qgfcqppq$
As with the static case, we executed each of the resulting slices and found that the slice \texttt{qgfcpppq} generates the desired computation, ORN. The Avida-ED depiction of the slice is shown in Figure 4.5(b).

The difference between the static slice shown in Figure 4.4 and dynamic slice in Figure 4.5 is the inclusion of statements \texttt{l (increment)} and the modifier \texttt{c (nop-C)} after \texttt{l}. If we look at the data dependency graph of the static slice graph (Figure 4.6), the \texttt{l} increment should be included in the slice. However, in the original genome, before the \texttt{l (increment)} there is an \texttt{e (if-less)} statement. At runtime, the if-less checks register BX and CX and decides to skip the next instruction. As such, the statement is not executed at runtime. In the dynamic slice the skip is not included and the dynamic slice gives the correct slice. The static slice fails to compute the ORN result because it does not contain any runtime control flow information.

Figure 4.6: The data dependency graph of the ORN static slice
4.3 Sequence Alignment Analysis

For a specific population, its history of evolution is the process in which the organisms gradually change from the ancestor organism to the current form. To understand this process, we need to know how the current organism is different from the organisms in the past. We use sequence alignment techniques to spot the differences as well as the similarities between the current organism and the history. Sequence alignment techniques look for matches between two genomes and try to produce an alignment that shows the highest score in similarities. By doing so, we can highlight the part of the genome that was probably preserved from the past and then look for the parts that have changed through time. We also look for patterns of changes that are under natural selection pressure. For instance, if a section of a genome produces a special calculation that can confer significant advantage, then that section is usually stable through time. Whereas a section that drags the performance of the organism tends to change frequently. The alignment algorithms used here are standard Needleman-Wunsch global alignment algorithm and Smith-Waterman local alignment algorithm [16,17]. Global alignment aligns entire genomes, which provides information on how an organism evolves through time. Local alignment aligns parts of the genome, which enable us to identify functional region in a genome using the program slicing result of related organisms.

The alignment algorithm takes two sequences as input. Then the user will arbitrarily assign scores to three possible situations: match, mismatch and gap. Using dynamic programming technique, the algorithm looks for the best alignment score. The scoring system in this thesis is set to be 1 point for match and −1 point for mismatch and gap. If we trace back how the best score is obtained, we can get the alignment that produces the score. Figure 4.3 illustrates one possible outcome for alignment those two short Avida sequences.
Figure 4.7: Alignment result for two short DNA sequence with Match score 2, mismatch score -1 and gap score -1

4.3.1 Global Alignment

The global alignment was used to identify changes and trends in the evolution process. The standard Needleman-Wunsch algorithm of global alignment was implemented to perform the alignment. One of the inputs to the algorithm was the selected Avida genome of interest, while the other input was parsed from the .history file automatically produced at the end of each Avida experiment. The .history file records every organism that was produced during the experiment in chronological order using Avida CPU cycle count as one time unit. Our selection criteria for the genome of interest was rather simple. We chose a genome that has a relatively large count of special computations. Such a genome provides more traits to be observed, since the goal of this research is to demonstrate possibility of visualization Avida evolution process. The selection criteria can be adapted to other research goals. For example, researchers that concerns only the EQU computation trait of the organisms can select organism with only EQU computation as their genome of interest. Below shows the genome we selected.

```
grrdhzalpvcocnlqdpplhivcsoagc0oafzgspachgqaqcojaqzqiyizdpbnafpiaq
fmpqisqnxsyhuttttycastvgab
```

Once the alignment finishes, each alignment will be outputted as the format shown in the background section in chronological order and further analysis will begin.

4.3.2 Local Alignment

The local alignment was used to identify similar functional regions shared among generations of Avida organisms. From observing Avida experiment, it is not difficult
to find that the special functions tends to remain in the population once it is evolved. The function gains merits for a organism so that the organism is more competitive against other non-function organisms. The offsprings of the organism who inherit the functional region also tend to fair better than their conterparts and produce more offsprings. This is called positive selective pressure of evolution. If a set of organisms are found to have the same functional regions, then we can trace it back to the ancestor who evolved the functional region. From there it is possible to use other Bioinformatic tools (such as phylogenetic trees) to better understand the Avida evolution process.

Local alignment can also be used to locate alternative organisms with the same functionality, which can be applied to genome optimization. For example, one EU-performing genome was isolated, but it lacked a complete structure for self-replicating. In order to find a full function organism, the EQU slice would be isolated and local alignment would find a set of potential candidates which contains the functional region. Then Avida-ED can be used to find the desired genome.

The standard Smith-Waterman local alignment algorithm was implemented. Again, the genomes from the .history file was parsed and was used as one of the input to the algorithm. The other input was a program slice of the special computation from the genome of interested. Just as with the global alignment, the output is a text file with the alignment shown in chronological order.
Chapter 5

Implementation and Verification

The proposed solution has been coded as four separated C++ programs that implement static slicing, dynamic slicing, global alignment and local alignment. A web version of the software is located at [http://cpath.csi.muohio.edu/huh/test/cbuild/work/](http://cpath.csi.muohio.edu/huh/test/cbuild/work/).

5.1 User Study Design

The Avida genome analysis tools are designed for Avida users. Hence the quality of our research is determined by the user experience. We designed and conducted a user study to collect feedback in three areas of interest: usefulness, effectiveness and usability. For usefulness, we looked at whether the analysis software addresses the real needs of Avida users. For effectiveness, we looked for evidence that our method of visualization and analysis provided a way of helping Avida users understand Avida genomes. For usability, we looked at user responses to see how easy it is to use the software.

The user study was designed as a Pretest-Posttest Experiment Design [26]. Each participant visited our online test site. After consent, the participants filled out a pretest online survey, finished a set of tasks designed to cover all aspects of the analysis software and then filled out a posttest online survey. All information entered
during the study remained anonymous.

## 5.2 Research Questions

The research questions were designed to evaluate the three focus areas: usefulness, effectiveness and usability. The research questions are outlined in Table 1. The first three questions (RQ1-RQ3) address usefulness of the three analysis tools by asking whether the analysis tools solve real issues that Avida users face. The next three questions (RQ4-RQ5) address effectiveness by questioning whether the representation of the analysis results help users understand Avida genomes. The last question (RQ7) addresses usability.

- **RQ1.** Do Avida users find understanding Avida genomes difficult?
- **RQ2.** Do Avida users find understanding Avida evolution during an experiment necessary?
- **RQ3.** Do Avida users need to find similar functional genomes for comparative research?
- **RQ4.** Does the program slicing tool help users understand how Avida organisms perform the special calculations?
- **RQ5.** Does the global alignment tool provide a way to understand Avida evolution?
- **RQ6.** Does the local alignment tool provide a way to locate similar functional regions among population?
- **RQ7.** Are the program slicing, global alignment and local alignment tools easy to use?

## 5.3 Pre-test Post-test Design

There were three sections in the Pretest-Posttest Design. First, a pretest survey was given to each participant. This survey was designed to facilitate understanding the
general background of the participant and to collect initial impressions on the idea of an Avida analysis tool. Then, a set of tasks was given so that users would use the software and experience the functionality provided by the software. Upon finishing all the tasks, a posttest survey was given to evaluate the correctness of the task that users performed and to collect user feedback about the software. The task result was collected by asking users to copy and paste their task result to the questionnaire text area. Other pretest and posttest survey questions were designed to let users rate their experiences on a scale from 1 to 5 (e.g., 1 - strongly disagree, 2 - disagree, 3 - neutral, 4 - agree, 5 - strongly agree).

5.3.1 Pre-test Design

The pretest survey was administered prior to performing the tasks. The survey enabled us to understand the background of the participant. The survey questions mostly considered the following aspects:

1. How much Avida and Avida-ED experience did the participant have?
2. How difficult it is to analyze Avida genomes?
3. How difficult it is to analyze Avida evolution?
4. How much computer science and bioinformatics experience does the participant have?

The first aspect indicated how familiar the participant is to Avida programs and how important it is in their research. This affected the degree in which the participants’ rating of the effectiveness and usefulness of our analysis tools, assuming the less a participant uses Avida, the less important it is to him or her. The second and third aspects indicated the desire for an analysis tool. The fourth aspect collects technical information concerning the participant population.
5.3.2 Tasks

The tasks enable participants to experience the functionality provided by the Avida analysis tools. They also showcase some potential applications that we think might be useful to Avida users. The tasks were designed with the 7 research questions in mind and to try to give the user enough experience to rate the effectiveness, usefulness and usability of the analysis tools.

Task 1 tries to address RQ1 and RQ7. It presents participants with a genome and asks them to use the tools to find all program slices and validate the slice correctness. The participants are given an opportunity to use program slicing and to see its potential to identify short functional regions from a long genome. Task 2 addresses RQ2 and RQ7. It provides participants with a genome and a history file listing all other genomes in the evolution in chronological order. The participants were asked to find the global alignments between the given genome and genomes in a list. Then, the results were presented to them so that they could see the potential global alignment of identifying evolution trends. Task 3 tries to address RQ1, RQ3 and RQ7. It asks participants to first find the a slice of the given genome, and then find the local alignment between the slice and the history file genomes. The participants will be able to observe how local alignment can be used to identify similar slices in other genomes. Task 4 tries to answer RQ4 to showcase the capability of using the program slicing tool to reconstruct a new genome that performs desired special calculations. Task 5 tries to answer RQ5 and asks participant to find a conserved region using global alignment. Task 6 tries to answer RQ6 by having the participant pick a genome among the local alignment results and validate whether or not it contains a required special calculation.

5.3.3 Post-test Design

The posttest design was given after participants finished all the tasks. It consists of two sections. In Section 1 the participant submits their task result. This data
was used to determine whether participants finished the given tasks correctly. An incorrect result may influence their opinion on the effectiveness and usefulness of the analysis tools. It may also indicate that the usability of the software needs improvement. Section 2 lets participants directly rate the effectiveness, usefulness and usability based on their experience. The questions scale from 1 (strongly disagree) to 5 (strongly agree).

5.4 Testing Site

A website has been created for the purpose of evaluating the software. The website contains 5 sections that correspond to the pre-test post-test design.

The first section is the pre-test survey. It was constructed in Google forms as shown in Figure 5.1.

The second section describes the tasks. It also provides a page for hosting intermediate results. The page is shown in Figure 5.2.

The third section is the program slice page. A user inputs a genome into the text area and the page will refresh and return the slice as shown in Figure 5.3.

The fourth and fifth sections are for global and local alignment. In both pages, the user is asked to input one genome in one text area and then a set of genomes to be aligned with in the next text area. The pages then refresh and the user can download their alignment result as a text file through a link, as shown in Figure 5.4.

The final section is for the posttest survey. It was constructed in Google forms similar to the pretest survey. A sample of the page is shown in Figure 5.5.
### Pretest Survey

This survey is for testing the program slicing and alignment tools for Avida genomes before the user begins the tasks. Please answer each question. Thank you.

* Required

**Your research area?**

**How long have you been using Avida?**

**I often use Avida to help with my research.**

1 2 3 4 5

**I often use Avida-ED for visualizing Avida populations and organisms.**

1 2 3 4 5

**Figure 5.1: Snippet of a history file**
Pretest Survey  Tasks  Program Slice  Global Alignment  Local Alignment  Posttest Survey

Please finish the following task using Avada analysis tools and save the results to answer the post-test survey. To finish these tasks, you will need a plain text editing software (Windows Notepad) and Avada-EI Download Avada-EI here.

Click here for the result page. Please save your results here. It will be used in the post-test survey.

Task 1: Program Slicing

Given genome:

_asm.xxx112233445566778899aabbcc

Find the EQU slice in Avada-EI and find the invalid slices in Avada-EI.
1. Copy and paste the above genome into the text area on "Program Slicing" page, click submit.
2. You can view the expanded form of the slices by clicking on each slice.
3. Open Avada-EI and follow the instructions to check the validity of each slice. You are looking for a slice that performs EQU after executing its last EOD instruction (denoted as q in the genome).
4. Please copy and paste your EQU slice to the result page above. This will be used in the post-test survey.

Task 2: Global Alignment

Given genome:

_asm.xxx112233445566778899aabbcc

and all other genomes appeared in the same Avada experiment (see those genomes here), generate the global alignment for the given genome showing how it evolved from the ancestor.
1. Copy and paste the above genome into the first text area on "Global Alignment" page.
2. Go to "Global Alignment" page and click on "Task 2 Submit" button.
3. View the global alignment by following the newly generated link.
4. Please copy and paste the first three lines of your alignment to the result page.

Task 3: Local Alignment and Program Slicing

Given genome:

_asm.xxx112233445566778899aabbcc

and all other genomes appeared in the same Avada experiment (see those genomes here), generate the local alignment for the XOR slice. 1. Copy the XOR slice (gfpqcrqgppgpygqpygpqgpygpqgpqgpygpqgpygpqgpy) into the first area below.
2. Go to "Local Alignment" page and click on "Task 1 Submit".

Figure 5.2: Task Page

Pretest Survey  Tasks  Program Slice  Global Alignment  Local Alignment  Posttest Survey

A short explanation of program slicing

On Avada genomes, Program Slicing is used to identify the functional region of an Avada organism. It produces a subsequence of the genome that performs a special calculation.

For example, genome:

_asm.xxx112233445566778899aabbcc

contains several special calculations including EQU. Now copy and paste it into the text area below and click submit.

Now you should see several smaller genomes below. These are the slices. You can check these slices in Avada-EI to see if they do produce the desired function. Because of the algorithm used, some of them are not valid, but the slices will encompass all special functions present in a genome.

Details and download of Avada can be found here.

Instructions on using Avada-EI to check slice validity

Enter your genome in genome form (no space or special characters):

Submit

Figure 5.3: Program Slicing Page
Global alignment compares one genome to another to identify similarity and difference. Global alignment align two entire genomes. We think it will be useful in looking at how genomes differentiate and change in the evolution process. You can input the genome of interest into the first text area and then align it with several genome (separated by new line) in the second text area.

Here is an example of the output and its meanings:

<table>
<thead>
<tr>
<th>Seq1:</th>
<th>ABBCCBBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line:</td>
<td>A vert. line means match</td>
</tr>
<tr>
<td>Seq2:</td>
<td>A----BBA</td>
</tr>
<tr>
<td></td>
<td>A dash means a missing genome element compared to the other sequence</td>
</tr>
</tbody>
</table>

Enter first genome, one genome only:

Enter second genome, multiple genomes separated by new line:

align
<table>
<thead>
<tr>
<th>Pretest Survey</th>
<th>Tasks</th>
<th>Program Slice</th>
<th>Global Alignment</th>
<th>Local Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Posttest Survey</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Posttest Survey**

This survey is for testing the program slicing and alignment tools for Avida genomes after the user finishes all the tasks. Please answer each question. Thank you.

* Required

1. Your research area?

2. Please copy and paste your EQU slice *

3. Please copy and paste the first three lines of the global alignment result. *

Figure 5.5: Posttest Survey Page
5.5 User Study Results

5.5.1 Participant Characteristics

There were twenty-two participants in the study but only twelve of them finished the entire survey. Others finished the pre-test survey but did not finish the post-test survey. The participants have diverse computer science backgrounds, including software engineering, machine learning, Bioinformatics and augmented reality. The participants are from Miami University and Michigan State University, though the exact number for each school is unknown since the survey is anonymous. A presentation was given to participants at Miami University explaining what Avida is and how Avida works. The twenty-three who finished the pretest survey provided an overview of the general attitudes towards Avida and the twelve who finished the whole survey revealed their attitude towards the usability of the software.

5.5.2 Pre-test Result

Overall Avida Knowledge

The first section of the pretest survey asked the following questions:

1. Q1: I often use Avida in my research
2. Q2: I often use Avida-ED for visualizing Avida population and organisms

These two questions measure how much participants have experience on Avida and Avida-ED by having participant to rate how often they use it in their research. The result is shown in Table 5.1. The result shows that most of the participant do not use Avida or Avida-ED.

Attitude Towards Avida Analytics

The second section of the questions on the pre-test survey is to gather participants’ opinion towards the difficulty of understanding Avida and the usefulness of a tool for
Table 5.1: Results for Avida background

<table>
<thead>
<tr>
<th>Opinion</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neutral</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>86.9%</td>
<td>4.3%</td>
<td>0.0%</td>
<td>4.3%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Q2</td>
<td>82.6%</td>
<td>8.7%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

supporting genome analysis

The first set of questions were aimed towards program slicing:

1. Q3: I care more about population than individual genomes.

2. Q4: There are many times when individual organisms are also important and I need to analyze them.

3. Q5: I find it difficult to understand Avida genomes by only looking at their sequences.

4. Q6: Observing the behavior of the organism is enough for my research. I do not need to fully understand how the behaviors are produced through their genomes.

Table 5.2 shows the result of the second part of the pretest.

Table 5.2: Results for program slicing questions

<table>
<thead>
<tr>
<th>Opinion</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neutral</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q3</td>
<td>39.1%</td>
<td>17.4%</td>
<td>30.4%</td>
<td>8.7%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Q4</td>
<td>43.5%</td>
<td>8.7%</td>
<td>34.8%</td>
<td>0.0%</td>
<td>8.7%</td>
</tr>
<tr>
<td>Q5</td>
<td>26.1%</td>
<td>21.7%</td>
<td>34.8%</td>
<td>8.7%</td>
<td>4.3%</td>
</tr>
<tr>
<td>Q6</td>
<td>34.8%</td>
<td>13.0%</td>
<td>34.8%</td>
<td>4.3%</td>
<td>8.7%</td>
</tr>
</tbody>
</table>

Q3 and Q4 shows that user generally cares about single organisms. Q3 reveals that most participants disagree that population analysis is more important than individual organism analysis. Q4 shows that more people do not analyze single organisms in their research. However, when looking at Q4 combined with results in Q1 and Q2, those who didn’t use Avida also rated low on Q4. For the two people who had experience using Avida in their research, their ratings are neutral and strongly agree.
Q5 asks participants if they find it difficult to understand Avida genomes. Most people disagree or remain neutral. At first it seems that understanding the genome is not a problem for most people. However Q6 provides more detailed information on the participants’ actual research need. Most participants who rated understanding Avida genome is easy also rated low on their need to fully understand Avida organisms. Those who rate higher on their need to understand Avida find it more difficulty to analyze Avida genomes.

The second set of questions in the second section is to obtain opinions towards the need for putting a single analyzed organism in the context of evolution. It contains the following questions:

1. Q7: I often need to analyze the evolution history of an experiment, meaning I need to look at how some of the organisms evolve over time.
2. Q8: I often need to freeze and extract the genomes of organisms to analyze it later.
3. Q9: After identifying an organism of interest, I often need to find similar Avida organisms in the same experiment.

The result is shown in Table 5.3. For Q7 and Q8, most people rated neutral or below on the need to work on entire history of an Avida experiment. It seems that once a desired organism is acquired, the history of how the organism evolve does not matter as much. Most people also rated neutral or below on the need of finding similar organisms for Q9.

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{Opinion} & \text{Strongly Disagree} & \text{Disagree} & \text{Neutral} & \text{Agree} & \text{Strongly Agree} \\
\hline
\text{Q7} & 52.2\% & 8.7\% & 26.1\% & 8.7\% & 0.0\% \\
\text{Q8} & 52.2\% & 8.7\% & 13.0\% & 13.0\% & 8.7\% \\
\text{Q9} & 52.2\% & 4.3\% & 34.8\% & 4.3\% & 0.0\% \\
\hline
\end{array}
\]
The last set of questions asks participants for their Bioinformatics and computer science background. It has the following questions:

1. Q10: I have heard and fully understand global alignment and local alignment used in Bioinformatics.

2. Q11: I often use global alignment and local alignment in my research.

3. Q12: I have a computer science background.

As the result shown in Table 5.4, few participants have experience with the alignment algorithm while most participants have a computer science background.

<table>
<thead>
<tr>
<th>Opinion</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neutral</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q10</td>
<td>34.8%</td>
<td>13.0%</td>
<td>26.1%</td>
<td>4.3%</td>
<td>17.4%</td>
</tr>
<tr>
<td>Q11</td>
<td>47.8%</td>
<td>13.0%</td>
<td>30.4%</td>
<td>4.3%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Q12</td>
<td>4.3%</td>
<td>0.0%</td>
<td>4.3%</td>
<td>4.3%</td>
<td>86.9%</td>
</tr>
</tbody>
</table>

### 5.5.3 Post-test Results

Twelve of the twenty three participants finished all the tasks and filled out the post-test survey.

**Task Result Correctness**

There is a question for each task asking participants to post their final result. The questions were designed to evaluate how accurate participants can follow the instructions, which reveals information about the software’s ease of use. The percentage of correct result to each task is shown in Table 5.5.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>52.3%</td>
<td>83.3%</td>
<td>33.3%</td>
<td>41.7%</td>
<td>50.0%</td>
<td>58.3%</td>
</tr>
</tbody>
</table>
The number shows relative low percentage of correctness for tasks that involve program slicing including Tasks 1, 4 and 6. Part of the reason could be that these tasks require participant to use Avida-ED to validate their result. It was a somewhat difficult task for a first time Avida-ED user to successfully complete the 9 steps necessary for validation. Task 2 has the highest correctness rate, indicating the global alignment task is the easiest to complete. Task 3 has surprising low correctness percentage given that it is mostly the same with Task 2. Part of the reason was that the wording of the instructions may have confused participants. The same confusion can be seen in Task 5, where most people seemed not to grasp the concept of consistently well-matched.

5.5.4 Software Usefulness and Usability

The last 12 questions ask participants to rate directly their opinion of each tool. For each tool, there is a set of three questions:

1. Q1: I find the software useful.
2. Q2: I find the software easy to use.
3. Q3: I will use the software in the future.

For program slicing, the feedback is shown in Table 5.6. The result of Q1 gives mixed views on usability with opinions polarized on both easy and difficult to use. The Q2 results shows the majority of participants found the tool to be useful. The Q3 shows only a few will potentially use the tool in the future, which is reasonable given that most participants do not research in this area.

<table>
<thead>
<tr>
<th>Opinion</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neutral</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>16.7%</td>
<td>33.3%</td>
<td>8.3%</td>
<td>8.3%</td>
<td>33.3%</td>
</tr>
<tr>
<td>Q2</td>
<td>8.3%</td>
<td>16.7%</td>
<td>8.3%</td>
<td>33.3%</td>
<td>33.3%</td>
</tr>
<tr>
<td>Q3</td>
<td>58.3%</td>
<td>0.0%</td>
<td>25.0%</td>
<td>0.0%</td>
<td>16.7%</td>
</tr>
</tbody>
</table>
For global alignment, the feedback is shown in Table 5.7. A little over 50% of participants thought the global alignment tool was easy to use. Most participant are neutral and below on possibility of future usage, which again is consistent with the participant pool.

For local alignment, the feedback is shown in Table 5.8. A little more people found the local alignment software easy to use despite the low correctness rate on the task. Most of them still thought it is useful and most participant still rated low on the future usage of the software.

### 5.5.5 Conclusion to Research Questions

Using the above result, we can answer the research questions to some degree.

**RQ1: Do Avida users find understanding Avida genomes difficult?** According to pre-test survey Q4 - Q6, people who find the need to fully understand Avida behavior do find it difficult to understand Avida genomes. For those who are not too concerned about the origin of organism behavior find Avida genomes easy to understand.
RQ2: Do Avida users find understanding Avida evolution during an experiment difficult? According to the result on pre-test survey Q7, it seems that few people find the need to analyze the evolution process for a given experiment.

RQ3: Do Avida users need to find similar functional genomes for comparative research? According to pre-test Q9, most people do not find the need to search for similar genomes among the population.

RQ4: Does the program slicing tool help users understand how Avida organisms perform the special calculations? According to the opinion on the slicing software, the majority of the participants viewed the program slicing tool as useful.

RQ5: Does the global alignment tool provide a way to understand Avida evolution? According to user opinion on the global alignment, the majority of the participants viewed it to be useful.

RQ6: Does the local alignment tool provide a way to locate similar functional region among population? Again, majority of the participants rated it to be useful.

RQ7: Are the program slicing, global alignment and local alignment tools easy to use? The majority of the participants rated the program slicing and global alignment to be useful. The local alignment tool was also rated easy by most participants despite some low percentage on the number of people finish the tasks correctly on local alignment. The low number might be due to the misleading instructions on the tasks.
Chapter 6

Conclusion

6.1 Discussion

The research described in this thesis focused on understanding and visualizing Avida organisms. For that purpose, several methods were proposed to present the genomes in a clear way, including program graphs, program slicing and alignment. The hypothesis was that Avida users will find them useful and will be able to use them to solve some of their real problems.

The first method this research explored was to generate program graphs. Three types of graphs were created in this research: Control Flow Graph (CFG), Data Dependency Graph (DDG) and Dead Code Graph (DCG). These graphs reveal how each instruction in a genome related to the others. CFG shows the order of execution in the genome. DDG shows where each instruction get the data from and shows how data is transfered inside the entire genome. DCG exams the part of the genomes that is never executed. These graphs visualize some basic information of Avida genomes and it was our first attempt at understanding Avida genomes. Some of the information in the graphs cannot be accurately obtained statically, such as how a template affects the flow of the program. The more in depth and accurate analysis requires dynamic information. Despite such limitations, the methodology and algorithm used
in generating those graphs were applied in the later program slicing analysis.

Static and dynamic program slicing was used to isolate functional parts of the Avida genome. A functional part is defined as the sequence within genome that is responsible for the special calculation. A program slice in this research is defined as reachability of the program dependency graph. The static slice was based solely on data dependency of the instructions since the limitation on statically predicting control flow still exists. To get the accurate control flow, a hybrid dynamic and static slice technique was used. In this method, a program first records the dynamic execution information into a new genome under the Avida environment. Then, the data dependency is resolved on the new genome. The end result is a slice that contains both control flow and data dependency information. Although the new genome might be different from the original genome, it is a snap shot of the original genome in execution. Therefore it contains, and displays explicitly, all the hidden control information that the original genome would have. Our result shows that this hybrid method indeed can extract the functional part of a genome.

This research also produces results for putting each organism into an evolutionary context. Global alignment was used on Avida genomes and populations to show the evolution pattern of that genome. Our results show that one can identify preserved regions in an evolution process. Further investigation into the regions show that those are either survival regions (i.e, regions that organism need in order to survive) or reward regions (i.e, functional parts that produce high rewards). The observation is somewhat consistent with the positive and negative selection pressure in the evolution theory [27]. Local alignment was used to identify similar functional regions among a population by aligning a known program slice against all genomes in the population. The goal was to use this method to find organisms that do the same special calculations. The results show that it is possible to perform this analysis using local alignment.
6.2 Future Work

We believe that our work is the first to apply program understanding to the problem of analyzing digital organisms. As such, this research provides some examples for understanding organisms in other digital evolution platforms.

Another unresolved challenge facing Avida users is to validate the evolution process and the end result of their experiment. Since everything happens randomly in Avida, it is difficult to understand them and quantify the result, hence no metric has set up to prove that Avida does accurately simulate natural evolution. This research provides a mean for understanding organisms and the evolution process. Future research might be able to address how to translate this understanding into measurable standards. Such standards can then be integrated into a system to validate the Avida platform as a whole.

Finally, the method used in this research can be improved. A more dynamic approach can be used in the dynamic slicing. Instead of a hybrid method, an entirely dynamic analysis tool will be more flexible at understanding different versions of Avida. Finally, Other Bioinformatics techniques, such as constructing phylogenetic trees, can also be applied to further investigate the evolution process.
Bibliography


