Co-evolutionary automated software correction: a proof of concept

Joshua Lee Wilkerson

CO-EVOLUTIONARY AUTOMATED SOFTWARE CORRECTION: A PROOF OF CONCEPT

by

JOSHUA LEE WILKERSON

A THESIS

Presented to the Faculty of the Graduate School of

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER SCIENCE

2008

Approved by

Dr. Daniel Tauritz, Advisor
Dr. Thomas Weigert
Dr. Bruce McMillin

ABSTRACT

The task of ensuring that a software artifact is correct can be a very time consuming process. To be able to say that an algorithm is correct is to say that it will produce results in accordance with its specifications for all valid input. One possible way to identify an incorrect implementation is through the use of automated testing (currently an open problem in the field of software engineering); however, actually correcting the implementation is typically a manual task for the software developer. In this thesis a system is presented which automates not only the testing but also the correction of an implementation. This is done using genetic programming methods to evolve the implementation itself and an appropriate evolutionary algorithm to evolve test cases. These two evolutionary algorithms are tied together using co-evolution such that each population plays a large role in the evolution of the other population. A prototype of the Co-evolutionary Automated Software Correction (CASC) system has been developed, which has allowed for preliminary experimentation to test the validity of the idea behind the CASC system. In these experiments, the CASC system attempts to correct various insertion sort implementations. The success of the CASC system in the insertion sort experiments demonstrates its potential, although further research is needed to prove its scalability and generalizability for general purpose use. The results of these experiments are presented along with a discussion of their significance.

ACKNOWLEDGMENTS

First I would like to thank my advisor Dr. Daniel Tauritz for contributing the idea behind the CASC project and for all of the excellent guidance he has provided me throughout this project and my graduate career. Without his help this work would have been much more arduous.

Also I would like to thank Dr. Bruce McMillin and Dr. Thomas Weigert for the guidance, assistance, and advice that they provided for this project.

I would like to thank Clayton Price for all of the advice and support he has given me throughout the years.

I would also like to thank Tess Baklor both for her contributions to the project and for the unfailing support she has given me throughout my work.

Lastly I would like to thank all of my family and friends for all the support and advice that they provided me.

This work was supported by the Missouri S&T Intelligent Systems Center.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

1. INTRODUCTION

This thesis describes the Co-evolutionary Automated Software Correction (CASC) system. As the name implies, the CASC system is used to perform automated software testing and correction. This is done by employing Genetic Programming (GP) techniques to evolve a software artifact while at the same time employing a problem appropriate Evolutionary Algorithm (EA) to evolve test cases. These two EAs are linked together to create a two-population competitive co-evolutionary system with each population indirectly guiding the other in its evolution. A high level depiction of the CASC system is shown in Figure 1.1.

In EAs, a fitness function determines how well an individual performs in the scope of the stated problem. One major strength of the CASC system concept is the reusability of a given fitness function to evolve multiple programs for the same problem statement. The CASC system exploits the reduced complexity of the fitness function relative to the software artifact being tested. The principle idea here is that if a fitness function that is not very complex can be determined for a particular problem, then that fitness function can be used to correct any program solution to that problem, regardless of the complexity of the solution. For problems that require complex algorithms, this is a particularly advantageous aspect of the CASC system.

The current state of the CASC system is an early prototype of the envisioned system. The largest difference between the envisioned system and the current system is that GP is used in a limited fashion to evolve the program population. Programs are represented by trees in the system and reproduction is done using sub-tree swaps (as per standard GP); mutation, however, is performed only on specific nodes in the program tree (this is discussed in detail in Section 2.3.2). Regardless, enough work has been done on the system to allow some preliminary experiments to be performed. The results from these experiments will be presented in Section 4.

The CASC system is an application of evolutionary computing to software testing, a subfield of software engineering. While this has been done before, this is the

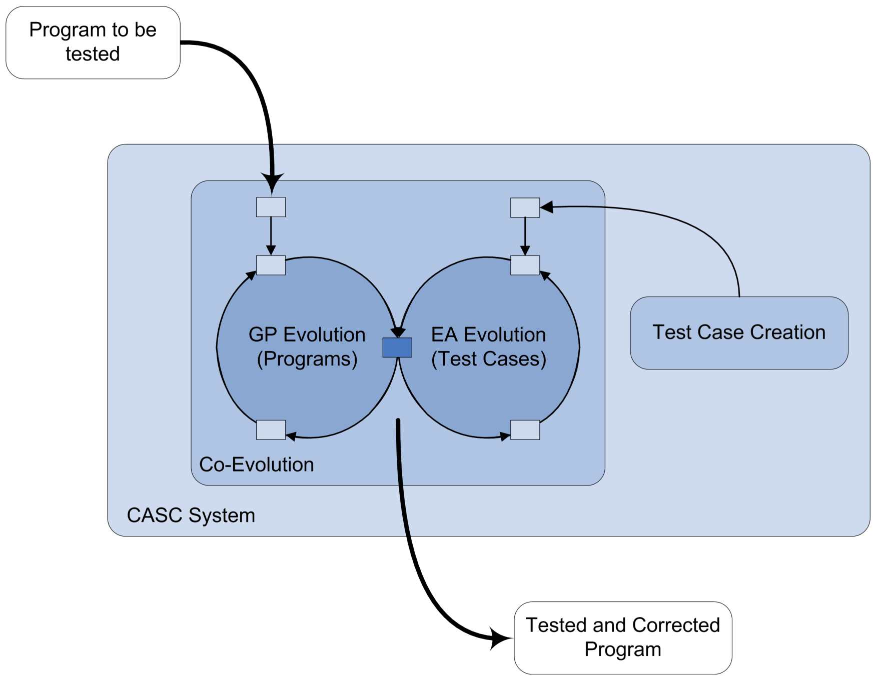


Figure 1.1: High Level Diagram of the CASC System

first time it has been extended to include automated software correction.

1.1. SOFTWARE ENGINEERING FOUNDATION

Software engineering can be defined as "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software" [36]. Software testing is a subset of the software engineering process that is focused on ensuring the correctness and completeness of a piece of software. There are many different types of formal software testing and each type has many different aspects. The general type of testing that the CASC system performs is black box functional testing [29]. Black box testing is a method of testing where the tester has no access to the internals of the program being tested, the only thing that can be seen is the input going in and output coming out. Functional testing is simply testing (at any level) for correct functionality of a program.

In the modern software development process, software testing is becoming increasingly important, particularly in financial terms. In 1978 Jones [10] estimated that catching an error in the system specification phase is approximately 50 times cheaper than it is to correct the error later in the system testing phase. In a 2002 news release [25] the National Institute of Standards and Technology (NIST) estimated that software errors cost the U.S. economy approximately $59.5 billion a year, which accounts for approximately 0.6 percent of the gross domestic product. In the referenced news release, the NIST states that "although all errors cannot be removed, more than a third of these costs, or an estimated $22.2 billion, could be eliminated by an improved testing infrastructure that enables earlier and more effective identification and removal of software defects". From this, it is clear that efficient and effective software testing and correction methods need to be developed in order to keep up with advancements in software development methods.

1.1.1. Manual Software Testing. The general process of testing a piece of software manually involves planning out the testing strategy to use on the software, developing the actual test cases to use, executing the tests, gathering the results and analyzing and interpreting them, and repeating the process for any bugs that were identified. Manual software testing is often performed by a team of testers who have been involved (at least to some degree) in the development process.

Most software testing performed today is done by human testers. The process of testing a piece of software is quite time consuming. This is problematic because there are many modern tools available to developers which makes them able to produce code more quickly and efficiently, which means that testers are being asked to test more and more code in less time. Because of this, testing is becoming a bottleneck in the software production process.

1.1.2. Automated Software Testing. Automated software testing is a growing field that is used to assist in the testing process. Typically, automated testing is used to generate test cases, execute the tests, and/or compile and analyze the results of tests [5]. The major drawback to automated software testing is that it is expensive to implement which makes it hard to justify in the short term. The number of tools and applications available to perform or assist in automated software testing is growing as demand/necessity for the methodology increases.

Automated software testing is particularly well suited for agile software development. Agile development is a method which develops software in a series of (relatively) short iterations [29]. Each iteration is focused on the development of a single component of the software project and is implemented as though the component is the entire project (i.e., each iteration implements the full development cycle for a component). This means that there are many testing phases for each project (one for each component), which is why automated testing lends itself well to this method. Automated testing is quicker than manual testing and is easier to justify since agile development yields more frequent testing phases.

In [4], the authors present a short survey of computational intelligence applied to software testing, specifically test case generation. The key to applying computational intelligence to test case generation is reducing the generation process to an optimization problem. Miller and Spooner were the first to do this in [24]. The method they presented involved setting the integers and conditional values in the program to arbitrary constants to drive the program down a pre-specified execution path, then various floating point inputs were provided as input to the program. Korel followed this idea up in [11] and [12] by actually executing the program being tested, whereas Miller and Spooner used symbolic execution. In Korel's implementation, if the execution follows the selected execution path at a branch point, then a 0 is assigned to that branch point, otherwise a positive integer is assigned. So by minimizing the assignments, an input can be selected which follows the selected execution path.

The survey in [4] also looks at evolutionary computation techniques which have been applied to test case generation. In [22] and [23] GADGET (Genetic Algorithm Data GEneration Tool) is presented by Michael et al. GADGET uses a method somewhat similar to Korel's except that instead of trying to find a single input which will match an execution path, a set of inputs is sought which will maximize execution path coverage in the program. This is accomplished using a genetic algorithm (a type of evolutionary computation, discussed further in Section 1.2.1) to evolve the inputs provided to the program. GADGET also supports random selection, gradient descent, and simulated annealing as other methods to generate test cases. The results shown in [23] are favorable, showing that the genetic algorithm slightly outperforms the other methods, which is a particularly interesting result since the CASC system employs a genetic algorithm to create test data.

The work of Pargas et al. is also briefly mentioned in the survey. In [26] Pargas et al. present another test data generation method involving a genetic algorithm called TGen. The TGen tool operates somewhat similarly to GADGET. TGen was compared to a random test data generator and performed quite well.

1.2. EVOLUTIONARY COMPUTATION FOUNDATION

Evolutionary Computation (EC) is a type of computational intelligence which is inspired by the biological mechanisms of evolution. EC is a broad field which encompasses many different varieties of EAs. A typical EA creates and evolves a population of potential solutions for a given problem. The EA population initially samples the problem space in a random fashion, but in successive iterations it becomes more directed in its search for a solution. EAs are effective for solving Combinatorial Optimization (CO) problems. CO problems usually have large problem spaces and are classified as NP-Hard [2] (a class of problems which are believed to not be efficiently solvable in general). EAs, however, typically have the innate ability of being able to navigate large problem spaces well; this is why EAs so readily apply to CO problems. CO problems can rise from many fields, such as mathematics, artificial intelligence, and software engineering, which makes EC applicable to a large set of typically hard problems.

As mentioned earlier, the field of EC encompasses many different algorithms that all follow the same general evolutionary model, despite the fact that many of these algorithms were developed independently of each other. Historically, the three major algorithm families that made up EC were genetic algorithms, evolutionary programming and evolutionary strategies. More recently, GP has also joined the EC field. The CASC system uses two historical EA flavors: genetic algorithms and GP; which are combined using co-evolution.

1.2.1. Genetic Algorithms. Genetic algorithms are one of the first EC methods conceived and are still some of the more commonly used EA's today. The concept of the Genetic Algorithm (GA) was popularized by John Holland in the 1970's, particularly in his book published in 1975 [9], which was focused on his studies of adaptive behavior. The canonical GA is an EA whose individuals (i.e., the members of the evolving population) are represented as fixed length binary bit strings (other representations have since been used), which favor crossover over mutation as the principle variation operator.

Since the early 1990's a push has been made to combine many of the original algorithms in the EC field into one unified EA model. A typical cycle in the unified EA closely follows the standard evolutionary cycle: initial creation (typically random, but possibly seeded) followed by a cycle of reproduction and mutation, evaluation, and competition. This cycle continues until a predetermined termination condition is reached, such as a set number of generations have passed or a goal fitness is reached in the population. The structure for an EA is laid out by the evolutionary mechanisms it is based on, the only part that is application specific and must be decided by the implementer is the representation for an individual and the fitness function used to determine how well an individual is performing. If a non-standard representation is used, then appropriate customized variation operators need to be defined.

In [19] researcher Timo Mantere discusses his research into the use of GAs to generate test data during software testing. Mantere presents the various successes he has had in applying a GA to automatically generate test data for applications in a series of seven publications detailing his research. While all of these publications are relevant to the CASC system, there are two in particular which not only automatically generate test data, but also use co-evolution to evolve program parameters. These papers are discussed further in Section 1.2.3.

1.2.2. Genetic and Evolutionary Programming. GP is a type of EA in which the individuals being evolved store trial solutions in a tree representation. As data structures, trees have a wide range of application; however, the application which is most relevant to the CASC system is that of evolving computer programs. In this implementation, each individual in a GP population will contain the parse tree for the program which it ultimately represents. The first reported results of GP were published by Steven Smith in 1980 [35]. In 1985 Nichael Cramer [3] also published results yielded by using GP techniques. Since the early 1990's, John Koza has done a lot to popularize GP, particularly through his classic four-book series on GP [13, 14, 15, 16]. William Langdon and Ricardo Poli are two researchers who also have contributed significantly to the field of GP. In [18] Langdon presents many new and emerging GP techniques are along with the original foundations of GP. Poli has also done a lot of work on parallel distributed GP [28], which is focused on the evolution of programs which can readily be parallelized.

The methods used by GP are very similar to that of a typical EA except the evolutionary operators have been modified to interact with tree structures. Reproduction is performed using a crossover method in which subtrees are interchanged between parents to create the offspring. This makes the reproduction operator a very pivotal part of the evolutionary process because even exchanging a single subtree in a program parse tree can greatly change the outcome of the program the parse tree represents.

Mutation is performed by replacing a randomly selected subtree in an individual with a randomly created subtree. This implies that the mutation operator must be aware of the subtree functionality as to maintain the integrity of the program, e.g., a subtree with a binary arithmetic operator as the root, such as the addition operator, can only mutate to another binary operator subtree, such as subtraction, multiplication, or division. Another option is to make the mutation operator capable of removing or supplying operands in the event that the arity of the node changes.

To evaluate a GP individual representing a computer program, first the individual's parse tree is pretty-printed into its program form. The program is then compiled, if necessary, and executed. The fitness is then determined based on the output of the program.

Historically, Evolutionary Programming (EP) was used by Lawrence Fogel in 1966 [7] to evolve finite state automata; however, its applications have widened since. In its current state, EP does not have any standard structure or representation. Some of the original variations made to the classical EP model are very similar to GP [6]. In these variants EP evolves expressions which can easily be transformed into computer programs, which makes the variants very similar to GP, but unlike GP only the numerical values are modified in the evolutionary process, the structure of the expression/program remains the same. This is the (temporary) method being employed by the CASC system to evolve the program population.

1.2.3. Co-Evolution. Co-evolution is an extension of the standard EA model where the fitness of an individual is dependent on other individuals in its own or other populations. This relationship can be categorized as either cooperative or competitive. In nature the relationship can take on many forms, for example any predator-prey or non-symbiotic parasite-host relationship represents a competitive (although necessary) mutual dependence. An example of a more cooperative relationship would be nectar seeking insects performing pollination for the plants which supply the nectar.

The CASC system uses competitive co-evolution between a population of evolving programs and a population of evolving test cases. This competition is intended to create a type of evolutionary arms race between the two populations. An evolutionary arms race works much like an actual arms race except it occurs on a genetic level. As the individuals in one population improve in fitness, pressure is placed on the individuals in the other population to improve as well. This process will continue and, if given enough time to evolve, each population will ideally be driven to perform as well as possible. The concept of an evolutionary arms race is not a new one. Christopher Rosin [31, 32] performed extensive research on methods for competitive co-evolution, examining the parasite-host relationship which yields the evolutionary arms race.

The co-evolutionary method has a unique set of problems which can arise during the evolutionary process. These problems are inherently hard to detect, and as such much work has been done in finding ways to pre-emptively counter these problems. One possible problem that can occur in the co-evolutionary process is the phenomenon known as evolutionary cycling. Evolutionary cycling is basically the evolutionary version of rock-paper-scissors, i.e., the genetic configurations of the populations cycle back on themselves and do not advance past a certain point. This phenomenon is hard to detect because the cycle can involve hundreds of states. In [30] Rosin introduced the concept of an evolutionary history or hall of fame. The main purpose of the evolutionary hall of fame is to counter evolutionary cycling. The hall of fame works by storing the best individuals of every generation, then the individuals in following generations compete against individuals sampled from the other population(s) as well as from the hall of fame. So, to perform well an individual must outperform both the current generation's best individuals as well as the best ancestral individuals, which ideally will disallow cycling. This method is used in modern co-evolutionary systems to not only counter the possibility of evolutionary cycling, but to also speed up (and generally improve) the evolutionary process.

Another problem which can arise during the co-evolutionary process is evolutionary equilibrium. This is where the evolving populations come to a point where they are content with their performance against the other population(s). This contentness causes the evolution to fail in that the populations are no longer pushing each other to improve. John Cartlidge is a researcher who has put considerable work into addressing the potential problems in co-evolution [1], and this problem is one which he addressed. The solution Cartlidge presents is to temporarily remove some of the better individuals from one population causing the equilibrium to be lost, which in turn would prompt the populations to start evolution again.

The third significant potential problem that can arise during co-evolution is disengagement. In this case, one population evolves so much faster than the other that all individuals of the other are utterly defeated, making it impossible to differentiate between better and worse individuals without which there can be no evolution. To counter disengagement Cartlidge uses his "reduced virulence" method to inhibit the development of the excelling population, allowing the other population(s) to catch up in terms of performance.

As it is so difficult to detect if and when the co-evolutionary problems are actually occurring, it is typically better to use a pre-emptive strategy to counter these problems. While the CASC prototype does not have any of the aforementioned counter measures installed, the coming versions of the system are planned to have at least some, if not all, of these counter measures implemented.

1.3. EVOLUTIONARY COMPUTATION APPLIED TO SOFTWARE ENGINEERING

Most of the current work in applying EC to software engineering is focused on automating the testing process using EC. Testing of programs written using the object oriented programming paradigm is currently a popular area of research. In [17, 38, 39] Wappler (et al.) discusses various approaches to evolutionary unit testing of object-oriented software, such as the use of strongly-typed genetic programming methods in the evolutionary process or white box testing for the testing method. In [37] Tonella discusses the use of an EA to perform the unit testing of objects (e.g., classes). Tonella's primary motivation was to test how an object performed given varying sequences of invocations of the object's methods.

In [40] Wappler discusses many recent results which show Particle Swarm Optimization (PSO, a close relative of EC) outperforming both general and problem specific genetic algorithms as a test case generation technique. Since the test cases in the CASC system are managed by an EA this result is particularly interesting. Wappler found that for their application (testing of objects using EC) PSO outperformed an EA in terms of both effectiveness and efficiency. This is discussed further as an option for the CASC system in the Future Work section.

In [20] and [21] Mantere introduces a software testing method similar to that of the CASC prototype. Mantere uses a two-population co-evolutionary system in which one population is a set of test cases for an application and the other is a set of various values for the parameters to be provided to the application being tested. Both populations were evolved using a GA. Mantere discovered through this research that the more control the GA had over the program parameters, (i.e., the more parameters being evolved) the more positive the results were. This is an important result for the CASC prototype (and, ideally, the envisioned CASC system), as it implies that the prototype's method of dynamically finding all the evolvable sections of code it can in a program should make the results as positive as possible. For details on the CASC prototype implementation, see Section 2.

2. CASC SYSTEM OVERVIEW AND DESIGN

The CASC system combines the concepts of GP (see Section 1.2.2) and competitive co-evolution (see Section 1.2.3). There are two populations being evolved: a population of software artifacts (currently C++ programs) evolving using GP and a population of test cases (i.e., program inputs) evolving using a test case appropriate EA. The fitness for each software artifact is determined by how well it performs against a set of test cases. Similarly, the fitness of a test case is determined by how well it performs against a set of software artifacts. Since each population is attempting to optimize these fitness values, an evolutionary arms race results.

The CASC system utilizes a two-population competitive co-evolutionary cycle, which is basically two overlaid evolutionary cycles intersecting at the point of fitness evaluation; Figure 2.1 depicts the co-evolutionary cycle used by the CASC system. The general flow of the system is as follows:

1. System Initialization: Prepare the system (i.e., initialize data structures, read in configuration settings, distribute relevant settings to system modules accordingly, etc.).

2. Population Initialization: The two populations (program and test case) are initialized/created.

3. Initial Evaluation: All of the individuals in the populations are evaluated and assigned fitness initial values.

4. Reproductive Phase: Parents are selected, crossover is performed creating new individuals, mutation is applied (if necessary) to a subset of the new individuals, and the individuals are entered into the general population.

5. Evaluation: Evaluate all individuals and assign appropriate fitness values.

6. Competition and Termination: Poorly performing individuals are selected (using modified tournament selection) and removed from their population.

7. Check Exit Conditions: If any exit conditions are satisfied exit the evolutionary cycle, otherwise go to step 4.

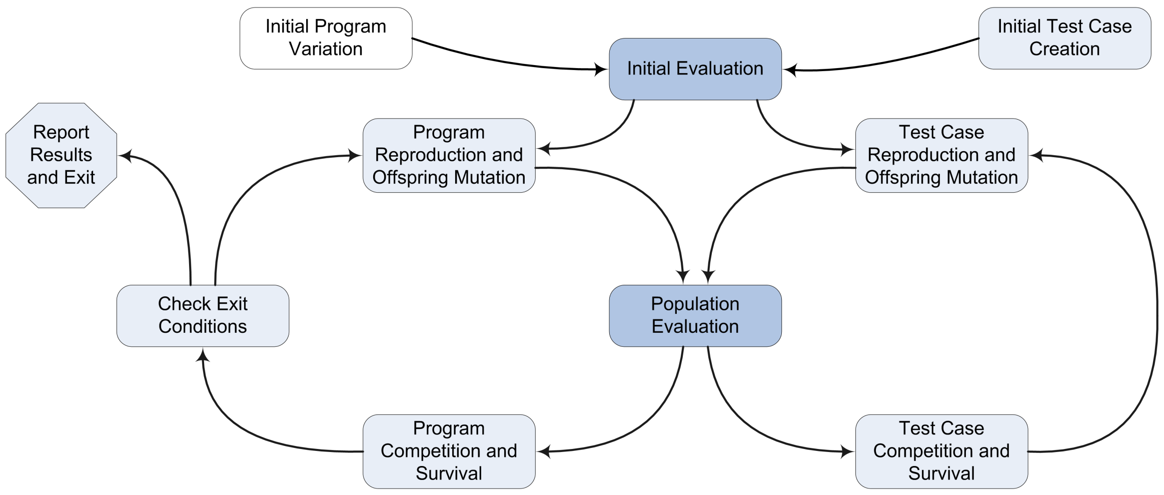

Initial Program Variation
Initial Evaluation
Initial Test Case Creation
Report Results and Exit
Program Reproduction and Offspring Mutation
Test Case Reproduction and Offspring Mutation
Check Exit Conditions
Population Evaluation
Program Competition and Survival
Test Case Competition and Survival

Figure 2.1: The CASC Co-Evolutionary Cycle

In the system flow enumeration, steps 1-3 serve as a primer for the system and are performed only once. Steps 4-7 are the actual evolutionary cycle used by populations. Each population has its own evolutionary cycle that is separate from the other population except for one point, the evaluation phase. These phases are all described in more detail in the following sections.

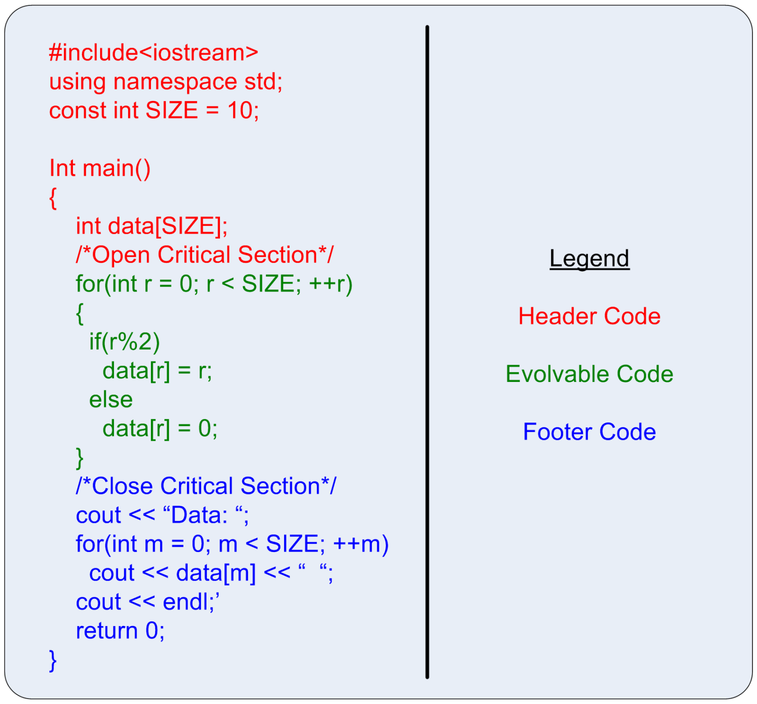
2.1. CASC SYSTEM INITIALIZATION

Before the CASC system can be used it must be initialized. The primary purpose of the system initialization is to read in the configuration values from the CASC configuration file. The configuration file specifies how the system is to perform the evolution, where it can find the program to test and correct, the population sizes, exit conditions, etc. While some of the configuration parameters have default values specified within the system (and as a result specifying them is optional), the majority of the parameters must be specified, otherwise the system cannot (and will not) function. See Appendix A for an example CASC configuration file.

2.2. POPULATION INITIALIZATION

Population initialization is performed differently for each population. The program population is based off a seed program (i.e., the program to correct) which is read in from a source file and transformed into an evolvable tree, whereas the test cases are generated randomly.

2.2.1. Program Initialization. The entire program population is based on a single seed program that is provided to the system. Currently the program must be written in C++ and the section of code to evolve must be enclosed by two comment statements, designating the code block as evolvable code to the system (this code section will be referred to as the evolvable section of the seed program). The format of the comment statements as well as the actual comment is configurable via the CASC configuration file.



```
#include<iostream>
using namespace std;
const int SIZE = 10;

Int main()
{
    int data[SIZE];
    /*Open Critical Section*/
    for(int r = 0; r < SIZE; ++r)
    {
      if(r%2)
        data[r] = r;
      else
        data[r] = 0;
    }
    /*Close Critical Section*/
    cout << “Data: “;
    for(int m = 0; m < SIZE; ++m)
      cout << data[m] << “  “;
    cout << endl;’
    return 0;
}
```

Figure 2.2: Valid Seed Program

Figure 2.2 contains an example of a simple seed program; in this program the tags to open and close the evolvable section are */*Open Critical Section*/* and */*Close Critical Section*/*, respectively. The code above and below the evolvable section is stored as the common program header and footer shared by all programs in the CASC program population. The header and footer sections are stored and used verbatim, no evolution is performed on these sections.

The seed program is read in and processed by the CASC parser. The code in the seed file is read in line by line and stored as the common program header until the evolvable section is encountered. The code in the evolvable section is read in and put into a separate, temporary file. Once the end of the evolvable section is encountered, the temporary file is closed and the remaining seed code is read in and stored as the common program footer. At this point the CASC parser re-opens the temporary file and parses the evolvable section. This parsing yields a lightweight parse tree representing the evolvable section of the seed program, i.e., the first individual of the program population. In an EA it is typically desirable to have the initial population(s) be as diverse as possible, e.g., generated randomly. The CASC system is to evolve a specified program, which makes a fully random initial population not possible. So, initial population diversity is achieved by first making a clone of the seed individual and then performing a modified mutation phase on the clone; this phase will be referred to as the initial variation (IV) phase. The IV phase is very similar to the mutation phase (which will be discussed in detail in Section 2.3.2). The principle difference between the two phases is that the amount that a given individual can change in the IV phase is determined randomly from a gaussian distribution, whereas this amount is static in the mutation phase. On average, this will produce a fair amount of moderately modified programs and a small amount of drastically modified programs.

2.2.2. Parsing in the CASC System. EAs are typically very computationally intensive processes. As with most systems which employ EAs, the computational complexity of the CASC system is dominated by its evolutionary aspect. For this reason every effort was made to reduce the computational complexity of the evolutionary processes used. A large focus was put on minimizing the overall complexity of the program Abstract Syntax Trees (ASTs, essentially a parse tree) that are used by the CASC system. The ideal structure for the program AST would be something that is lightweight and easy to traverse and manipulate.

The CASC system employs a parser generated by the ANTLR system [27] as a front end to read in and parse the seed program. Basically, ANTLR is a parser

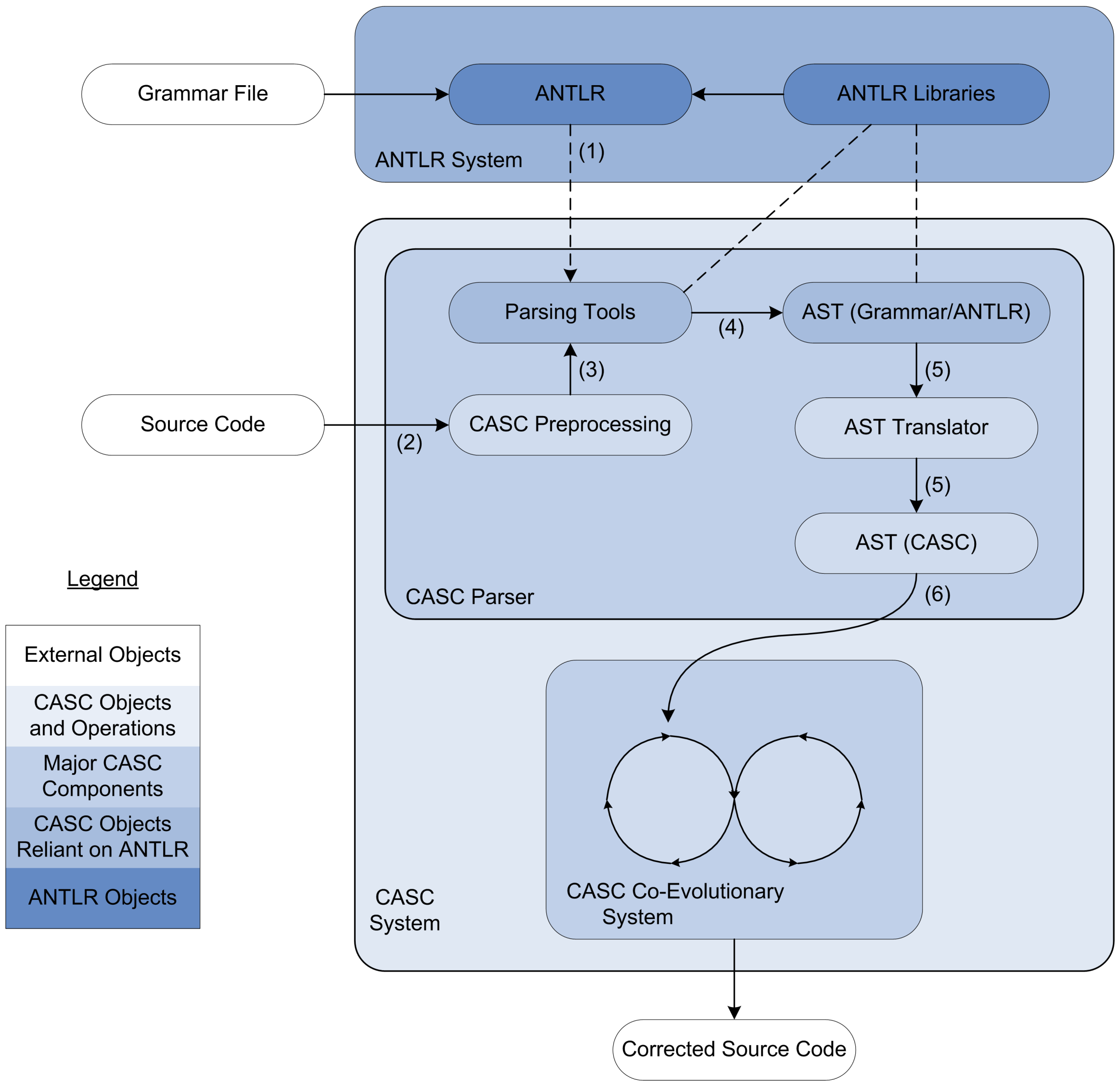


Figure 2.3: The CASC Parser

generator. There are many capable parser generators available. ANTLR was chosen due to its high degree of functionality and general usability. ANTLR works by taking in a file containing the specifications for a language grammar and based off of the grammar rules in the file a parser is produced. The output parser can be written in a number of common programming languages. The CASC system itself is written in C++ so that is the language that was used for the parser (i.e., a parser which parses C++ and is written in C++). The ANTLR library of objects and functions is used as a backbone for the code generated from the grammar file. The output produced by the parser (that is useful to the CASC system) is an AST. The form that the AST takes (i.e., how the parse tree(s) are actually constructed) completely depends on how the grammar is defined in the original grammar file; so for a given language there could be many possible AST representations that are all correct.

The ANTLR system is used initially to create the parsing tools necessary to parse a given language. These tools (and any output from them) are derived from the various base objects in the ANTLR libraries; so any AST yielded from parsing would be reliant on the ANTLR libraries. Also, the ANTLR data structures used to hold the AST's are large complex objects derived from multiple base classes in the ANTLR libraries; these objects contain data and functionality that is largely not useful to the CASC system. For these reasons the actual AST's produced by ANTLR are not used directly during evolution, instead trimmed down versions of the AST's are created by analyzing the ANTLR output AST's. The trimmed down trees are very lightweight and are not reliant on the ANTLR system. These reduced versions of the AST's are then passed on to the CASC system to be used in evolution. Of course, this means that each new grammar that is used by the system needs to have an associated AST translator either developed or provided. This may seem like a drawback of the system; however, the alternative (of using the raw AST produced from the parser) would most likely require that the grammar file itself be modified to make the AST match what is expected in the CASC system, which in many cases is expected to be more time consuming that writing an appropriate AST translator.

After the parsing tools have been created (step (1) in Figure 2.3), a library is created that serves as the front end parsing for the CASC system. In a typical run, a source code file will first be provided to the system (2). Next, the source code file is preprocessed, during which the code sections to be evolved are picked out and provided to the (ANTLR) parsing tools (3), which produces the ANTLR AST (4). The AST produced by the parsing is then translated into a light-weight AST (5). Lastly, this AST is provided to the CASC co-evolutionary system (6), which then takes over.

The CASC AST deviates in some ways from the generally accepted idea of how an AST/parse tree is represented. The main deviation is in how statements (conditionals and loops) are represented. Typically these are represented with the body of the statement attached to the root node, e.g., an *if* statement would have one child for the logic statement, a second child for the *then* section, and a third child for the *else* section (if necessary). Instead, the CASC system gives both the statement and the line(s) of code in the body of that statement their own trees (see Figure 2.4 on page 20 and Appendix B for examples of CASC ASTs). Bodies of statements are kept attached to the statements by keeping track of the scope level that each tree belongs in. This implementation was used to decrease the access time of the bodies of statements and to make it easier to separate statement trees from operator/expression trees. Neither of these benefits are realized in the CASC prototype. The former benefit will become apparent for larger programs with many nested statements, while the latter is anticipated to be useful once GP has been implemented in the system.

Currently the CASC system supports a basic/foundational subset of the C++ language. The parsing tools created by the ANTLR grammar file can handle the majority of the C++ language; however, the AST translator only handles code pieces relevant to the code that CASC has been evolving so far. In its current state the AST translator supports all common operators, looping statements (*for*, *while*, *do-while*), decision branching (*if*, *if-else*), numeric constants, primitive numeric variables (and arrays), and function invocations. Updating the amount of the C++ language supported by CASC is a matter of updating the AST translator (assuming the ANTLR parsing tools can handle the new code structure(s)) and updating the CASC pretty printer that converts the trees back into code.

2.2.3. Test Case Initialization.

Test case initialization is much simpler than program initialization. The input for most programs can be expressed as one or more lists of values of some type. This is exactly what a general test case is defined as in the CASC system. The values for a test case are randomly generated and assigned to an individual. Obviously different programs will require different types of input, so there is an aspect to the test case generation that is problem specific and must be specified for each new program. Along with the type of values, there may be problem specific limitations on the actual values themselves. In simpler cases this can be specified in the CASC configuration file, however, for more complicated limitations some problem specific code enforcing the limitations in question will need to be written.

2.3. REPRODUCTION

Reproduction in both populations is performed using a tournament selection method to determine the individuals to use as parents and a form of biased crossover (in which the more fit parent will be favored to supply a gene) to produce the actual offspring. The selection and crossover algorithms used are shown in Algorithm 1 and Algorithm 2, respectively. After the specified number of offspring have been produced it is then determined which offspring (if any) should be subjected to mutation. Lastly, the offspring are introduced into the general population.

Algorithm 1 Parent Selection

```
//Parent 1 Selection
for $i = 0$ to $TOURN\_SET\_SIZE$ do
  $TournSet1[i] \leftarrow$ Unique Random Individual ID
end for
$Parent1 \leftarrow$ ID of Most Fit Individual Represented in $TournSet1$

//Parent 2 Selection
for $i = 0$ to $TOURN\_SET\_SIZE$ do
  $TournSet2[i] \leftarrow$ Unique Random Individual ID
  while $TournSet2[i] = Parent1$ do
    $TournSet2[i] \leftarrow$ Unique Random Individual ID
  end while
end for
$Parent2 \leftarrow$ ID of Most Fit Individual Represented in $TournSet2$
```

2.3.1. Program Reproduction.

As was mentioned previously, the CASC prototype utilizes EP for the program evolution. In EP, the variables and numeric values in a program evolve while the general program structure remains the same across all individuals. The way it does this is by keeping the root nodes of the program parse trees the same for all program individuals. This guarantees, for example, that a loop tree will not transform to an assignment operator during crossover or mutation.

Algorithm 2 Biased Crossover

```
$MParent \leftarrow$ More fit parent
$LParent \leftarrow$ Less fit parent
$Offspring \leftarrow$ Clone of MParent
for $i = 0$ to $NUM\_GENES$ do
    $Roll \leftarrow$ Random Value Between 0 and 100
    if $Roll < (50 - BIAS\_AMOUNT)$ then
        $Offspring.gene[i] \leftarrow LParent.gene[i]$
    end if
end for
```

This method is used as a stepping stone towards the ultimate goal of using GP to evolve the programs (i.e., evolution of code structure as well as variables and numeric values).

Program structure is maintained during reproduction by keeping the parse tree root nodes in the offspring the same as the parents. Crossover is performed by using the children of the root nodes as the genes which are being crossed over. Each offspring starts out as a clone of the more fit parent. For every root in the offspring the child nodes are iterated through. A random roll is made for each child to see if that child (and the subtree it is the root of) will be replaced by the respective subtree in the less fit parent. See Algorithm 2 for the general algorithm for this process; also Figure 2.4 gives a visual example of program crossover being performed (note: in this example assume parent 1 is more fit than parent 2).

2.3.2. Program Mutation. After an offspring is created the system then determines if the offspring will be mutated and, if so, by what degree; both the chance for mutation and the amount of mutation are specified in the CASC configuration file. Not every node in the parse tree is considered for mutation, though (this also helps limit the problem space to a manageable size). Instead only very specific nodes are considered. Performing mutation in this way allows the system to restrict mutation to points in the program which affect its output the most while still maintaining the general functionality of the program. This method also provides a way to get around the question of how to mutate nodes which the concept of mutation does not readily apply to (e.g., 'else' nodes, '=' nodes, 'do' nodes, etc.). Algorithm 3 shows how the

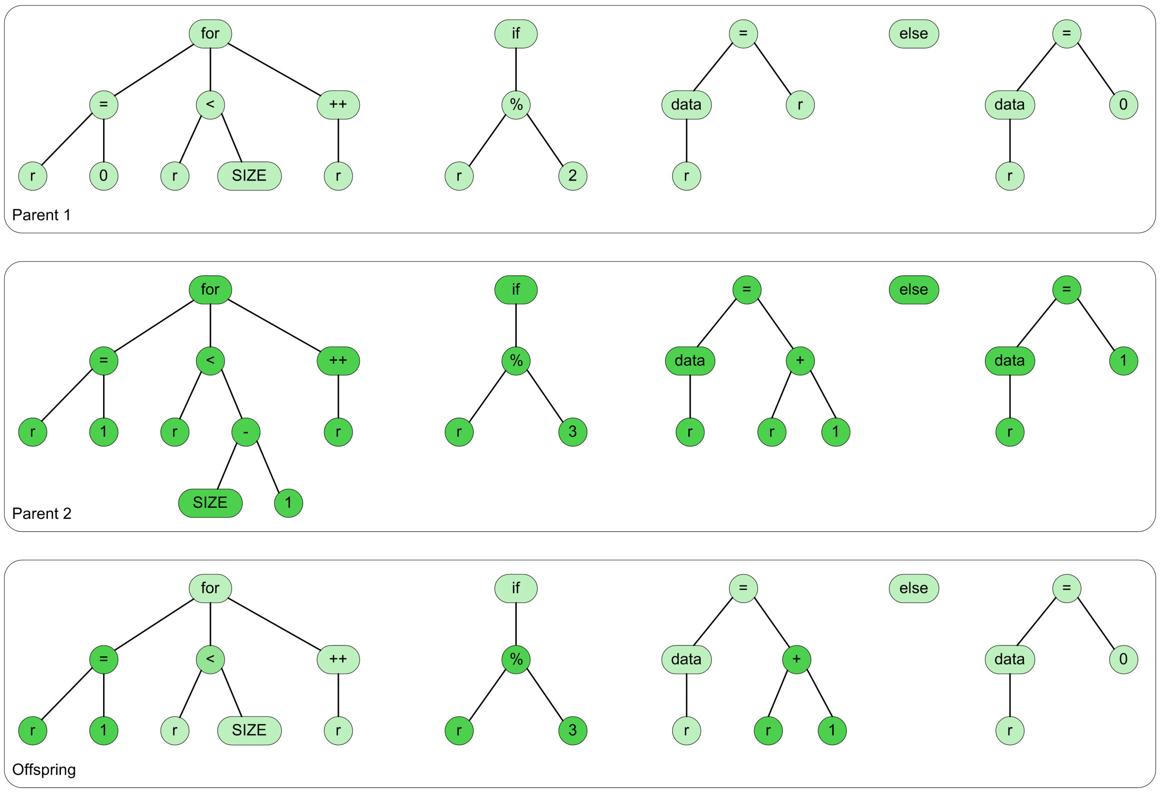


Figure 2.4: Program Crossover

CASC prototype determines if mutation should occur and, if so, to which control points. An example of program mutation is shown in Figure 2.5 (page 22).

First, for each offspring individual the system randomly determines if mutation should be performed (the chance of mutation occurring is specified in the configuration file). If an individual is to be mutated, the system then analyzes the parse tree(s) in the individual program, looking for points that are valid for mutation (these points will be referred to as critical points). Critical points are defined as any numeric constants and any variables in arithmetic expressions not already involving a numeric constant. The reason for considering this subset of variables is that once an unmodified variable (i.e., a variable not involved in an arithmetic expression) is mutated, the variable is modified by adding or subtracting some amount from it and this amount is expressed as a numeric constant, which is valid for mutation; so instead of mutating an already modified variable, the numeric constant modifying the variable is mutated. This method of expression modification is a naive method employed to

Algorithm 3 Program Mutation

```
$R \leftarrow$ The set of all offspring for the current generation
for $i = 0$ to $|R|$ do
  $Roll \leftarrow$ Random number between 0 and 100
  if $Roll < MUTATIVE\_CHANCE$ then
    $CP \leftarrow$ Node IDs for control points in $R[i]$
    $NumChange \leftarrow |CP| \cdot MUTATIVE\_PROPORTION$
    for $j = 0$ to $NumChange$ do
      $ToChange \leftarrow$ Unique random value between 0 and $|CP|$
      Mutate $CP[ToChange]$ in $R[i]$
    end for
  end if
end for
```

test the CASC prototype's functionality on the simple programs used for the preliminary experiments. A more sophisticated mutation method will come with further iterations of the system.

After all of the critical points have been determined it is then decided which points are to actually be mutated. This is done by randomly selecting points from the total set of critical points. A critical point is mutated by adding or subtracting (there is an equal chance of either occurring) some value to or from the value of the node. If the point is a variable, then the variable node is replaced by a + or - node, of which the variable is one child of and the numeric constant being added or subtracted is the other. This arithmetic operation is guaranteed to be legal since the CASC prototype only employs variables that are of numeric primitive types.

If the control point to be mutated is a numeric constant, then the value of the node is modified by changing the value to be anywhere from -150% to 150% of the current value. The amount of actual change is pulled from a gaussian distribution, implying that the majority of changes will be moderate and a small number of the changes will be drastic.

Once the specified number of critical points have been mutated, the process is repeated for the next offspring to be mutated.

2.3.3. Test Case Reproduction. At the low level, test case reproduction is implementation specific; however, from a high level perspective the general concept is

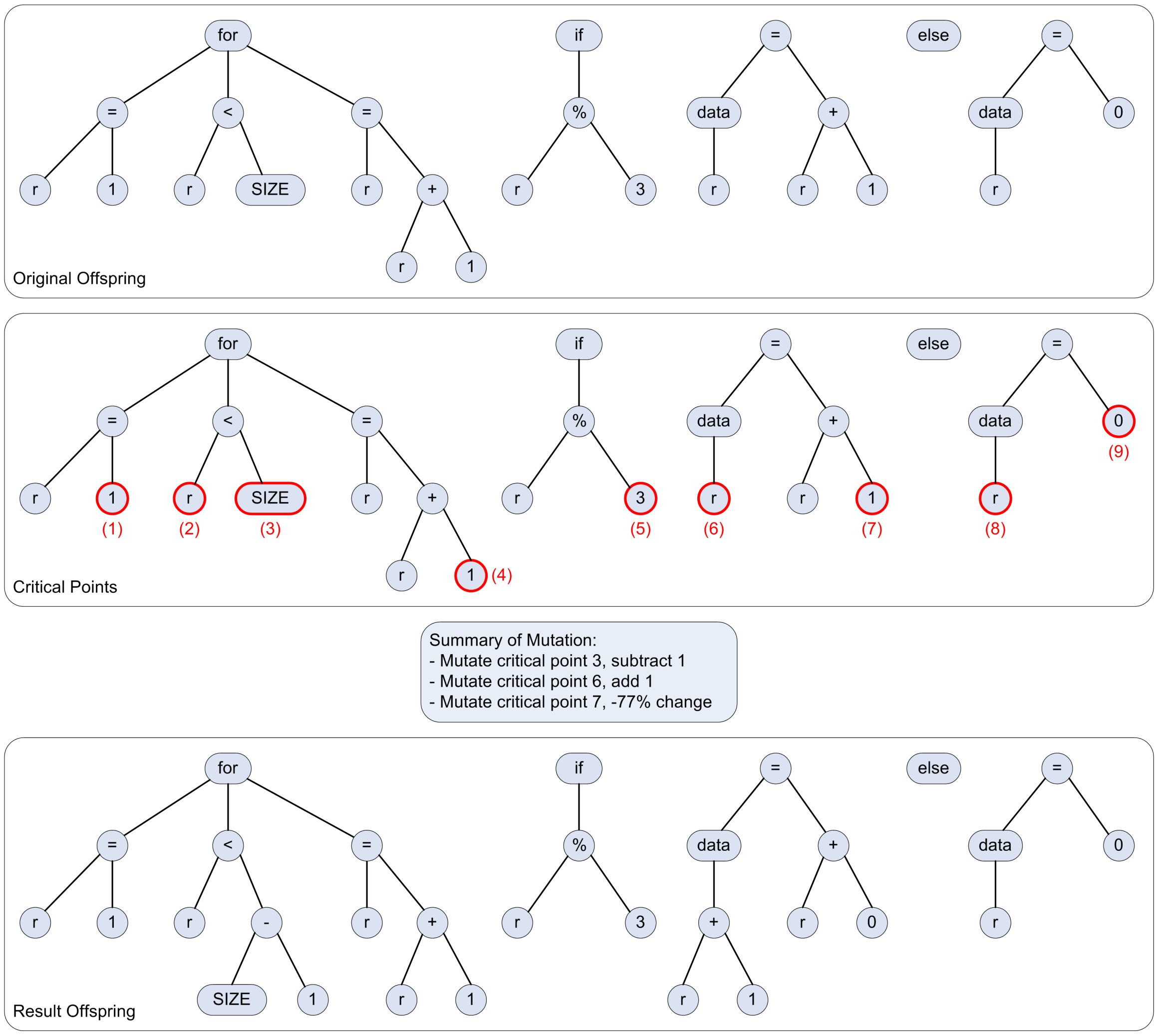


Figure 2.5: Example Program Mutation

the same regardless of implementation. First, parents are selected using the selection algorithm shown in Algorithm 1. Next, the biased crossover algorithm shown in Algorithm 2 is used to create the offspring, where the genes are the values in the test case. Both selection and crossover follow the algorithms in the referenced figures closely. This process can easily be extended if and when more complicated test cases are required.

2.3.4. Test Case Mutation. Like test case reproduction, the low level details of test case mutation are implementation specific, but the high level process is the same regardless of implementation. For each offspring the system randomly determines if mutation will be performed (the chance of mutation occurring is specified in the CASC configuration file, separate from the mutative chance in programs). For each test case that is to be mutated, values are randomly picked from the test case and modified according to the implementation. This process is repeated until the specified number of values have been modified, then the mutation is complete.

2.4. FITNESS EVALUATION

The evaluation phase is the point where the two evolutionary cycles meet and the populations interact. It is assumed that for the evolving programs, each set of inputs will map to one and only one output (i.e., the programs being evolved are deterministic). So, since it is very possible that a particular program and a particular test case be paired up more than once in their lifetimes, the results of each pairing are stored in a hash table in order to make sure that no program is executed twice with the same input and to make previously obtained results efficiently available.

Both populations follow the same general algorithm for performing fitness evaluation. For each individual in the evaluating population the following steps are performed. First a set of (unique) opponents to test the individual against is selected from the opponent population. For each opponent selected, the hash table is checked to see if the program-test case pairing has been done before; if it has, then the result is retrieved and the opponent is removed from the opponent set.

Next, for each opponent still in the opponent set, program execution occurs. If a program is being evaluated, then it is first compiled (if necessary) and then, if the compile was successful, run with the various opponent test cases as inputs. If the program fails to compile, then the evaluation is complete and the program is assigned an arbitrarily low fitness. If the program execution takes longer than allowed (by the configuration file) then it is considered to time-out and the trial is given a low fitness.

If a test case is being evaluated, then, for each program in the opponent set, the program is compiled (if necessary) and executed with the test case as input. If any of the programs fail to compile, then that trial is not considered in the test case fitness calculation; if a program times-out the trial is still considered legitimate.

Often it makes most sense for the test case fitness to be the direct inverse of the program fitness, in which case assigning the test case fitness is trivial. However, in some cases it may be more appropriate to, for instance, reward a test case for exposing a unique error or execution path in a program, in which case the program and test case fitnesses are related, but not a direct inverse of each other. For this reason, the test case fitness design is problem specific; the fitness function used in the prototype experiments is described in detail in Section 3.2.2.

After all executions have been completed the outputs are analyzed according to the implementation specific fitness function and a fitness is applied for each trial. The pairings and results are then hashed. Lastly, the fitness for the evaluating individual is determined as the average fitness across all trials.

As with most programs that employ an EA, execution of the CASC system is a very time intensive task. The evaluation phase is by far the most time consuming phase for the CASC system. In order to reduce the amount of time needed to perform the evaluations, the CASC system uses distributed computing to spread out its work load. In its current state the CASC system can only work in a cluster environment that supports the MPI (Message Passing Interface) system [8].

The CASC system uses a master-slave topology to perform evaluations. The main node (the master) determines what program executions need to be performed for a given evaluation and then writes commands detailing the program to run and the data to use as input out to command files, creating one command file for each worker (slave) node. Once a command file is completely written the worker which the command file belongs to is woken up. The worker then reads in the commands from the main node and begins to run the specified program(s), gathering fitness values as the trials complete. After all the workers have been sent off to work the main node prepares to receive the fitness values for the trials from the workers. After all the trials have been accounted for the overall fitness is calculated and the workers are put back to sleep. The distributed fitness evaluation algorithm used in the CASC system is shown in Algorithm 4.

2.5. SURVIVAL SELECTION

Survival selection is performed using the same algorithm for both populations. The only factor considered when deciding who to terminate is fitness, i.e., the new individuals in the populations are just as likely to be removed as the older individuals.

Algorithm 4 Distributed Fitness Evaluation

```
$P \leftarrow$ Population being evaluated
$R \leftarrow$ Opponent population

for $i = 0$ to $|P|$ do
  // Determine Evaluation Set
  for $j = 0$ to $EVAL\_SET\_SIZE$ do
    $t \leftarrow$ Unique opponent ID
    if $Hashed(P[i], R[t])$ then
      Retrieve old result for $P[i]$, $R[t]$ pairing and store it
    else
      $EvalSet.push(t)$
    end if
  end for

  // Fill Worker Command Files and Wake Workers
  $Load \leftarrow NUM\_WORKERS \ / \ |EvalSet|$
  $Extra \leftarrow NUM\_WORKERS \ \% \ |EvalSet|$
  for $k = 0$ to $NUM\_WORKERS$ do
    Write $< Load >$ executions to worker $k$ command file
    if $Extra \neq 0$ then
      Write one more execution to worker $k$ command file
      Decrement $Extra$
    end if
    Wake worker $k$
  end for

  // Gather Results
  Receive results from workers
  Hash results
  Calculate and apply fitness to $P[i]$
end for
```

For each population a reverse version of tournament selection is performed to determine what individuals to remove. A group of individuals is randomly selected from a population and the individuals with the lowest fitness in the group are removed. If the size of the tournament set is *s* (for *s* greater than zero and less than the current population size) then this method guarantees that the top *s-1* individuals in the population are safe from being terminated, which in most cases is desirable. This process is repeated for each population until the population sizes are what was originally specified. This algorithm is shown in Algorithm 5.

Algorithm 5 Competition

```
$P \leftarrow$ Population Competing

while $|P| > MAX\_POP\_SIZE$ do
  for $i = 0$ to $COMPETE\_TOURN\_SIZE$ do
    $TournSet[i] \leftarrow$ Unique random individual ID
  end for
  $rem \leftarrow$ ID of least fit individual represented in $TournSet$
  $Remove(P[rem])$
end while
```

2.6. TERMINATION CRITERION

At the end of every generation the CASC system checks to see if any of the termination criterion have been met before continuing on. The CASC system currently utilizes two termination criteria. First, if a specified fitness value is either met or exceeded by a program individual, then the run is considered a success and the system terminates. Alternatively, if a specified number of generations is completed without finding the goal fitness the the run ends and the best program found is presented as the result. If neither of the termination criterion are satisfied, then the statistics on the generation are output and the evolutionary cycles start all over again with the reproductive phase.

3. EXPERIMENTAL SETUP

The experiments performed on the CASC prototype focused on having the system correct a sorting implementation, specifically insertion sort.

3.1. INSERTION SORT

Insertion sort is a simple sorting algorithm which divides the input data into two sets, a sorted set and an unsorted set. The sorted set initially consists of only a single element but with each iteration of the algorithm an element from the unsorted list is inserted into the sorted list in the position it belongs relative to the other element(s) of the list. This is continued until the unsorted list is empty and the entire data set is sorted. Figure 3.1 shows a (correct) C++ implementation of insertion sort. The bugged implementations that were used in the experiments were based off of this implementation. The actual seed programs used in the experiments are discussed further in Section 3.3.

```
for(i = 0; i < SIZE; i = i + 1)
{
    for(j = i; j > 0 && data[j] < data[j-1]; j = j - 1)
        Swap(data[j], data[j-1]);
}
```

Figure 3.1: Correct C++ Insertion Sort Implementation

3.2. IMPLEMENTATION SPECIFIC COMPONENTS

For each new program to correct there are implementation specific details about the program that need to be addressed. The first of which is how fitness calculation should be performed based on the output of the program. Of course, the insertion sort algorithm has a well known solution which could easily be used to determine if the output from a trial was correct; however, if the CASC system were to ever be used in a practical situation a known solution would likely not exist for the program being evolved. So in order to emulate this the fitness calculation was made completely independent of the known solution.

The second implementation specific portion of the CASC system is the representation of a test case. Obviously, different programs require different input. The CASC system needs to know how the inputs need to be formatted and how to present them to the program being evolved. The design of the CASC system test cases was made with this in mind, in an attempt to streamline the process of specifying new test case types as much as possible.

3.2.1. Test Case Representation. For the insertion sort programs a test case was a set of values to be sorted. The representation for this was a single array of values. The length of the array was constant and the values themselves were generated randomly but limited by bounds specified in the configuration file.

3.2.2. Fitness Calculation. The goal of the algorithm used to calculate fitness for the insertion sort trials is to basically tell how sorted the output data is. Naturally, an array of values is either in sorted order or it is not, but such binary judgment is of no use to the CASC system. In order for the evolutionary process to work, the fitness must be graduated; so a scoring function is used to determine how sorted each output actually is. The optimal case obviously being having the data sorted in correct order and the worst case being the data in reverse order.

For each element x the elements before and after x are inspected. A penalty is incurred for each element a that came after x such that $x > a$ and each each element b that came before x such that $x < b$. Also, if an element is lost in the process (i.e., if the output data does not have all of the input data) a heavy penalty is incurred. The algorithm for this scoring function is shown in Algorithm 6 and Figure 3.2 shows an example fitness calculation.

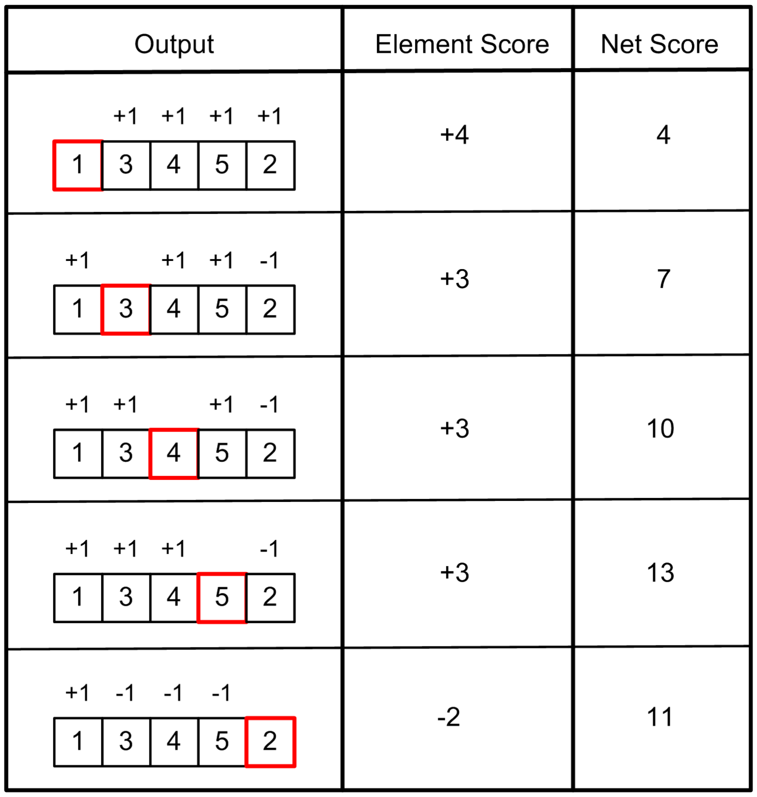
The score for an individual can range from $-(n^2 - n)$ to $n^2 - n$. Since the size of the data arrays can vary between runs the last calculation in Figure 3.2 is used to normalize the score to be between zero and one. This normalization is done just to make the analysis of results easier and has no effect on the evolution.

For these experiments the program fitness function is defined as identical to the output of the scoring function, while the test case fitness function is defined as one

Algorithm 6 Scoring Function for Sorting Implementation

```
$Score \leftarrow 0$
$input \leftarrow$ Data originally provided to the program
for $i = 1$ to $n$ do
    //Check if the data element is in the input
    for $m = 1$ to $|Input|$ do
        if $output[i] == input[m]$ then
            Mark $output[i]$ as found
            Remove $input[m]$ from the input array
            Exit the for loop
        end if
    end for

    // Only score values that were in the input
    if $output[i]$ is marked as found then
        // Look at values before the current value
        for $j = 1$ to $i - 1$ do
            // If preceding value greater than current then penalize, otherwise reward
            if $output[i] < ouput[j]$ then
                Decrement $Score$
            else
                Increment $Score$
            end if
        end for

        // Look at values after the current value
        for $k = i + 1$ to n do
            // If value less than current then penalize, otherwise reward
            if $output[i] > ouput[k]$ then
                Decrement $Score$
            else
                Increment $Score$
            end if
        end for
    end if
end for

// Calculate Fitness
$fitness \leftarrow (Score + n^2 - n)/(2 \cdot (n^2 - n))$

// Penalize for any elements that weren't found
for $w = 1$ to $|input|$ do
    Apply $LOST\_ELM\_PENALTY$ to $fitness$
end for
```

minus the output of the scoring function.



Output	Element Score	Net Score
+1 +1 +1 +1 $\boxed{1}$ 3 4 5 2	+4	4
+1 +1 +1 -1 1 $\boxed{3}$ 4 5 2	+3	7
+1 +1 +1 -1 1 3 $\boxed{4}$ 5 2	+3	10
+1 +1 +1 -1 1 3 4 $\boxed{5}$ 2	+3	13
+1 -1 -1 -1 1 3 4 5 $\boxed{2}$	-2	11

$$Fitness = \frac{NetScore+n^2-n}{2(n^2-n)} = \frac{11+25-5}{2(25-5)} = \frac{31}{40} = 0.775$$

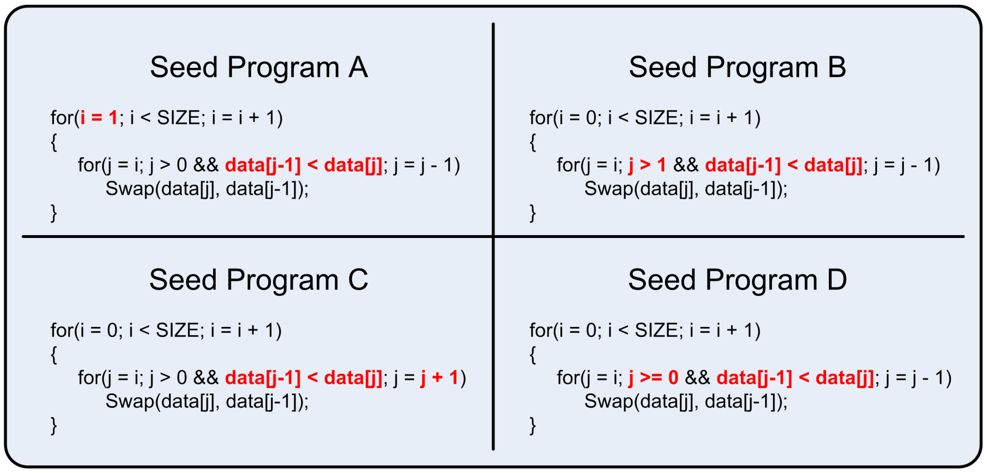
Figure 3.2: Example Insertion Sort Fitness Calculation

3.3. SEED PROGRAMS

Four programs were used to test the CASC prototype. Each seed program had one common error (shared by the other seeds) and one unique error. The common error causes the implementations to sort the values in descending order. This defect was chosen because the worst case in the fitness function described in Section 3.2.2 would be yielded by an array of values in reverse order (i.e., descending instead of ascending). Since the system is attempting to correct sorting programs which sort in ascending order this seemed like a logical choice for the worst case. This defect, while relatively simple to correct (modification of two array subscripts), affects the fitness of the seeds greatly.

The unique errors in the seed programs are “off by one” errors. This type of error was chosen because, while it is simple to correct, it is very common in development. The effect that the unique errors have can vary from having only a small impact on the fitness to causing a run-time or time-out error in the program. The errors used in the seeds attempt to span this large range of effects that these errors can have on a program.

The evolvable sections of the four seed programs are shown in Figure 3.3 with the errors in each highlighted. Figure 3.1 can be used as a reference to see the exact errors.



Seed Program A

```
for(i = 1; i < SIZE; i = i + 1)
{
    for(j = i; j > 0 && data[j-1] < data[j]; j = j - 1)
        Swap(data[j], data[j-1]);
}
```

Seed Program B

```
for(i = 0; i < SIZE; i = i + 1)
{
    for(j = i; j > 1 && data[j-1] < data[j]; j = j - 1)
        Swap(data[j], data[j-1]);
}
```

Seed Program C

```
for(i = 0; i < SIZE; i = i + 1)
{
    for(j = i; j > 0 && data[j-1] < data[j]; j = j + 1)
        Swap(data[j], data[j-1]);
}
```

Seed Program D

```
for(i = 0; i < SIZE; i = i + 1)
{
    for(j = i; j >= 0 && data[j-1] < data[j]; j = j - 1)
        Swap(data[j], data[j-1]);
}
```

Figure 3.3: Seed Programs

3.4. CASC CONFIGURATIONS

The role of the variation operator **mutation** is to explore the problem space. So for this reason a base experimental set was used as well as three other sets of experiments which used modified mutation values. The first set used an enhanced mutation rate (i.e., higher chance of child mutation) relative to the base experiment. The second set used an enhanced mutative proportion (i.e., mutation affects more genes). Lastly, the third set has both an enhanced mutation rate and proportion. The relevant configuration values used in all the experiments are shown in Table 3.1 and Table 3.2. The termination conditions used in the experiments were maximum number of generations and goal fitness, as specified in Table 3.1.

Configurations Common to Both Populations	
Max. Number of Generations	200
Goal Fitness	1.0
Population Size	20
Number of Children Per Generation	5
Number of Opponents During Evaluation	6
Program Population Configuration	
Parent Selection Tournament Size	5 (20%)
Competition Tournament Size	5 (20%)
Test Case Population Configuration	
Parent Selection Tournament Size	3 (15%)
Competition Tournament Size	3 (15%)
Number of Values in Test Case	20
Range of Values	0-100

Table 3.1: Configuration Values

	Program Mutation (%)		Test Mutation (%)	
	Rate	Proportion	Rate	Proportion
Base Set	33	9	33	25
Set 1	66	9	66	25
Set 2	33	27	33	50
Set 3	66	27	66	50

Table 3.2: Mutation Percentages

Five full runs were conducted, each run consisting of 16 experiments (four seed programs with four different configurations). A typical single experiment using the above configuration values takes around 90 hours to complete on the Missouri S&T Numerically Intensive Computing (NIC) cluster. During experimentation, four to five experiments would typically be running at one time on the NIC-cluster. The end result is that a full run can take anywhere from a week to two weeks to execute. Clearly, the amount of time required to run a single experiment is decidedly less than optimal. Potential solutions to this issue are discussed further in the Future Work section (Section 6).

4. RESULTS AND DISCUSSION

Four different programs were run using four different configurations at five runs a piece, yielding 80 experimental runs in total. First the results will be presented and discussed in the scope of each seed program. Next a look at the performance of the CASC system as a whole will be presented. In this discussion, let the notation SP_X denote seed program X.

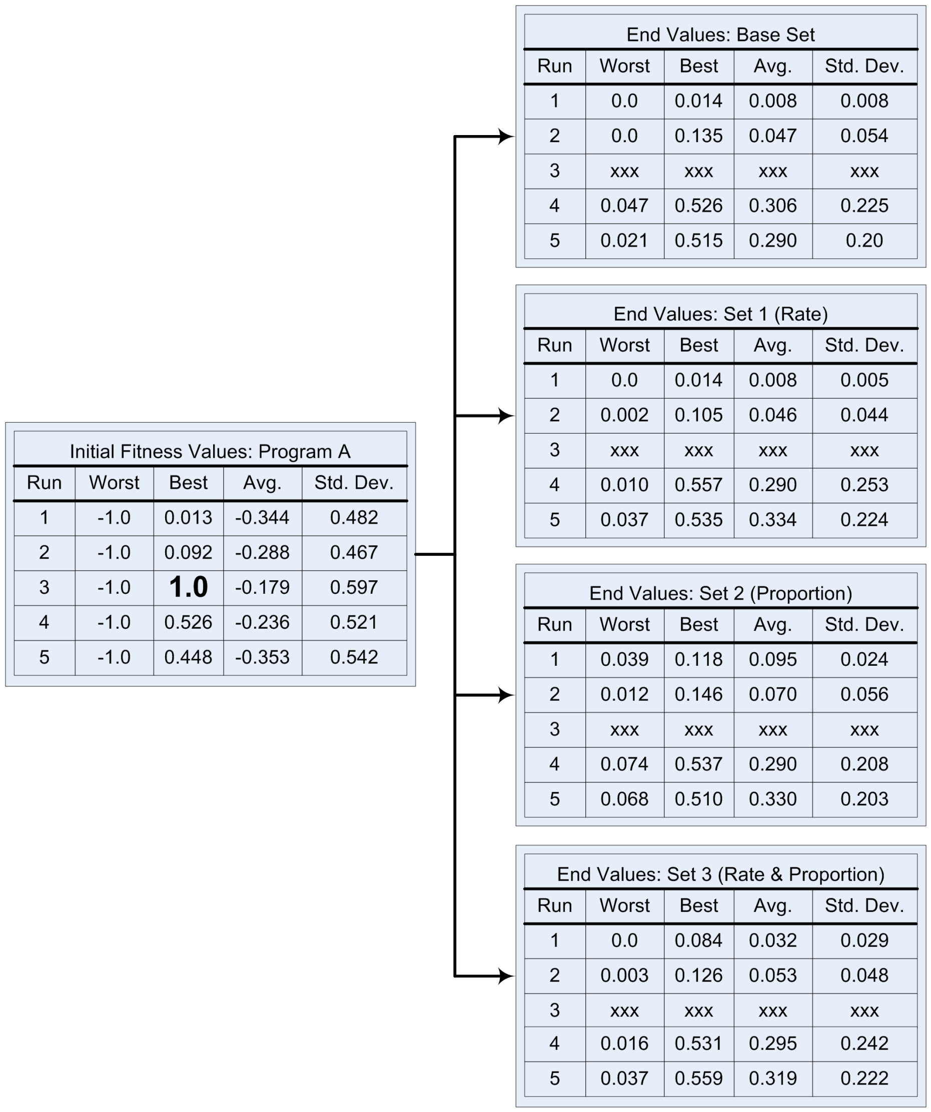
4.1. SEED PROGRAM A RESULTS

The unique bug placed in SP_A was not really a bug at all. The modification of the starting point in the outer loop actually just resulted in the loss of an iteration that would not perform anything. Thus, the main problem presented in SP_A was the common error causing the data to sort in reverse order. The correction of this error requires a minimum of two modifications, specifically adjustment to the array indices used in the second expression in the inner for-loop.

As can be seen in Figure 4.1, the first two runs struggled to correct the error, achieving a max fitness of 0.146. Runs four and five had slightly better results, yielding a top fitness of 0.557. The third run was the most successful of the five, yielding an individual with a 1.0 fitness in the initial population. The evolvable section for this individual is shown in Figure 4.2.

There are a few interesting things about this result. The first thing is that both of the bugs present in SP_A have been fixed in this individual. Even though the unique bug in SP_A was not really a bug at all it is still corrected. In and of itself this does not affect the program at all; however, when paired with the other modifications made it becomes necessary.

The sections highlighted in blue and orange in Figure 4.2 are the next sections of note. The blue section actually has no affect on the program, it just makes the logic evaluation in the outer for loop a little different than in the correct version shown in Figure 3.1, leaving the program itself functionally unmodified. The orange section, on its own basically accelerates the sort by one iteration, causing the program to skip the first intended iteration and adding an extra iteration to the end, which



Initial Fitness Values: Program A

Run	Worst	Best	Avg.	Std. Dev.
1	-1.0	0.013	-0.344	0.482
2	-1.0	0.092	-0.288	0.467
3	-1.0	**1.0**	-0.179	0.597
4	-1.0	0.526	-0.236	0.521
5	-1.0	0.448	-0.353	0.542

End Values: Base Set

Run	Worst	Best	Avg.	Std. Dev.
1	0.0	0.014	0.008	0.008
2	0.0	0.135	0.047	0.054
3	xxx	xxx	xxx	xxx
4	0.047	0.526	0.306	0.225
5	0.021	0.515	0.290	0.20

End Values: Set 1 (Rate)

Run	Worst	Best	Avg.	Std. Dev.
1	0.0	0.014	0.008	0.005
2	0.002	0.105	0.046	0.044
3	xxx	xxx	xxx	xxx
4	0.010	0.557	0.290	0.253
5	0.037	0.535	0.334	0.224

End Values: Set 2 (Proportion)

Run	Worst	Best	Avg.	Std. Dev.
1	0.039	0.118	0.095	0.024
2	0.012	0.146	0.070	0.056
3	xxx	xxx	xxx	xxx
4	0.074	0.537	0.290	0.208
5	0.068	0.510	0.330	0.203

End Values: Set 3 (Rate & Proportion)

Run	Worst	Best	Avg.	Std. Dev.
1	0.0	0.084	0.032	0.029
2	0.003	0.126	0.053	0.048
3	xxx	xxx	xxx	xxx
4	0.016	0.531	0.295	0.242
5	0.037	0.559	0.319	0.222

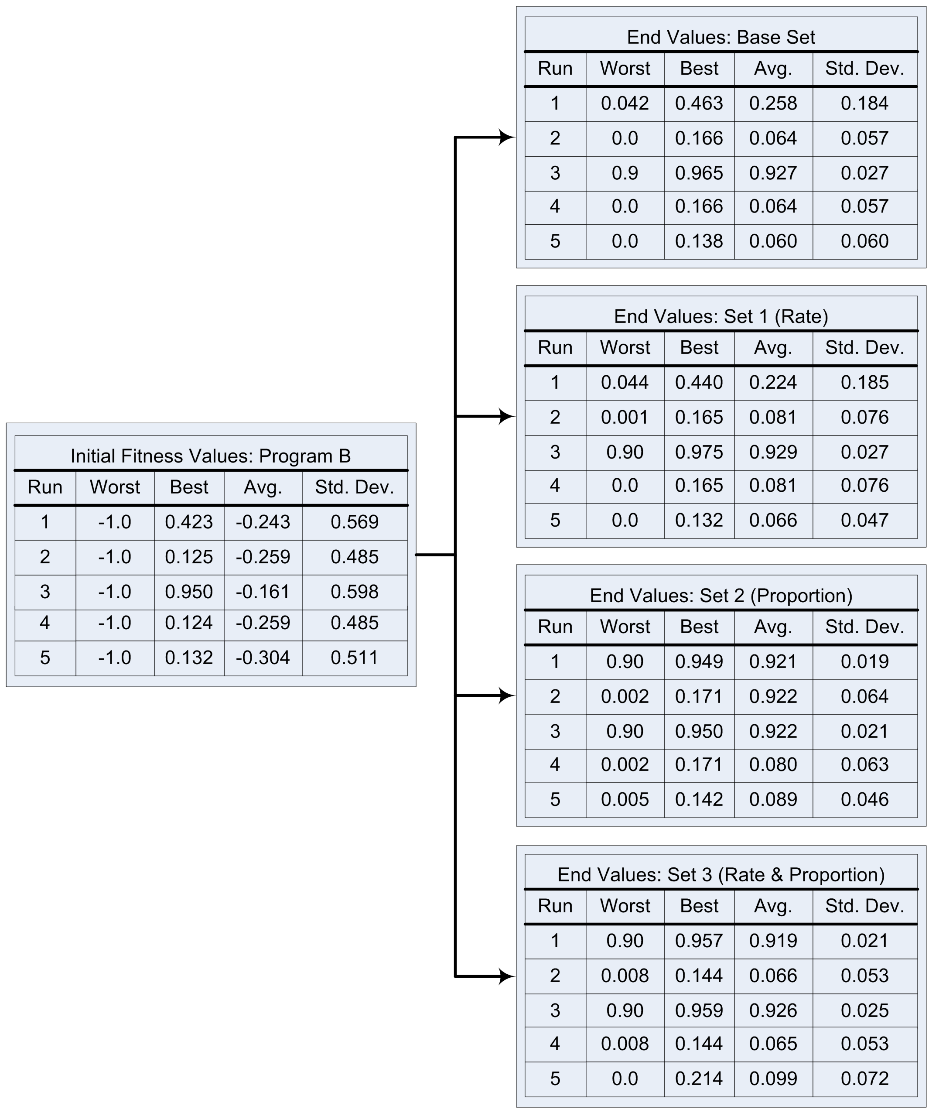
Figure 4.1: Summary of Seed Program A Data

```
for(i = 0; i - 1 < SIZE - 1; i = i + 1)
{
    for(j = i + 1; j > 0 && data[j] < data[j-1]; j = j - 1)
        Swap(data[j], data[j-1]);
}
```

Figure 4.2: Evolvable Section of Most Fit Result From Seed Program A

would cause the program to walk off the data array (which could potentially yield a run-time error or lose a value from the data set).

The orange modification paired with the correction of the unique SP$_A$ bug makes the loss of the first iteration acceptable (since, as was noted earlier, with unique SP$_A$ bug corrected the first iteration does nothing). However, the program can still walk off the data array. This means that the program does sort data, which is a good result, but also inconsistently causes a run-time error. This inconsistency brings up some interesting points, which are discussed in the Future Work section (Section 6).



Initial Fitness Values: Program B

Run	Worst	Best	Avg.	Std. Dev.
1	-1.0	0.423	-0.243	0.569
2	-1.0	0.125	-0.259	0.485
3	-1.0	0.950	-0.161	0.598
4	-1.0	0.124	-0.259	0.485
5	-1.0	0.132	-0.304	0.511

End Values: Base Set

Run	Worst	Best	Avg.	Std. Dev.
1	0.042	0.463	0.258	0.184
2	0.0	0.166	0.064	0.057
3	0.9	0.965	0.927	0.027
4	0.0	0.166	0.064	0.057
5	0.0	0.138	0.060	0.060

End Values: Set 1 (Rate)

Run	Worst	Best	Avg.	Std. Dev.
1	0.044	0.440	0.224	0.185
2	0.001	0.165	0.081	0.076
3	0.90	0.975	0.929	0.027
4	0.0	0.165	0.081	0.076
5	0.0	0.132	0.066	0.047

End Values: Set 2 (Proportion)

Run	Worst	Best	Avg.	Std. Dev.
1	0.90	0.949	0.921	0.019
2	0.002	0.171	0.922	0.064
3	0.90	0.950	0.922	0.021
4	0.002	0.171	0.080	0.063
5	0.005	0.142	0.089	0.046

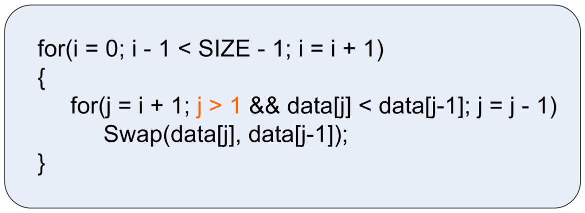
End Values: Set 3 (Rate & Proportion)

Run	Worst	Best	Avg.	Std. Dev.
1	0.90	0.957	0.919	0.021
2	0.008	0.144	0.066	0.053
3	0.90	0.959	0.926	0.025
4	0.008	0.144	0.065	0.053
5	0.0	0.214	0.099	0.072

Figure 4.3: Summary of Seed Program B Data

4.2. SEED PROGRAM B RESULTS

The unique error in SP$_B$ caused the first element in the data array to be left out of the sort. The effect this error has on the fitness is absolutely dependent on the data itself. If the first element in the input data is supposed to be the last element, then the error will have a considerable effect on the fitness. However, if the first element is in its proper place, then the error would have no effect at all.



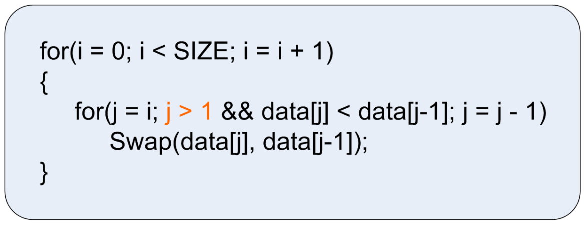
```
for(i = 0; i - 1 < SIZE - 1; i = i + 1)
{
    for(j = i + 1; j > 1 && data[j] < data[j-1]; j = j - 1)
        Swap(data[j], data[j-1]);
}
```

Figure 4.4: Evolvable Section of Most Fit Result From Seed Program B

The results of the SP$_B$ experiments are summarized in Figure 4.3. As can be seen in the figure, the second, fourth and fifth runs seemed to really struggle with the bugs, yielding a top fitness of 0.214. Runs one and three, however, performed quite well. The top individual produced by the run one experiments yielded a fitness of 0.957. The top individual from run three (and the best individual produced by the SP$_B$ experiments) yielded a fitness of 0.975. The evolvable section of this individual is shown in Figure 4.4.

The first thing of note about this result is that it is very similar to the best result produced for SP$_A$. The only difference is the section in Figure 4.4 highlighted in orange. This section is the unique bug for SP$_B$. The evolution managed to correct the common error but was not able to correct the unique SP$_B$ error, and in the process of trying introduced a new error (the inconsistent error described at the end of the previous section).

Inspection of the evolvable section of the top individual produced by run one (shown in Figure 4.5) brings up an interesting point. The error unique to SP$_B$ is still present, however the errors from the SP$_A$ result are not. This means that this program actually has fewer errors than the more fit program from run 3, yet it has a slightly lower fitness. The test cases used to evaluate the individuals along with the nature of the unique SP$_B$ error (described at the beginning of this section) can safely be assumed to be the cause of the discrepancy between the two fitness values. This would imply that future experiments may benefit from subjecting individuals to more trials during the evaluation phase, to make it less likely that an individual is only tested against easy opponents.



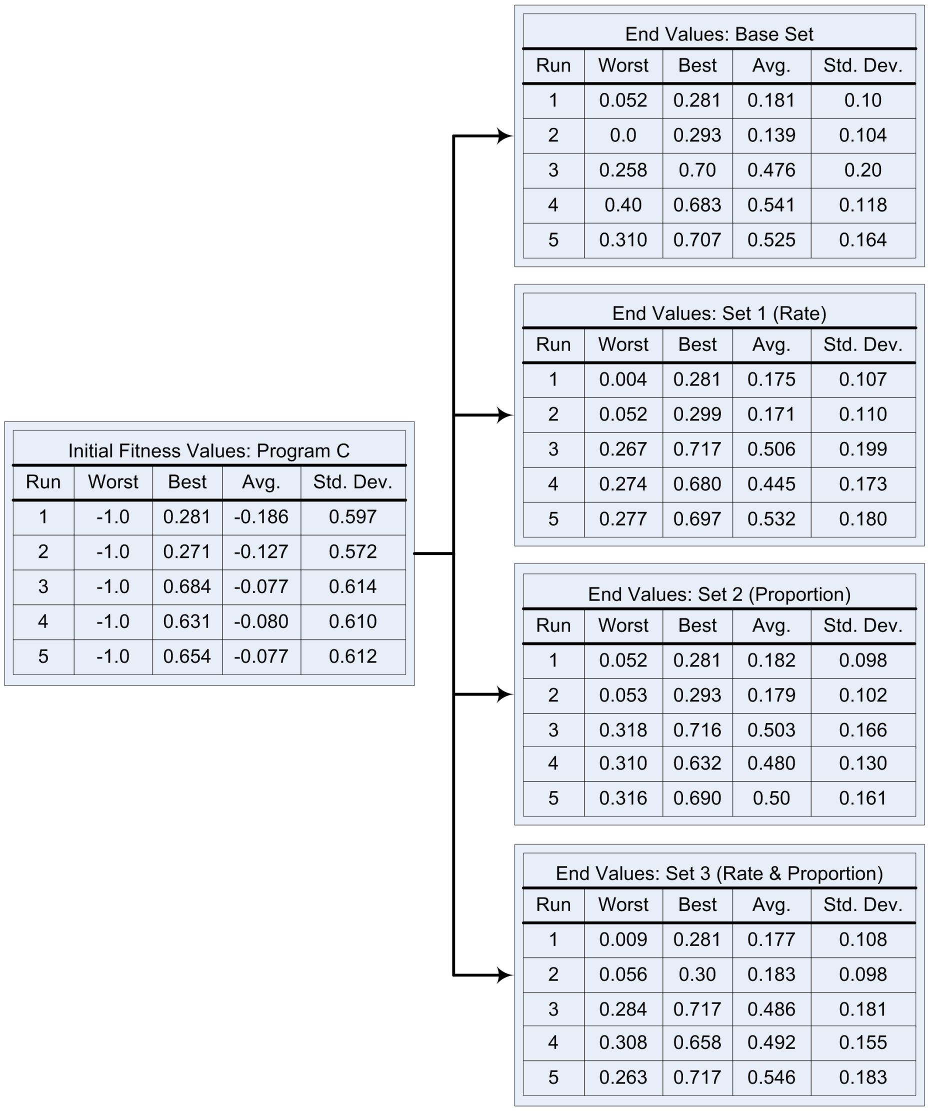
```
for(i = 0; i < SIZE; i = i + 1)
{
    for(j = i; j > 1 && data[j] < data[j-1]; j = j - 1)
        Swap(data[j], data[j-1]);
}
```

Figure 4.5: Evolvable Section of Second Most Fit Result From Seed Program B

4.3. SEED PROGRAM C RESULTS

The unique error in SP$_C$ is a fairly common error which has a large effect on the program (and the fitness). The inner for loop is supposed to be a decrement loop (i.e., the loop control variable decreases by one each iteration) but in SP$_C$ it is an increment loop. In the worst case, this error can cause the program to walk off of the data array resulting in the loss of values from the data set and/or cause a run-time error. Correcting the error is simple enough, only requiring the modification of the third expression in the inner for loop. The results from the SP$_C$ experiments are shown in Figure 4.6.

Runs one and two performed moderately well, yielding a top fitness of 0.30. The experiments in runs three, four, and five had more success. Both runs four and five yielded individuals with a fitness of 0.717, the highest in the SP$_C$ experiments. The evolvable section of the program yielded by run 3 is shown in Figure 4.7. The red



Initial Fitness Values: Program C

Run	Worst	Best	Avg.	Std. Dev.
1	-1.0	0.281	-0.186	0.597
2	-1.0	0.271	-0.127	0.572
3	-1.0	0.684	-0.077	0.614
4	-1.0	0.631	-0.080	0.610
5	-1.0	0.654	-0.077	0.612

End Values: Base Set

Run	Worst	Best	Avg.	Std. Dev.
1	0.052	0.281	0.181	0.10
2	0.0	0.293	0.139	0.104
3	0.258	0.70	0.476	0.20
4	0.40	0.683	0.541	0.118
5	0.310	0.707	0.525	0.164

End Values: Set 1 (Rate)

Run	Worst	Best	Avg.	Std. Dev.
1	0.004	0.281	0.175	0.107
2	0.052	0.299	0.171	0.110
3	0.267	0.717	0.506	0.199
4	0.274	0.680	0.445	0.173
5	0.277	0.697	0.532	0.180

End Values: Set 2 (Proportion)

Run	Worst	Best	Avg.	Std. Dev.
1	0.052	0.281	0.182	0.098
2	0.053	0.293	0.179	0.102
3	0.318	0.716	0.503	0.166
4	0.310	0.632	0.480	0.130
5	0.316	0.690	0.50	0.161

End Values: Set 3 (Rate & Proportion)

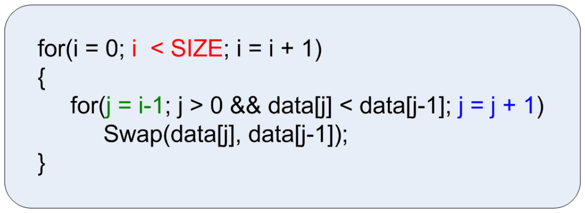
Run	Worst	Best	Avg.	Std. Dev.
1	0.009	0.281	0.177	0.108
2	0.056	0.30	0.183	0.098
3	0.284	0.717	0.486	0.181
4	0.308	0.658	0.492	0.155
5	0.263	0.717	0.546	0.183

Figure 4.6: Summary of Seed Program C Data

```
for(i = 0; i + 1 < SIZE - 1; i = i + 1)
{
    for(j = i+1; j > 0 && data[j] < data[j-1]; j = j + 1)
        Swap(data[j], data[j-1]);
}
```

Figure 4.7: Evolvable Section of Most Fit Result From Seed Program C (Run 3)

section in the figure will cause the program to not consider the last two elements in the data array, the green section will cause the first element in the data set to not be considered and the blue section is the error unique to SP$_C$. This means that the system managed to correct the common error; however, while trying to correct the unique error introduced two more errors into the program.



```
for(i = 0; i  < SIZE; i = i + 1)
{
    for(j = i-1; j > 0 && data[j] < data[j-1]; j = j + 1)
        Swap(data[j], data[j-1]);
}
```

Figure 4.8: Evolvable Section of Most Fit Result From Seed Program C (Run 5)

The evolvable section of the best program seen in run 5 is shown in Figure 4.8. Despite having the same fitness, the run 3 and run 5 results are quite different. The red section in Figure 4.8 is correct, whereas in the run 3 result the same section has two errors. Instead of skipping the first data element, the green section in the run 5 will result in the program walking off the data array. The one similarity that the run 5 result has with the run 3 result is that they both fixed the common error, while missing the unique error.

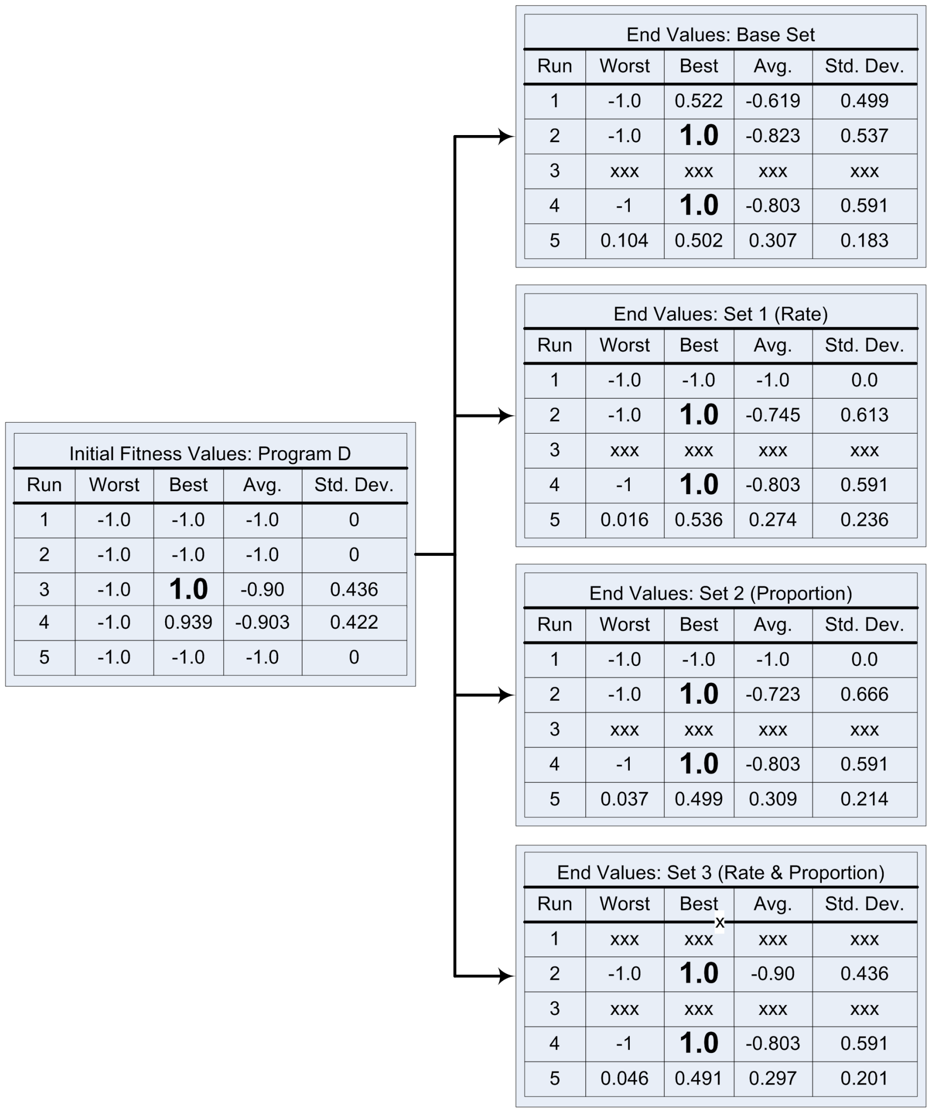
Based on these results it may seem like the SP$_C$ experiments were not very successful; however, there is one more aspect to consider. The average standard deviation in the end population (based on fitness) for the SP$_C$ experiments is 0.142, which is slightly larger than would be ideal for an end result. A large standard deviation in the final population implies that convergence on a solution had not yet occurred; however, the average end standard deviations in the SP$_C$ experiments are only somewhat large (relative to the observed fitnesses), meaning that convergence was likely to happen soon. which means that this result is more of an intermediate result rather than a final one. The average end standard deviation in the SP$_A$ experiments is 0.130, which shows that the SP$_A$ experiments may have terminated prematurely as well. Due to the amount of time these experiments typically take, it would be ideal if the system could either detect if this was the case before ending an experiment or possibly continue the execution of a terminated experiment. These possibilities are discussed further in the Future Work section (Section 6).

4.4. SEED PROGRAM D RESULTS

The unique error in SP$_D$ results in an extra iteration of the inner loop, allowing the program to provide negative array indices to the data array. This is similar to the error in the SP$_A$ result, allowing the program to walk off of the array; however, most operating systems are much more cautious when it comes to negative array indices. This means that this error will result in a run-time error much more often than the error in the SP$_A$ result.

This makes these experiments particularly hard, since if a run-time error occurs, then a very low, constant fitness value is given to the program. Since the majority of the initial population will most likely still have this error, then the majority of the population will have the same fitness. This can be seen in the results shown in Figure 4.9.

In runs one, two, and five the standard deviation of the initial population is zero, meaning that there is absolutely no diversity in the population. Population diversity is very important in EC since it is responsible for guiding the evolutionary process. If there is no diversity at all, then the algorithm is reduced to a glorified random search. The effect this has can be seen in the results for the SP$_D$ experiments. In sets one and two of run one the system was unable to make any progress whatsoever with the program; however, in the base set it was able to find an individual which yielded an individual with a fitness of 0.522 (note: set three of run one experienced technical difficulties and cannot be shown at this time). However, run two (which started in the exact same way as run one) had success in all four of its experiments. Each one found an individual with a 1.0 fitness within the first four generations of evolution. Run three also found an individual with a 1.0 fitness, in the initial population. Run four was also able to find an individual yielding 1.0 fitness in all four of its experiments. Lastly, the run 5 experiments all started without much promise, but ended up performing fairly well, yielding a top fitness of 0.536. So the results of the SP$_D$ experiments range from complete inability to make any advancement with



Initial Fitness Values: Program D

Run	Worst	Best	Avg.	Std. Dev.
1	-1.0	-1.0	-1.0	0
2	-1.0	-1.0	-1.0	0
3	-1.0	**1.0**	-0.90	0.436
4	-1.0	0.939	-0.903	0.422
5	-1.0	-1.0	-1.0	0

End Values: Base Set

Run	Worst	Best	Avg.	Std. Dev.
1	-1.0	0.522	-0.619	0.499
2	-1.0	**1.0**	-0.823	0.537
3	xxx	xxx	xxx	xxx
4	-1	**1.0**	-0.803	0.591
5	0.104	0.502	0.307	0.183

End Values: Set 1 (Rate)

Run	Worst	Best	Avg.	Std. Dev.
1	-1.0	-1.0	-1.0	0.0
2	-1.0	**1.0**	-0.745	0.613
3	xxx	xxx	xxx	xxx
4	-1	**1.0**	-0.803	0.591
5	0.016	0.536	0.274	0.236

End Values: Set 2 (Proportion)

Run	Worst	Best	Avg.	Std. Dev.
1	-1.0	-1.0	-1.0	0.0
2	-1.0	**1.0**	-0.723	0.666
3	xxx	xxx	xxx	xxx
4	-1	**1.0**	-0.803	0.591
5	0.037	0.499	0.309	0.214

End Values: Set 3 (Rate & Proportion)

Run	Worst	Best	Avg.	Std. Dev.
1	xxx	xxx	xxx	xxx
2	-1.0	**1.0**	-0.90	0.436
3	xxx	xxx	xxx	xxx
4	-1	**1.0**	-0.803	0.591
5	0.046	0.491	0.297	0.201

Figure 4.9: Summary of Seed Program D Data

```
for(i = 0; i < SIZE; i = i + 1)
{
    for(j = i; j >= 0 && data[j] < data[j-1]; j = j - 1)
        Swap(data[j], data[j-1]);
}
```

Figure 4.10: Evolvable Section of Most Fit Result From Seed Program D

the program up to success.

The evolvable section of the individual with 1.0 fitness from run two, set one is shown in Figure 4.10. The system was able to correct the common error, but unable to correct the unique error. As was mentioned earlier, the error unique to SP$_D$ is very similar to the error in the SP$_A$ result. This is a good example of how an error of this type can in one case result in a run-time error for the program (e.g., the individuals of run one, sets one and two) and in another case result in no problem at all (e.g., the most fit individuals of the SP$_A$ and SP$_D$ experiments). This further underscores the fact that something needs to be done to address this type of error.

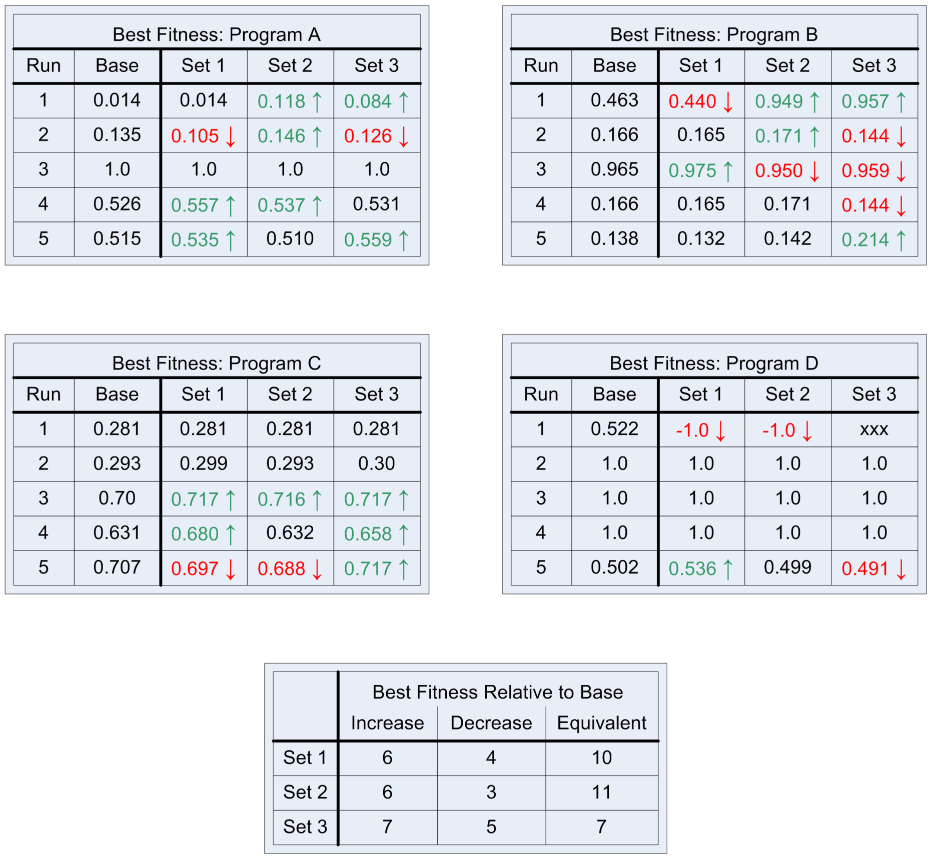
4.5. CASC PERFORMANCE ACROSS ALL EXPERIMENTS

The experiments as a whole provide information on how the CASC system performed in general. Information regarding the effect that the different mutation values had on the evolutionary process, the system's ability to navigate the problem space, and the overall success rate of the system are the major results yielded from these experiments.

4.5.1. Effect of Mutation Configurations. As was described in Section 3, there were four different configurations used on each seed program. These configurations differed only in the mutative rate and mutative proportion. The base used fairly low values for both of these, set one used an increased mutative rate (relative to the base set), set two used an increased mutative proportion, and set three used increased values for both rate and proportion.

The various configuration sets were used to explore the effect that the mutation configuration could have on the CASC evolutionary process. Figure 4.11 shows a summary of the experimental results with a focus on the four configurations. In this figure, values highlighted in red performed worse than the base set, values highlighted in green performed better than the base set, and values in black were equivalent (or very nearly equivalent) to the base set.

There is no clear trend in the values shown in Figure 4.11 so the Wilcoxon rank-sum test was applied to these values in an attempt to determine if they came from distributions with equal means (i.e., if the population means are significantly different). The test was performed between the base configuration and set one results,



Best Fitness: Program A				
Run	Base	Set 1	Set 2	Set 3
1	0.014	0.014	0.118 ↑	0.084 ↑
2	0.135	0.105 ↓	0.146 ↑	0.126 ↓
3	1.0	1.0	1.0	1.0
4	0.526	0.557 ↑	0.537 ↑	0.531
5	0.515	0.535 ↑	0.510	0.559 ↑

Best Fitness: Program B				
Run	Base	Set 1	Set 2	Set 3
1	0.463	0.440 ↓	0.949 ↑	0.957 ↑
2	0.166	0.165	0.171 ↑	0.144 ↓
3	0.965	0.975 ↑	0.950 ↓	0.959 ↓
4	0.166	0.165	0.171	0.144 ↓
5	0.138	0.132	0.142	0.214 ↑

Best Fitness: Program C				
Run	Base	Set 1	Set 2	Set 3
1	0.281	0.281	0.281	0.281
2	0.293	0.299	0.293	0.30
3	0.70	0.717 ↑	0.716 ↑	0.717 ↑
4	0.631	0.680 ↑	0.632	0.658 ↑
5	0.707	0.697 ↓	0.688 ↓	0.717 ↑

Best Fitness: Program D				
Run	Base	Set 1	Set 2	Set 3
1	0.522	-1.0 ↓	-1.0 ↓	xxx
2	1.0	1.0	1.0	1.0
3	1.0	1.0	1.0	1.0
4	1.0	1.0	1.0	1.0
5	0.502	0.536 ↑	0.499	0.491 ↓

	Best Fitness Relative to Base		
	Increase	Decrease	Equivalent
Set 1	6	4	10
Set 2	6	3	11
Set 3	7	5	7

Figure 4.11: Results of Configuration Sets

base configuration and set two results, and base configuration and set three results. For all three tests on all four seed programs it was found that the population means were not significantly different. This result could be due to lack of samples; each test was run with only 5 samples per set. Another possible cause could be that the CASC evolutionary system is not very sensitive to mutative adjustments. In any case, this result needs to be investigated, which is discussed more in the Future Work section (Section 6).

4.5.2. Population Diversity. The diversity of the evolving population is very important to an EC algorithm. If a population is diverse that means that there is a large amount of unique genetic material in the population, making the reproduction and mutation phases very effective. Alternatively, if a population is not very diverse, then the individuals are similar, meaning that there is not very much unique genetic material in the population. Lack of unique genetic material hinders the reproduction and mutation phases, which in turn hinders the entire evolutionary process. The standard deviation of a population fitness is one metric for the diversity of a population.

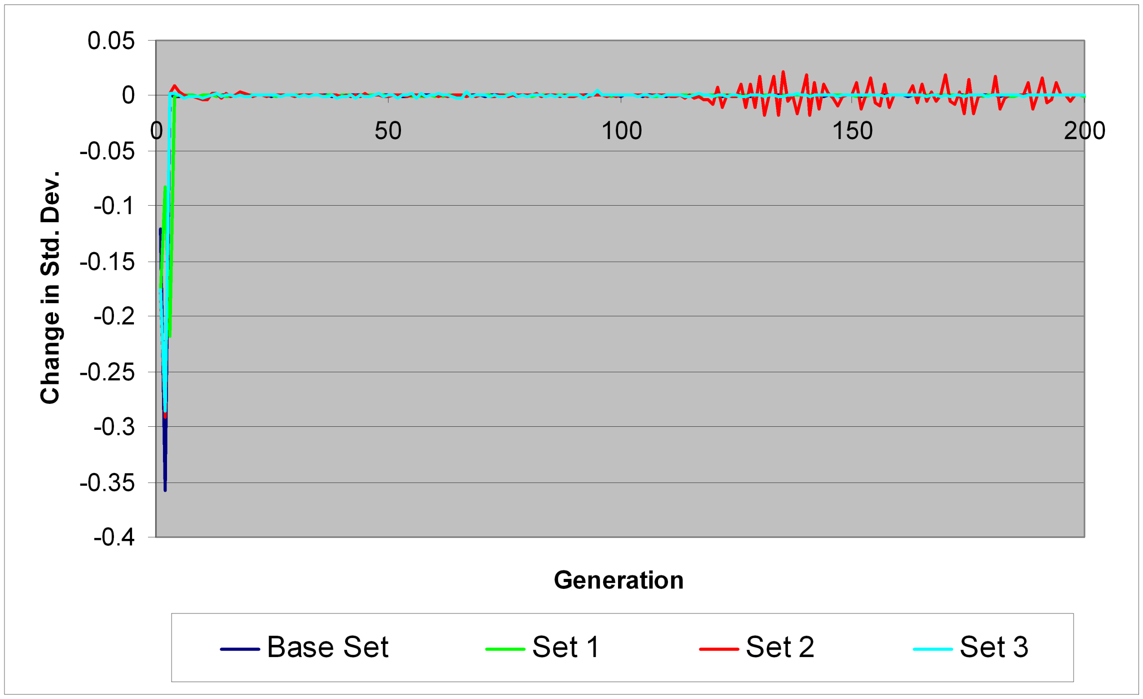


Figure 4.12: Example of Standard Deviation Change in Program Population

Figure 4.12 shows a graph representing the amount of change per generation in the standard deviation of the program population's fitness in four experiments which were run to completion (i.e., 200 generations); specifically the four SP_A run one experiments. A negative value in this graph means a reduction in the standard deviation of the population (i.e., a loss of diversity). A typical EC algorithm will have quick initial convergence on a solution and then display minor to moderate perturbations, representing the individuals exploring the problem space around the local maxima (and potentially moving to other local maxima). All four sets show the quick initial convergence but the perturbations that follow are very small. Around generation 120, however, set 2 begins to show the type of movement that is desirable in the population diversity, even though the movements are still somewhat small. This further promotes the hypothesis from the previous section, stating that higher mutative proportion seems to be beneficial to the CASC evolutionary process. Even though set 2 does display some desirable behavior in diversifying its population, the changes are still somewhat diminutive. For this reason it would most likely be beneficial to look into methods which further promote population diversity in the evolutionary process.

The initial population diversities indicate how well the method of creating the program population by cloning and mutating the seed program is working. The majority of the initial populations have standard deviations around 0.5, except for runs one and two of the SP_D experiments. In fact, if those two extreme cases are omitted, the average initial population standard deviation is 0.542, indicating that on average there is a good spread of individuals in the initial population. This implies that the population creation method is quite effective and is working as intended.

5. CONCLUSION

The focus of this thesis was to show whether or not a co-evolutionary system linking an evolutionary cycle employing GP techniques with a evolutionary cycle employing EA techniques could be used to perform software testing and correction. A prototype of such a system was introduced for use in preliminary experimentation and ultimately show the validity of this concept. The principle difference between the prototype and the system envisioned was the use of an EP algorithm (rather than GP) to evolve the program population. While the use of this algorithm somewhat changes the applicability of the system, EP and GP are similar enough that the prototype is still functional as a proof of concept for the envisioned CASC system.

A total of 80 experiments were conducted using the prototype. Each experiment consisted of seeding a program with two errors (of varying severity) and providing it to the system. Also the configurations which control the system's behavior were varied for each experiment. Success in these experiments is indicative of the validity of the CASC concept. The prototype reported 16 such successes. While the results yielded from these successes still had some errors (all of which fall in a category of errors which the prototype cannot yet detect consistently) the majority of the successful results were very near being truly correct. All of the results (both successful and otherwise) have conveyed a great deal of information about the system's current operation, possible venues of future experimental investigation, and potential improvements that can be made to the system which were not apparent before. While the prototype may not have yielded the success rate that will be expected from the completed system it still served its purpose as a prototype well, yielding enough successes to indicate that there is in fact validity to the CASC concept. Further work is required to determine the system's real world validity.

6. FUTURE WORK

As was mentioned before, the current CASC system is a prototype. There is still much work to be done on the system. These are some of the main improvements that could be made to the system:

- The highest priority improvement to the system is making the program evolution actually able to modify the structure of the code, not just the critical points; i.e., make the system use an unrestricted GP algorithm to evolve the program population. Employing unrestricted GP for this problem will result in an explosion in the size of the problem space, and thus pose a very considerable challenge to the CASC system. Methods to counter this increase in problem space size are already being investigated. The method currently being investigated to counter this problem is to represent (and evolve) programs in a more abstract fashion, i.e., rather than using specific code elements (e.g., for, while, do-while) in program representation, use more generalized code elements (e.g., loop). From each abstract program multiple specific programs (represented in the typical fashion) could be created. The specific programs could be evaluated in the same way as they are currently and based off of these evaluations the abstract programs could be evaluated and, as a result, evolved.
- The long run-times of the CASC prototype are a serious issue. A few options to alleviate this problem are:
  - A method has been devised to make the CASC system use symmetric multiprocessing as well as cluster parallel computing. The main idea is to make the system support server-client style multi-computing, making the system able to run on any networked set of computers capable of using shared (primary or secondary) memory. In the proposed system, the server would perform the duties that the main node does in the current system and the client(s) would be the worker nodes.
  - The system spends a great deal of time compiling programs. Reducing the number of compiler invocations would significantly decrease the system

run-time. Batch compiling the evolvable sections of the programs would result in much fewer compiler invocations. The feasibility of batch program compilation in the CASC system needs to be investigated.

  – Being able to run more experiments concurrently would also increase the productivity of the system. In order to do this, more computing resources are required. The National Science Foundation manages a large cluster known as the Teragrid, allotting computing time on this cluster would be a huge benefit to the CASC project. More locally, the Missouri S&T Intelligent Systems Center manages a cluster for their funded research and the Missouri S&T Natural Computation Laboratory is considering construction of a Beowulf cluster to be used by the lab members.

- Work has been discussed to make the CASC system not only able to modify program structure but to also write the programs itself. It would do this according to some set of specifications that would be provided to the system. If this addition was successful, then the system would be able to write and correct its own code.

- Wappler's Particle Swarm Optimization (PSO) implementation [40] was able to outperform an EA as a search technique for test data generation. Since the CASC system uses an EA to evolve the test cases, some work should be done to investigate if PSO would be appropriate for the CASC system. Use of PSO in a co-evolutionary system falls close the concept that Travis Service, a fellow lab-mate in the Missouri S&T NC-Lab, recently published on a method called co-optimization [34, 33]. Co-optimization is based on co-evolution, the primary difference is that the controlling algorithms for each of the populations are not required to be based on an evolutionary model. Co-optimization allows the evolutionary model to be removed in favor of a technique better suited to the class of problem being addressed. It is likely that co-optimization would be useful to the CASC system, since the scope of problems that the system can face is large.

- As was seen in many of the experimental results, there is an error which can occur inconsistently in programs produced by the system, which may result in a false positive. Work needs to be done to enable the system to catch such errors, since this is clearly undesirable.

- Currently, parameter tuning is used to determine the configuration values used by the system (i.e., values are determined before hand and stay static during the run). Adaptive parameter control [6] allows for the various configuration values in the system (e.g., mutative rate, mutative proportion, etc.) to change during the evolutionary process in response to the state of the population(s). Dynamic configuration parameters may help the system navigate the problem space more effectively, possibly overcoming the problems seen in the prototype with local maxima.

- It appeared that the SP_C experiments were terminated somewhat prematurely. It may be beneficial to have a way to use population diversity (or lack thereof) as a termination condition. Additionally, if the state of the system could be stored, it would be possible to pick up a previously terminated experiment and continue its execution. Storing the state of the system would entail storing the program population, test case population, and the state of the random number generator used in the system. The feasibility of these additions should be investigated.

- A parameter sensitivity analysis is necessary to find the configuration vales which have the most affect on the evolutionary process in the CASC system.

- Currently the CASC system uses an evolutionary model similar to that of the general EA model rather than the classical GP evolutionary model. The main difference between the two models is that in the general EA model, reproduction occurs and then mutation possibly occurs and in the classical GP model either reproduction or mutation can occur for an individual, but not both. The use of the EA model instead of the GP model may explain the reason why the system is not very sensitive to changes in the mutation configurations. This needs to be investigated further.

Aside from the major improvements listed previously, there are certainly many more smaller improvements that need to be made, and many more will be discovered with further experimentation and development.

APPENDIX A

CASC Configuration File

```
# *********************************
# ***** General Configuration *****
# *********************************
# File to output run info to
Log File = ./runLog.txt

# Method to seed random number generator, can be 'random' or a file
# can be specified which contains seed values
Seeding Method = random

# Number of generations per run
Number of Generations = 200

# Whether or not to report all individual's fitness values and the end
Dump Fitness Upon Completion = 0

# Number of slots to have in the hash table
Hash Table Size = 200

# Amount of time to allow a program to run before deciding time-out
Alloted Program Run Time (sec) = 2

# Number of runs to complete
Number of Runs = 1

# Fitness value that, if achieved, signals completion
Goal Fitness = 1

# Directories to hold the temporary test bed files
Test Bed Root Dir = ./tbfiles/
Test Bed Exe Dir = ./tbfiles/exe/
Test Bed Output Dir =  ./tbfiles/output/
```

```
# Directory to hold command files for the worker nodes
Command File Dir = ./workerCmd/

# Number of opponents to evaluate an individual against
Set Size = 6



# ***********************************
# ***** Test Case Configuration *****
# ***********************************
# Number of test case individuals to have in the test case population
Test Population Size = 20

# Number of values in the test case
Test Array Length = 20

# How many pairs of parents to pick out in reproduction (i.e. how
# many children are produced per generation)
Number of Parent Pairs (test) = 5

# Percentage of the population to use in selection tournament
Tournament Percentage (test) = 0.15

# The chance that a child will be mutated
Mutation Rate (test) = 0.66

# The percentage of genes that will mutate during mutation
Mutative Proportion (test) = 0.25

# Min and max values for the values in a test case
Element Min Value = 0
Element Max Value = 100

# Percentage of the population to use in competition tournament
Competition Tournament Percentage (test) = 0.15



# ********************************
# ***** Program Configuration *****
# ********************************
# Number of program individuals to have in the program population
Program Population Size = 20

# Program that is to be evolved and corrected
Program To Evolve = ./someProgram.cpp

# Tags to designate evolvable code in the seed program
Critical Section Open Tag =  //Open Critical
Critical Section Close Tag = //Close Critical

# File to hold temporary proeprocessor output
Preprocessor Output File = ./preproc.csc

# How many pairs of parents to pick out in reproduction (i.e. how
# many children are produced per generation)
Number of Parent Pairs (prog) = 5

# Percentage of population to use in tournament parent selection
Tournament Percentage (prog) = 0.2

# The chance that a child will be mutated
Mutation Rate (prog) = 0.66
```

```
# The percentage of critical points that will mutate during mutation
Mutative Proportion (prog) = 0.09

# Percentage of the population to use in competition tournament
Competition Tournament Percentage (prog) = 0.2


# ***********************************
# ***** Implementation Specific *****
# ***********************************
# The number of distinct values to use in the LCS alphabet
LCS Alphabet Size = 5
```

APPENDIX B

The Phases of a Program in the CASC System

B.1. INSERTION SORT SEED PROGRAM

```
#include<iostream>
#include<stdlib.h>
using namespace std;

template<typename T>
void Swap(T& a, T& b)
{
  T c = a;
  a = b;
  b = c;
}

int main(int argc, char* argv[])
{
  long int int1, int2
  const int data_size = argc-1;
  int data[data_size];

  cout << data_size << endl;
  for(int i = 1; i <= data_size; ++i)
  {
    data[i-1] = atoi(argv[i]);
    cout << data[i-1] << " ";
  }
  cout << endl;

  //Open Critical
  for(int1 = 0; int1 < data_size; int1 = int1 + 1)
  {
    for(int2 = int1; data[int2] > data[int2-1] && int2 > 1; int2 = int2 - 1)
      Swap(data[int2], data[int2-1]);
  }
  //Close Critical

  for(int i = 0; i < data_size; ++i)
    cout << data[i] << " ";
  cout << endl;

  return 0;
}
```

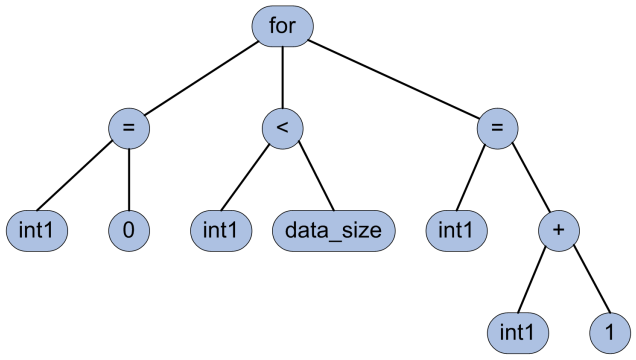
Legend

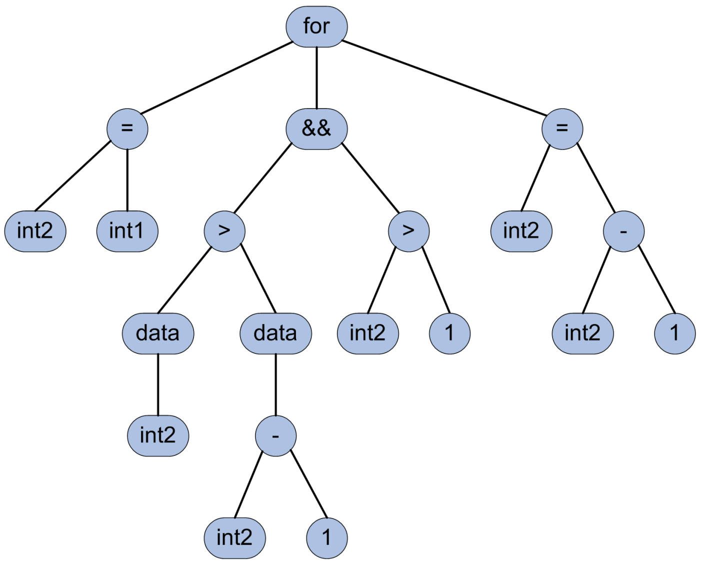
Header Code

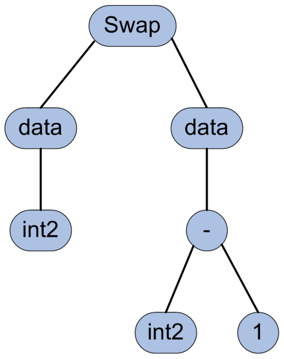
Evolvable Code

Footer Code

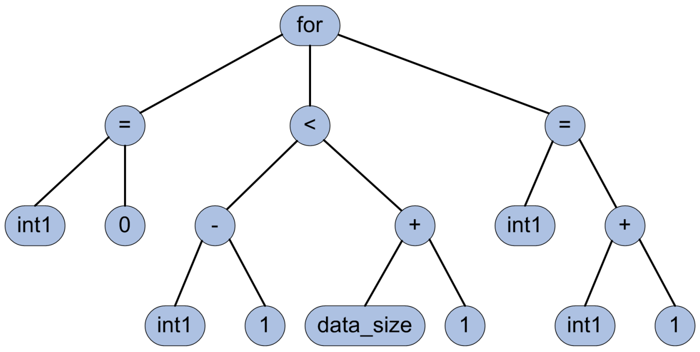
B.2. CASC AST OF SEED PROGRAM

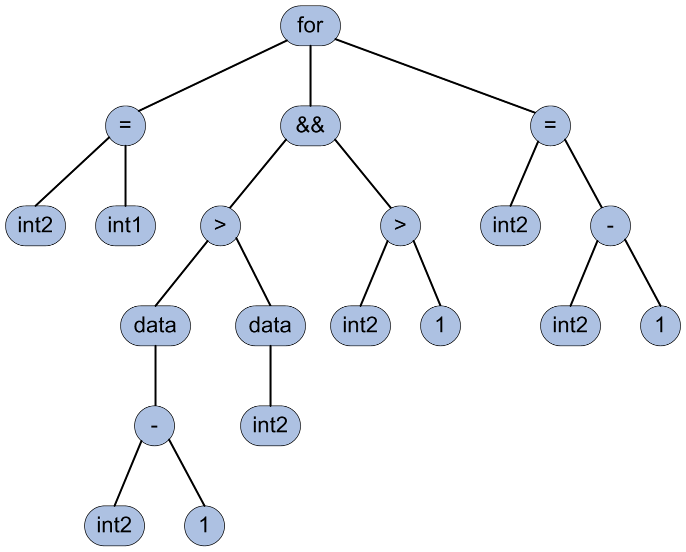


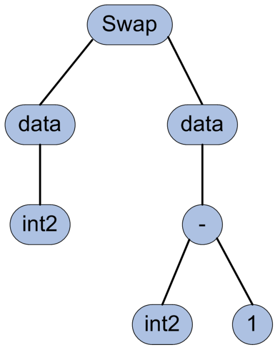




B.3. AST OF PROGRAM RESULTING FROM EVOLUTION







B.4. PROGRAM RESULTING FROM EVOLUTION

<u>Legend</u>

Header Code

Evolved Code

Footer Code

```
#include<iostream>
#include<stdlib.h>
using namespace std;

template<typename T>
void Swap(T& a, T& b)
{
  T c = a;
  a = b;
  b = c;
}

int main(int argc, char* argv[])
{
  long int int1, int2
  const int data_size = argc-1;
  int data[data_size];

  cout << data_size << endl;
  for(int i = 1; i <= data_size; ++i)
  {
    data[i-1] = atoi(argv[i]);
    cout << data[i-1] << " ";
  }
  cout << endl;

  //Open Critical
  for(int1 = 0; int1-1 < data_size+1; int1 = int1 + 1)
  {
    for(int2 = int1; data[int2-1] > data[int2] && int2 > 1; int2 = int2 - 1)
      Swap(data[int2], data[int2-1]);
  }
  //Close Critical

  for(int i = 0; i < data_size; ++i)
    cout << data[i] << " ";
  cout << endl;

  return 0;
}
```

BIBLIOGRAPHY

[1] J. P. Cartlidge. *Rules of Engagement: Competitive Coevolutionary Dynamics in Computational Systems*. PhD thesis, University of Leeds, 2004.

[2] W. Cook, W. Cunningham, W. Pulleybank, and A. Schrijver. *Combinatorial Optimization*. John Wiley and Sons, 1997.

[3] N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In J. John, editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*. Carnegie Mellon University, 1985.

[4] S. Dick and A. Kandel. *Computational Intellignece in Software Quality Assurance*. World Scientific, 2005.

[5] E. Dustin, J. Rashka, and J. Paul. *Automated Software Testing*. Addison-Wesley, 1999.

[6] A. Eiben and J. Smith. *Introduction to Evolutionary Computing*. Springer-Verlag, 2003.

[7] L. J. Fogel. *Artificial Intelligence through Simulated Evolution*. Wiley, Chichester, UK, 1966.

[8] M. P. I. Forum. Mpi: A message passing inerface standard. Technical Report CS-94-320, University of Tennessee, 1994.

[9] J. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, Cambridge MA, 1992, 1st Edition: 1975, University of Michigan Press, Ann Arbor.

[10] T. C. Jones. Measuring programming quality and productivity. *IBM Systems Journal*, 17(1):39–63, 1978.

[11] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, pages 209–215, August 1990.

[12] B. Korel. Automated test data generation for programs with procedures. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 209–215, 1996.

[13] J. R. Koza. *Genetic Programming: On the Programming of Computers by the Means of Natural Selection*. MIT Press, Cambridge MA, 1992.

[14] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge MA, 1994.

[15] J. R. Koza. *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann, 1999.

[16] J. R. Koza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

[17] F. Lammermann and S. Wappler. Benefits of software measures for evolutionary white-box testing. In *Proceedings of GECCO 2005 - the Genetic and Evolutionary Computation Conference*, pages 1083–1084, Washington DC, 2005. ACM, ACM Press.

[18] W. B. Langdon and R. Poli. *Foundations of Genetic Programming.* Springer-Verlag, 2002.

[19] T. Mantere. *Automatic Software Testing by Genetic Algorithms.* PhD thesis, University of Vaasa, 2003.

[20] T. Mantere and J. T. Alander. Developing and testing structural light vision software by co-evolutionary genetic algorithm. In *QSSE 2002 The Proceedings of the Second ASERC Workshop on Quantative and Soft Computing based Software Engineering*, pages 31–37. Alberta Software Engineering Research Consortium (ASERC) and the Department of Electrical and Computer Engineering, University of Alberta, Feb 2002.

[21] T. Mantere and J. T. Alander. Testing digital halftoning software by generating test images and filters co-evolutionarily. In *Proceedings of SPIE Vol. 5267 Intelligent Robots and Computer Vision XXI: Algorithms, Techniques, and Active Vision*, pages 257–258. SPIE, October 2003.

[22] C. Michael and G. McGraw. Automated software test data generation for complex programs. In *Proceedings of the 13th IEEE International Conference on Software Engineering*, pages 136–146, 1998.

[23] C. Michael, G. McGraw, and M. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, pages 1085–1110, December 2001.

[24] W. Miller and D. Spooner. Automatic generation of floating point test data. *IEEE Transacitons on Software Engineering*, pages 223–226, September 1976.

[25] M. Newman. Software errors cost u.s. economy $59.5 billion annually. NIST News Release, June 2002.

[26] R. P. Pargas, M. J. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification & Reliability*, pages 263–282, 1999.

[27] T. Parr. ANTLR Parser Generator. 2007. University of San Francisco. Accessed June 2007. `http://www.antlr2.org`.

[28] R. Poli. *Parallel Distributed Genetic Programming, Invited Chapter in D. Corne, M. Dorigo and F. Glover (Eds), New Ideas in Optimisation*, chapter 27, pages 403–431. McGraw-Hill, 1999.

[29] R. Pressman. *Software Engineering: A Practitioner's Approach*, chapter 4,13. McGraw-Hill, 6th edition, 2005.

[30] C. D. Rosin. *Coevolutionary Search Among Adversaries.* PhD thesis, University of California: San Diego, 1997.

[31] C. D. Rosin and R. K. Belew. Methods for competitive co-evolution: Finding opponents worth beating. In L. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 373–380, San Francisco, CA, 1995. Morgan Kaufmann.

[32] C. D. Rosin and R. K. Belew. New methods for competitive coevolution. *Evolutionary Computation*, 5(1):1–29, 1997.

[33] T. Service. Co-optimization: A generalization of coevolution. Master's thesis, Missouri University of Science and Technology, 2008.

[34] T. Service and D. Tauritz. Co-optimization algorithms. In *Proceedings of GECCO 2008 - the Genetic and Evolutionary Computation Conference*, pages 387–388, 2008.

[35] S. Smith. *A Learning System Based on Genetic Adaptive Algorithms.* PhD thesis, University of Pittsburgh, 1980.

[36] U. S. A. Standards Coordinating Committee of the IEEE Computer Society. *IEEE Standard Glossary of Software Engineering Terminology.* IEEE Computer Society, December 10 1990 (Reaffirmed in 2002).

[37] P. Tonella. Evolutionary testing of classes. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 119–128, Boston, Massachusetts, 2004. ACM Press.

[38] S. Wappler and F. Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *Proceedings of GECCO 2005 - the Genetic and Evolutionary Computation Conference*, pages 1053–1060, Washington DC, 2005. ACM, ACM Press.

[39] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *Proceedings of GECCO 2006 - the Genetic and Evolutionary Computation Conference*, pages 1925–1932, Seattle, Washington, 2006. ACM, ACM Press.

[40] A. Windisch, S. Wappler, and J. Wegener. Applying particle swarm optimization to software testing. In *Proceedings of GECCO 2007 - the Genetic and Evolutionary Computation Conference*, pages 1121–1128, London, United Kingdom, 2007. ACM, ACM Press.

VITA

Joshua Lee Wilkerson was born on September 13, 1982 in Springfield, Missouri. He graduated with honors from Strafford High School in the spring of 2001 and enrolled as a undergraduate at the University of Missouri - Rolla later that fall. He graduated cum laude in summer of 2005 with a BS in computer science. He was enrolled in the computer science graduate program in fall of 2005. He received his master's degree in computer science in the fall of 2008 and plans to continue on and complete his PhD degree.