The Performance of Coevolutionary Topologies in Developing Competitive Tree Manipulation Strategies for Symbolic Regression

A dissertation submitted in satisfaction of the requirements for the degree Master of Science in Computer Science

by

Martin Ombura

Supervised by:
Geoff Nitschke

February 2020

I know the meaning of plagiarism and declare that all the work in this document, save for that which is properly acknowledged, is my own.

Abstract

Computer bugs and tests are antagonistic elements of the software development process, with the former attempting to corrupt a program and the latter aiming to identify and fix the introduced faults. The automation of bug identification and repair schemes through automated software testing is an area of research that has only seen success in niche areas of software development but has failed to progress into general areas of computing due to the complexity and diversity of programming languages, codebases and developer coding practices. Unlike traditional engineering fields such as mechanical or civil where project specifications are carefully outlined and built towards, software engineering suffers from a lack of global standardization required to "build from a spec".

In this study we investigate a coevolutionary spec-based approach to dynamically damage and repair programs mathematical programs (functions). We opt for mathematical functions instead of software due to their functional similarities and simpler syntax and semantics. We utilize symbolic regression (SR) as a framework to analyze the error maximized by bugs and minimized by test. We adopt a hybrid evolutionary algorithm (EA) that implements the tree based phenotypic structure of genetic programming (GP) and the list-based chromosome of genetic algorithm (GA) that permits embedding of mathematical tree manipulation (MTM) strategies, as well as adequate selection mechanisms for search. Bugs utilize the MTM strategies in their chromosome to manipulate the input program (IP) with the aim of maximizing the error while tests adopt a set of their own MTM strategies to repair the damaged program using a spec generated from the IP to guide the repair process. Both adversarial agents are investigated in four common coevolutionary topologies, Hall of Fame (HoF), K-Random Tournaments (KRT), Round Robin (RR) and Single Elimination Tournament (SET).

We ran 1556 simulations each generating a random polynomial that the bugs and tests would have to contend over in all 4 topologies. We observed that KRT with a low k value of 5 performs best from a computational and fitness standpoint for all bugs and tests. Bugs were dominant in nearly all topologies for all polynomial complexities, whereas tests struggled in the HoF, RR and SET topologies as the input programs became more complex. The competitive landscape however was quite chaotic with the best individuals lasting a maximum of 14 generations out of 300, with the average top individuals lasting only 1 generation. This made predictions on when the best individuals would be born nearly impossible as the coevolutionary landscape changed quite rapidly and non-deterministically.

The kinds of MTM strategies selected by both bugs and tests depended on the level of complexity of the input programs. For input programs that had negative polynomials, the best bugs opted to delete the program entirely and build a completely new tree, whereas the best tests were unable to select viable specialized strategies to repair such programs. For programs that had large polynomial degrees, bugs opted for strategies that added nodes their underlying GP tree, in the hopes of damaging the input program more. Tests on the other hand implemented strategies to carefully reduce the complexity of the polynomial. Tests however, frequently overcompensated when attempting to fix the fit bugs, leading to mediocre solutions.

TABLE OF CONTENTS

List of Tables

LIST OF FIGURES

1 INTRODUCTION

The term “bug” has been around since the days of Thomas Edison, and was used to refer to a small, easily repairable problem (Kidwell, 1998). Software reliability has been a recurring issue plaguing software engineers since the dawn of computing (Hangal & Lam, 2002). As requirements for applications grow, so do the chances of finding bugs in software. Weimer, Nguyen, Le Goues, and Forrest (2009) estimate that in some scenarios the cost of fixing bugs can hit 90% of a software project’s budget. Automated debugging techniques have come about to alleviate most of the problems brought about by the arduous nature of debugging. Automated debugging although helpful, has shown to contain flaws as most techniques involve identifying potential issues in isolated regions of code without inferring context from the larger codebase as whole. This has previously led developers to apply patches that have had unwarranted effects elsewhere in the code base (Parnin & Orso, 2011). Approaches used in automated debugging have also shown to be of little practical utility as they are architected on assumptions of how software engineers’ program (Parnin & Orso, 2011). This approach has the damaging side-effect of weakening programmers’ abilities to correctly identify faults (Xia, Bao, Lo, & Li, 2016). The consensus behind automated debugging is split, and best practices urge software engineers to first develop tests in order to mitigate the chances of bugs appearing in the code base.

Software testing enables engineers to find bugs in their programs by (Arcuri & Yao, 2014). Software tests place isolated sections of code under a series of checks to ensure code blocks work as intended (Arcuri & Yao, 2014; Sagarna, Arcuri, & Yao, 2007). Testing has the drawback of being a costly, tedious and time-consuming process that cannot be overlooked within the development process (Arcuri & Yao, 2008a). Arcuri and Yao (2008a) report that approximately $20 billion is spent annually on remedying code that could have been avoided if better tests were written. Exhaustive software testing is an intractable problem and developers look to ensure maximal code coverage with the least amount of effort (Kuhn, Wallace, & Gallo, 2004). There is therefore a desire among developers and project managers to mitigate the issues associated with testing, and one solution is by automating the testing process (Arcuri & Yao, 2008b). The automation of testing is not trivial, and researchers have looked to artificial intelligence (AI) to solve this issue.

Wappler and Wegener (2006) state that AI has shown to be less error prone than humans in certain technological domains. AI error rates in areas such as image and voice recognition have decreased sharply over the past decade (Brynjolfsson, Rock, & Syverson, 2018). Despite these advances, AI has shown difficulty transferring this success to areas such as automatic programming and automated self-repair (Arbuckle & Requicha, 2010). The latter where AI is used to automatically detect, diagnose and mend errors in programs has seen exploration, and research has shown promise in niche areas of programming, however nothing groundbreaking or production-ready has materialized (Brun et al., 2009). Majority of automated test techniques still require significant manual intervention and complex behavior modelling that can at times defeat the purpose (Wang, Pastore, Goknil, Briand, & Iqbal, 2015). The complexity behind automating testing is exacerbated by the several philosophies behind how good testing should be accomplished, making it difficult to standardize the process. A push for test specifications that outline requirements for a given unit of code has enabled developers to partially overcome this issue (Wang et al., 2015).

The process of standardization is particularly useful in engineering where standards and specifications are a requirement for most systems. The idea involves developers declaratively outlining the requirements of the program, and the intelligent machine would generate a program based on those requirements. The difficulty comes in bridging high-level specification and low-level implementation (Arcuri & Yao, 2014). Each programming language contains a unique set of syntactic and semantic rules that make generalizations on how to repair code or generate tests difficult. Research is still ongoing in this field, and we present a proof of concept that utilizes simpler programs in the form of mathematical functions to simulate the repair process done by tests using a given specification. Mathematical functions have a lot in common with software programs. They both take in inputs, process those inputs, and return an output. Mathematical functions are conceptually simpler to work with as mathematical axioms are easy to intuit and correctly manipulate in contrast to the several ways in which code can be semantically interpreted and expressed.

In relation to building from a specification, one can construct a mathematical function given a set of data points through a process known as symbolic regression (SR) (Augusto & Barbosa, 2000). In SR, a collection of points in cartesian space act as a specification that guides the SR algorithm to generate an expression that encompasses all those points. The branch of AI that has had significant traction with SR is evolutionary computation (EC), particularly genetic programming (GP) due to ability to represent mathematical operators and variables as a phenotypic tree structure (McConaghy, 2011). As the evolutionary process (EP) takes place, selection recombines and mutates various GP solutions until a solution that matches the spec is obtained.

Although GP is ideal for solving SR problems, most research does not address how GPs or evolutionary algorithms (EAs) in general, can adapt to anomalies such as faults or bugs that manifest along the EP when attempting to solve SR based problems. Most attempts at SR involve obtaining the perfect solution in an environment where there are no agents that are directly attempting to introduce error in the SR trees that are being evolved. This non-adversarial environment is useful in solving for an optimal SR-GP individual that adheres to a given specification, but often does not hold well when attempting to find generalizable techniques that can be used to guarantee satisfactory fitness across multiple evolving faulty programs. The question of whether there exists a set of generalizable program fixers remains to be investigated.

In our research we introduce two kinds of evolutionary individuals known as bugs and tests. Bugs and tests are adversarial in nature, with one trying to damage an input program (IP), and the other attempting to apply fixes. Due to their adversarial nature, we opt for a competitive coevolutionary approach to model the adversarial interaction between these agents with regards to fault introduction and repair. From an SR domain, where programs are represented as mathematical functions, we introduce the idea of a mathematical tree manipulation (MTM) strategy, which is an operation an agent can take to manipulate a given mathematical expression. As SR-GP representations are in the form of tree-structures, some examples of MTM strategies include, adding an integer to a leaf node, mutating a non-terminal operation or removing a sub tree. These general MTM strategies can be directly embedded in the chromosome of the evolutionary agent. Since SR-GP only uses a tree as its representation, an additional list-based chromosome will need to be added to the agent to house the set of MTM strategies. We opt for a hybrid EA approach known as a GA-P that contains characteristics of both genetic algorithms (GAs) for the list-based chromosome and GPs for the underlying mathematical tree representation (Zelinka, Oplatkova, & Nolle, 2005). In our model, a set of MTM strategies can form the chromosomal structure of both bug and test. During the EP, these adversarial agents will sequentially apply their different MTM strategies embedded in their chromosome to manipulate the underlying tree. Bugs will first apply MTM operations to damage an input tree, then tests will apply their MTM strategies to attempt to fix the damage introduced by the bug. The selection process in the EP will hopefully find a set of viable SR MTM strategies from the tests that can be used to handle anomalies in the mathematical function introduced by bugs.

The interaction between bugs and tests is of great interest during the EP. Competitions in the real world are played using various formats as individuals compete against each other to progress through the tournament. In some sporting formats, league style competitions pit every individual against every other individual keeping track of points each individual accrues as they win their respective matches. In knockout/bracket formats individuals compete against a randomly predetermined set of individuals in a bracket. Each winner of each bracket moves on to face the winner of another bracket, therefore halving the competition size. Knockout style competitions culminate into a finals game where the overall winner is chosen. Using these various competition-based approaches, it may be of importance to attempt various competition formats between bugs and tests. These formats are more formally referred to as coevolutionary algorithms or topologies. Some well-defined topologies in the coevolutionary literature include, Round Robin (RR), Hall of Fame (HoF), K-Random Tournaments (KRT) and Single Elimination Tournament (SET). In RR every individual in a generation competes against every other individual in the same generation. RR is similar to a league style format where the individual's average fitness is used to gauge their selection probability (Jaśkowski, Krawiec, & Wieloch, 2008a). HoF uses the RR format to guide competition, however the fittest individual from each generation is archived and reintroduced later in the tournament to hopefully strengthen and guide the EP to obtain the fittest individual across all generations (Ficici & Pollack, 2003). In KRT, all individuals play *k* randomly selected competitors from the population. Those with the best record are more likely to be selected for the next generation (Hiew, Tan, & Lim, 2017). Lastly the SET format mimics the knockout style format of popular sports like tennis or knockout football. In SET, like individuals compete against each other and the winner of that set of individuals is added to a set of champions. Those champions then undergo selection to determine who moves on to the next generation.

The effect of the different topologies and the quality of individuals they produce would be of benefit to analyze and discuss in addition to how the complexity of the input program affects the performance of both individuals.

1.1 Research Objectives

This research aims to fulfill the following objectives:

1. Design a hybrid GA-P solution to represent both sets of adversarial individuals *(bugs and tests)* that uses a list-based GA chromosome structure to embed SR MTM strategies that when expressed, manipulate an underlying phenotypic GP tree.
2. Implement HoF, RR, KRT and SET as a set of diverse coevolutionary topologies to compare the performance of the competitions between bugs and tests.
3. Analyze the set of SR strategies that allow both competing individuals to perform the best over each topology and over a diverse set (over 1500) of polynomial input programs (mathematical expressions).

1.2 Contributions

The primary contribution proposed by this research is the ability to coevolve and enable the mutual development of competent adversarial agents *(bugs and tests)* that can adapt strategies to outperform each other in damage and repair of SR based domains. This form of symbiotic learning can ensure if scaled to more complex SR or even software domains, more complex bugs can be created that can be repaired by equally advanced tests. The resulting EA parameters can be used to generate competent bugs or tests depending on the requirements of the user. The research also expands on standard EAs such as GAs and GPs to provide a fitting hybrid that adequately fulfills the requirements of the domain of adversarial damage and repair of SR programs. In this research we also highlight the use of spec-oriented design that can be used with EAs to regenerate faulty programs from the spec provided. We provide information on the performance of various coevolutionary topologies and their effect on competing solutions in an SR domain.
Lastly the research invites readers to abstract domain specific strategies that can be used when attempting to introduce deliberate faults or fix existing faults in their respective domains. In our paper, we focus on SR as the primary domain, and we generate a set of SR-based MTM strategies that can be used for general manipulation of mathematical GP trees to produce the desired outcomes.

1.3 Scope

The scope of the research will not deal with actual software but replace the concept of a software program with a mathematical function. The mathematical functions tested will be single variable higher order polynomials. We chose single variable functions due to the ability to quickly generate a spec that ranges over a given independent domain. This spec is generated before the EP begins and is evaluated against by each individual during fitness calculation. It is a critical code section whose performance has been optimized to allow for reasonably timed runs.

2 Literature Review

2.1 Symbolic Regression

Symbolic regression (SR) is generally defined as a *"white-box approach to mapping input variables to output variables"* (McConaghy, 2011, p. 235). In mathematics, SR refers to the creation of a mathematical expression that generalizes the relation between a given set of function based domains *(independent variables)* to a set of codomains *(dependent variables)* (Karaboga, Ozturk, Karaboga, & Gorkemli, 2012). For example, given a set of coordinates on the x, y cartesian plane, find the mathematical expression that best maps the x values onto the y values. The SR process uses the relationships of the various variables to determine a generalized formula (Smits & Kotanchek, 2005).

SR has shown to be helpful in the field of mathematics, where traditional techniques struggle to find complex relationships. However, a drawback with most SR implementation strategies is dealing with symbolic complexity. As a result, SR is noted to be slower than other non-linear modeling techniques such as neural networks (Smits & Kotanchek, 2005). A subsequent risk noted by Smits and Kotanchek (2005), is the balance of computational performance with the selection of what is deemed a "good" solution. This may not be clear to an evolutionary algorithm (EA), therefore requiring candidate solutions to be further reduced externally. Obvious areas of expression reduction to humans may not be apparent to EAs solving SR problems (Howard & D'Angelo, 1995). For instance, the following expression would be easily simplified by a human to the value 2:

$$\frac{3+3}{3}$$

However, to an EA, the reduction of this expression is not immediately obvious, and can lead to bloated complicated expressions that can easily be simplified.

2.1.1 SR Implementation Strategies

From an evolutionary computing (EC) standpoint, SR has been attempted through different means, such as artificial bee colony algorithms that implement swarm-like foraging behavior of insects to optimize numerical problems (Karaboga & Basturk, 2008). SR has also been used in analytical programming which is an evolutionary technique that places emphasis on the pursuit of analytical representations (Zelinka, Oplatkova, Nolle, & Technology, 2005). SR appears more commonly in genetic programming (GP) *(See Section 2.2)* research often called SR-GP (Icke & Bongard, 2013). In short, GP is a type of EA whose solution representation makes use of tree structures whose nodes hold various types of data. In SR-GPs, GP nodes hold mathematical symbols such as constants, variables, operators. This ability to represent mathematical expressions using trees and the use of a guiding fitness function, allows SR-GPs to begin to approximate the best tree structure that maps the independent and dependent variables the best (Koza, 1994).

2.1.2 Calculating Error in SR

The fitness function is a required component in SR-GP, as it aims at guiding the EP to produce desired (fitter) solutions. In SR-GP, fitness strategies attempt to model the magnitude between observed and anticipated solutions (McKay, Willis, & Barton, 1995). Common fitness methods utilize a mean squared error function that mimics that of a variance calculation shown in *Equation 1* (Keijzer, 2004).

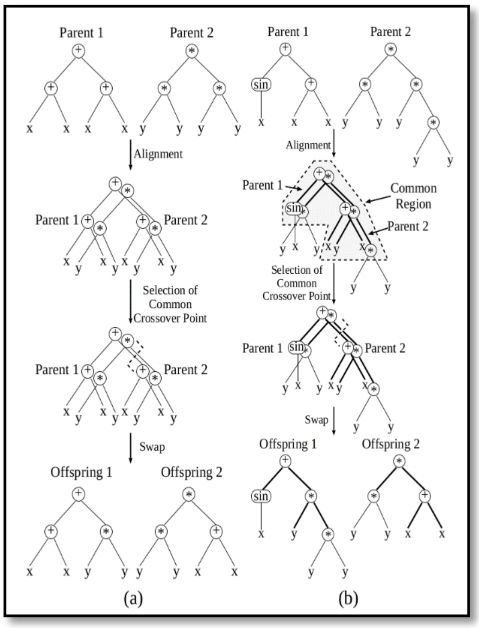
$$E(y,t) = \frac{1}{n}\sum_{i}^{n}(t_i - y_i)^2 \qquad (1)$$

$$E(y,t) = \left(\frac{1}{n}\sum_{i}^{n}(t_i - y_i)^2\right)^{1/2} \tag{2}$$

Where n is the number of points, t is the anticipated variable and y is the observed variable? The variables t_i and y_i are elements in the set of both anticipated variables and observed variables. In some studies, the root mean squared error (RMSE) is preferred as a means of analyzing variance in projected and observed functions (Barmpalexis, Kachrimanis, Tsakonas, & Georgarakis, 2011). The RMSE equation is shown in *Equation (*2).

2.2 GENETIC PROGRAMMING

Genetic programming (GP) involves the search for a tree structured program that is of unspecified size and shape that can be used to solve or approximate a problem (Koza, 1994). Genetic Programs (GPs) offer an intuitive, flexible and efficient way to represent programs using tree structures (Castelli, Silva, & Vanneschi, 2015; Eiben & Smith, 2015). GP is the typical candidate for performing SR due to its representation aligning with the tree-like structure of a mathematical expression (Zelinka, Oplatkova, Nolle, et al., 2005). In SR, GP tree nodes contain terminal and non-terminal sets, where terminal sets contain constants or variables, and non-terminal *(or function-sets)* contain operations (Barmpalexis et al., 2011). GPs use this tree structure to encode mathematical symbols as part of their chromosomes (Searson, Leahy, & Willis, 2010). There is evidence of the success of SR applied to GP in both industrial and academic domains (Searson et al., 2010).



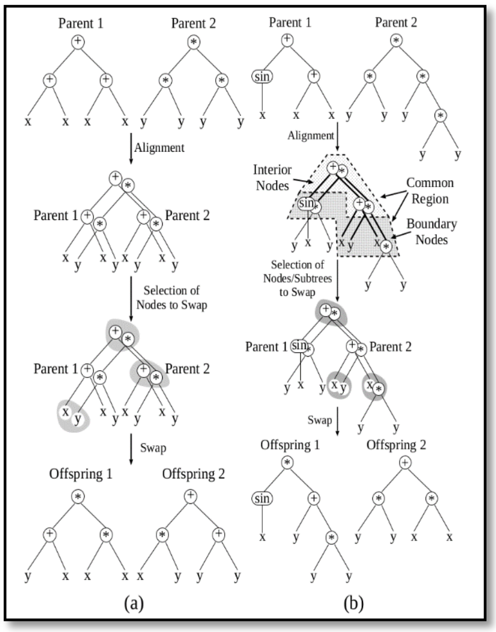


***Figure 2-i**: An example of one-point crossover between two sets of parents. Parents (a) represent same structured parents and parents (b) are different in structure. Note the variation in structure and how one-point crossover accounts for this (Poli & Langdon, 1998). On the right, An illustration of uniform crossover for two sets of parents with varying structure. Parents (a) have the same tree structure and parents (b) parents have irregular structure (Poli & Langdon, 1998).*

2.2.1 GP operators

When GPs evolve, they do so through mutation or crossover. Mutation *(or subtree mutation)* involves selecting a random node within a parent and replacing it with a randomly generated subtree of the same structure (Poli, Langdon, McPhee, & Koza, 2008). Crossover *(or traditional crossover)* swaps entire subtrees from one parent tree with another subtree from another parent tree (Luke & Spector, 1997). The nodes on the parent tree chosen for crossover, are not often selected uniformly and in most situations the crossover can be between terminal sets, consequently swapping minimal genetic material (Poli et al., 2008). A crossover technique commonly used in genetic algorithms (GAs) known as one-point crossover can be translated to a GP representation.

One-point crossover is similar to traditional crossover, however two parents only swap subtrees defined under a common crossover point *(See Figure 2-i).* Uniform crossover on the other hand selects a set of nodes in common regions of the GP and swaps them between parents *(See Figure 2-i)*. Both crossover and mutation are applied probabilistically, with greater probability about 90% or greater being applied to crossover and about 1% applied to mutation (Poli et al., 2008).

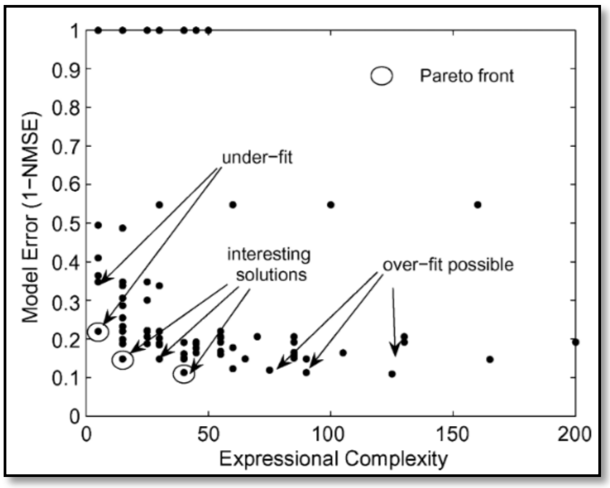


Figure 2-ii*: Pareto front illustration of error against solution complexity. Solutions along the pareto front (closer to the axes) exhibit varying traits ranging from, high error – low expressional complexity, to high expressional complexity – low error (Vladislavleva, Smits, & Den Hertog, 2008, p. 335).*

2.2.2 GP Performance

GPs have been criticized for their hyper-exploratory searches that encompass large solution spaces and spend long times before finding a correct program (Weimer et al., 2009). GP operators such as crossover add new material to solutions that increase the complexity and size of trees, leading to a phenomenon known as GP bloat (Luke & Panait, 2006). In SR-GP there is the risk of accumulating “introns” which are nonfunctional-substructures that contribute little to the actual expression, but are considered areas of noise that introduce complexity (Smits & Kotanchek, 2005). Occam’s Razor principle from machine learning is used to mitigate noise by ensuring only necessary genes are present (Vladislavleva et al., 2008).

Overfit solutions shown in *Figure 2-ii* pose an interesting problem in GPs attempting to solve SR problems. They contain a lot of noise and do not adhere to Occam’s razor principle despite low error rates. GPs inherently have to solve two problems simultaneously the minimization of error, and the minimization of complexity in SR domains (Vladislavleva et al., 2008). In work by Smits and Kotanchek (2005) regarding the performance of a biomass inferential sensor using GP, the pursuit of simple expressions can have diminishing returns on computation performance. Other techniques have attempted to enhance SR-GPs by using guided machine-learning (ML) GP techniques with relative success depending on the problem domain (Icke & Bongard, 2013).

2.2.3 Strongly Typed Genetic Programming and Grammar Guided Genetic Programming

Regular GP parse trees fail to encapsulate key syntactic programming rules within their tree structure. There are no limits governing which nodes are allowed to be children of other nodes consequently leading to the generation of illegal tree structures (Ribeiro, 2008). The reduction semantics of certain arithmetic operations are not fully understood by GPs, for instance multiple addition operations may be used instead of a single or few multiplication operations further attributing to bloat (Davidson, Savic, & Walters, 2003).

Strongly typed genetic programming (STGP) defines and restricts interactions between encoding constructs such as variables, constants and operators. These restrictive rules must be known before initialization and adds overhead to the initialization process (Ribeiro, 2008). GP trees are also prone to semantic incorrectness despite having syntactic integrity. For example, a divide by zero, or square root of a negative number are syntactically accurate, but are semantically undesirable (Keijzer, 2003). Another constraint based technique, is grammar guided genetic programming that uses predefined constraint grammars to place syntactic restrictions on GPs (Hoai, McKay, Essam, & Chau, 2002). Both these techniques aim to manipulate GP trees to grow in constrained manner that is appropriate for their respective domain.

2.2.4 Pathologies of GP Tree Semantics and Syntax

Division by Zero: The semantics required to handle division by zero errors must be explicitly stated prior to GP runs. For instance, Koza opted to return the value of 1 in the event of a divide by zero (Keijzer, 2003). The division operator can also be problematic in its ability to coalesce individuals around local optima, with researchers opting to neglect it entirely (Keijzer, 2003).

Unpredictable Behavior: As GPs explore a solution space, there is the possibility of unlikely mixing of various mathematical operators and constants that would otherwise not interact. For instance trigonometrical, polynomial and exponents intermingling in uncommon ways, but are valid to the GP process (Keijzer, 2003). Function protection is advised in certain areas to maintain mathematical integrity. These checks come with a computational and sometimes exploration cost that ay converge solutions around local optima, preventing exhaustive searches (Keijzer, 2003; Shayan & Chittilappilly, 2004).

GP Repair: GP may employ repair methods to remedy abnormal and at times over-fit expressions (Barbosa & Lemonge, 2002). Repair strategies are research domain specific and can be computationally intensive and in some cases, impossible to implement without having detrimental effects to the search process (Barbosa & Lemonge, 2002). It is important to note that GPs are also bound by only being able to alter the structure of a mathematical expression but not the internal contents, unless through a bespoke mutation (Howard & D'Angelo, 1995). In some cases repair methods may not be feasible and penalties may be a viable alternative (Barbosa & Lemonge, 2002). Depending on the problem domain penalties may range from naïve penalties such as death penalties for abnormal individuals, to more domain specific adaptive penalties that scale along with the EP (Barbosa & Lemonge, 2002).

GPs are a viable choice when attempting to solve SR problems, however the limitations listed above come at a performance and complexity cost. In most cases GPs are often paired with other techniques such as ML in order to specialize the solutions being generated for a given problem domain (Icke & Bongard, 2013). In our research, we also look to augment traditional GP techniques by taking into consideration STGP ideas mentioned in *Section 2.2.3* with regards to program encoding. In order to do so, we utilize a set of mathematical tree manipulation (MTM) strategies *(See Section 3.8)* to carefully limit and control how GP solutions interact. This will hopefully allow us to create better solutions for our test vs bug problem domain. Having these clearly defined strategies will also enable us to clearly articulate how the best bugs or tests are being generated.

These MTM strategies will be contained in a list (chromosome) that each solution will carry. When the chromosome is expressed, the strategies will be applied to the underlying subtree sequentially, manipulating the tree in a predictable manner. This form of list-based chromosome solution resembles that of genetic algorithms (GAs). The solution will therefore be a hybrid, containing both GA characteristics (list-based MTM strategies), and GP characteristics (mathematical subtree). We now take a look at GAs and their characteristics.

2.3 Genetic Algorithms

Genetic Algorithms (GAs) are highly adaptive highly EAs that can be applied to a variety of problem domains (Poon, Carter, & Research, 1995). GAs are an evolutionary methodology originally borrowed from natural selection (Gonçalves, de Magalhães Mendes, & Resende, 2005). GAs were first introduced by Holland for solving difficult optimization problems through the stochastic search of a problem domain (Kao & Zahara, 2008; Poon & Carter, 1995). GAs use linear binary strings of finite length as their chromosomal encoding mechanism (Chai, Huang, Zhuang, Zhao, & Sklansky, 1996). These binary strings encode an area in solution space and are best suited to identifying parameters that maximize a fitness function under a set of rules of natural selection (Kaya & Uyar, 2011; D. H. Kim, Abraham, & Cho, 2007).

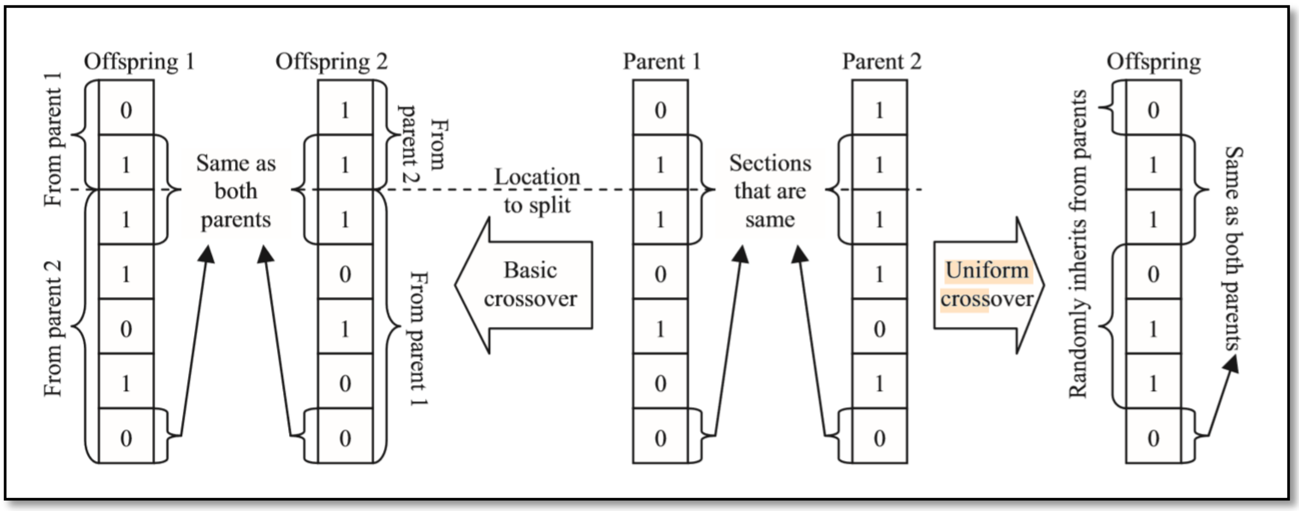


Figure 2-iii: *Single Point Crossover vs Uniform Crossover. Note the random (probabilistic) nature of how uniform crossover selects genes from either parent. Thus creating offspring that are less predictable than traditional crossover strategies (Hu & Di Paolo, 2009)*

2.3.1 Genetic Algorithm Reproduction

Crossover is the primary search methodology for GAs (Deb & Agrawal, 1995). Crossover in GAs (similar to GPs) allows two parents to copy sections of their encoding genetic information to produce two new offspring (Kaya & Uyar, 2011). The literature on GA crossover presents a myriad of options, the following are some of the most common:

Uniform Crossover (UX): Uniform crossover operates on a bitwise basis by probabilistically selecting a parents gene and copying it to an offspring (Poli & Page, 2000). Parameterized version of uniform crossover allows researchers to fine tune the probability of parent genes being selected for the offspring. The tunable parameters also allow uniform crossover to mimic other classical crossover techniques depending on the parameters set (Semenkin & Semenkina, 2012). Uniform crossover is said to be potent in its ability to search all probabilities of parent gene recombination before attributing a probabilistically selected gene to an offspring. Uniform crossover is argued as being quite disruptive particularly when the population size is large (Spears & De Jong, 1995).

Single Point Crossover (SPX): Single point crossover or *one point* crossover splits both same length chromosomal parents at the same randomly chosen index *(See Figure 2-iii),* swapping both segments and attaching the result to the offspring (Kaya & Uyar, 2011; Wright, 1991). In scenarios where slow exploitative search is required, SPX has shown to flourish due to its non-disruptive abilities (Williams & Crossley, 1998).

Two Point Crossover (TPX): Two point crossover, extends on single point crossover, by adding another section along both parent's chromosome to split and reconstitute to their offspring (Kaya & Uyar, 2011). Some crossover strategies may add more split points, but this may prove too disruptive to chromosomes (Kaya & Uyar, 2011).

The search efficacy of crossover techniques varies considerably, and usually depends on the relationship between chromosomal encoding and the fitness domain (Deb & Agrawal, 1995). If a quick rate of chromosomal exchange is desired uniform crossover is an appropriate candidate, conversely if a slow rate of exchange is desired, SPX has been advocated for (Prügel-Bennetf, 2001). The selection of crossover technique can only be perfected through experimentation in the problem domain. Williams and Crossley (1998) argue the selection of a crossover technique such as uniform crossover cannot inform other parameter choices such as mutation rates. What is uncommon in the field of crossover is crossover among individuals with variable length chromosomes (Bentley & Wakefield, 1996). Specialized crossover techniques such as ring crossover have been created to allow for this in constrained heuristic environments (Kaya & Uyar, 2011).

2.3.2 Genetic Algorithms and Symbolic Regression

GAs are not as popular as GPs when it comes to solving SR but a hybrid EA that comprises of both GA and GP has been experimented with to allow the chromosomal linear structure of GAs to be paired with the tree representation offered by GPs (McKay et al., 1995). This pairing of EAs makes way for hybrid evolutionary algorithms.

2.4 Hybrid Evolutionary Algorithms

When it comes to obtaining and archiving underlying tree manipulation operations, GPs fail to carry excess information of how the ordering of tree manipulation strategies result in a phenotype. From an EA lens, GA and GP differ mainly in representation and certain selection techniques due to the initial representation (Zelinka, Oplatkova, Nolle, et al., 2005). Both EA techniques when abstracted present a similar model for exploration and can intuitively be abstracted into a hybrid model containing characteristics of both. Howard and D'Angelo (1995) hybrid approach maximizes on the differences in genetic representation of both evolutionary algorithms known as a genetic algorithm-program (GA-P) where each solution consisted of a string and tree expression. Similar hybrid strategies such as binary tree genetic algorithms used in binary decision trees have been found in the literature (Cha & Tappert, 2009; Chai et al., 1996). Hybrid evolutionary algorithms are not new. For example, techniques such as combining GAs and particle swarm optimization for optimizing multimodal functions have been developed to solve certain problems in AI domains (Kao & Zahara, 2008).

GA-P provides a valid EA to carry out SR. The GA list-based chromosome structure can hold the MTM information required to manipulate the underlying GP tree. Like individuals (tests with tests, or bugs with bugs) can share these MTM strategies along the EP with the goal of outdoing the adversarial competition. Given our research goals require allowing adversarial groups to interact by modifying SR mathematical expressions and using a spec to guide repair back to a valid state, we require an evolutionary system that can handle the interactions between competing individuals. Coevolutionary algorithms are an ideal candidate to handle these kinds of interactions.

2.5 Coevolution

Coevolution refers to any scenario where two or more individuals evolve in a manner where their respective fitness landscapes are closely determined by the evolutionary characteristics of other individuals (Cliff & Miller, 1995). In nature coevolution can be shown to either be cooperative (symbiotic) or competitive (parasitic) (Eiben & Smith, 2015). In competitive coevolution, individuals do not compete against a static fitness function as with traditional evolutionary algorithms, instead, they use each other as a basis for their fitness evaluation, where advances in fitness to a set of individuals is often at the cost to the fitness of other individuals (Cardona, Togelius, & Nelson, 2013; Miconi & Channon, 2006). Consequently,

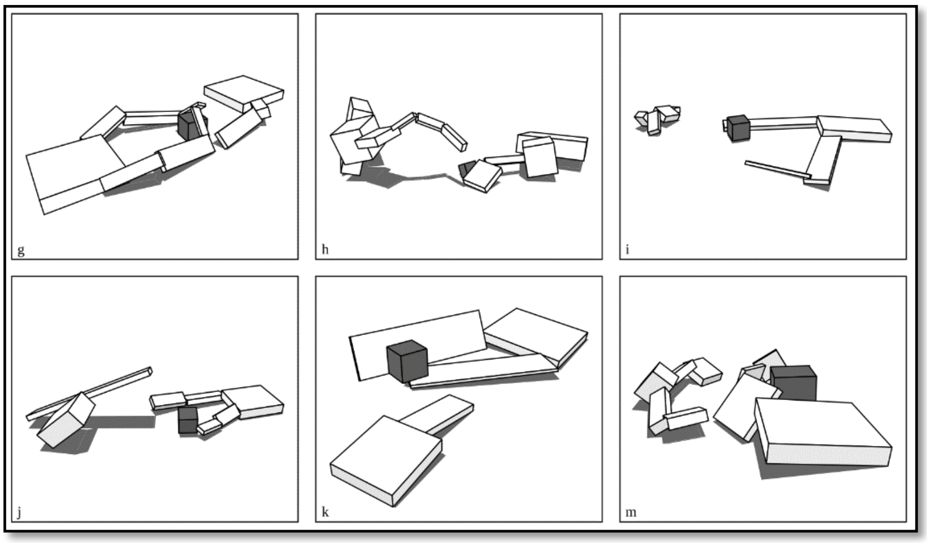


***Figure 2-iv** - An illustration of Sims (1994) 3D simulation of individuals competing to obtain the target object (dark cube). Individuals would configure themselves with a panoply of sensor-effector combinations that would hopefully attain the most fitness (Sims, 1994)*

fitness functions in a coevolutionary realm are simpler as an individual's fitness is based on its interaction with another individual (Rosin & Belew, 1997; Runarsson & Lucas, 2005).

Sims (1994) showcased an interesting application of coevolution where 3D virtual individuals were generated to compete in capturing an objective similar to the game capture the flag. Individuals were paired on opposing sides with the objective placed at the center. The setup included accurate modelling of effects such as gravity and collisions that would affect the choices made by the competing individuals (Sims, 1994a; Sims, 1994b). Fitness was calculated not by simple winning or losing but by the margins in which a win or loss happened. Selection pressure would encourage individuals to win by larger margins in order to claim higher fitness (Sims, 1994). Competition between individuals was carefully considered, if all individuals competed against their opponents an $\frac{n^2-n}{2}$ number of competitions would have to take place which would be infeasible for larger populations. A variety of competitions were formulated ranging from tournament, to random competitions between friends and foes, to all individuals versus the best within the species. The anatomy of the species ensured variation in configuration for both sensors such as limbs that could detect contact, opponent proximity, physical strength of the opposing phenotype and details about the surroundings that would inform subsequent action. Larger phenotypes could elicit physical characteristics such as strength, whilst smaller phenotypes could elicit other physical responses such as greater agility.

The results of this study revealed creatures that would evolve to have two arms would in most cases succeed, by using their arms either to shield access to the target *(see Figure 2-iv:g)* or pass the target from arm to arm rapidly to prevent enemy from getting hold of it *(see Figure 2-iv**Error! Reference source not found.**:i, j, k)*. The most successful individuals were often ones that would gain control of the target object and be able to walk away with it *(see Figure 2-iv:m)*. Larger creatures were able to bat smaller opposing creatures away to protect the target objective (Sims, 1994).

2.6 COEVOLUTIONARY PATHOLOGIES

Coevolutionary pathologies are defined as epiphenomenon that occur during the coevolutionary process that create differences between the expected outcomes of the coevolutionary process and what is seen in reality (Ficici & Pollack, 2004). There are number of coevolutionary pathologies, 5 examples are discussed in this section. These are evolutionary arms races, disengagement, red-queen effect, over-specialization and fitness intransitivity.

2.6.1 Evolutionary Arms Races

When adaptations occur to specific groups of individuals, for instance in a fox and rabbit scenario, if foxes undergo a set of beneficial predatory adaptations, this translates to an increased selection pressure on the rabbits forcing the rabbits to evolve in a positive manner to outdo the advantage gained by the fox. This in turn, places pressure back on foxes to adapt to match or surpass the new advantage obtained by the rabbits. This back and forth creates a positive unstable equilibrium often coined as an evolutionary arms race (Dawkins & Krebs, 1979). Initiating an evolutionary arms race is a desired outcome as each competing party becomes increasingly competent (Lohn, Kraus, & Haith, 2002). The mechanics of initiating an evolutionary arms race are not well understood, often resulting in mediocre-stable states (MSS) where individuals are unable to evolve to a state that can be improved (Ficici & Pollack, 1998; Gomes, Mariano, & Christensen, 2015). In addition, coevolution has the potential to conjure counterintuitive results that can worsen the gap between hypothesized potential and the actual realization (Ficici & Pollack, 2004). Arms races are often the expected outcome in coevolutionary experiments, however in some cases evolution continues along an uninteresting and undesirable path that researchers do not initially seek (Miconi, 2009).

When coevolving species interact, it is difficult to measure if an arms race has occurred or is taking place. Evaluating fitness at the end of an evolutionary run may not paint an encompassing picture of the process as results obtained at the end of a generation could represent the start of a genetic drift from an individual (Cliff & Miller, 1995). Periods of stagnation such as MSS may represent no improvement in individual traits or a tight coupling between individuals such that if one progresses, the other has an adequate response offering no positive net result. These various issues are often referred to as fitness ambiguities. Thorough statistical analysis is often required to gain insight on such latent trends and behaviors (Cliff & Miller, 1995).

2.6.2 Disengagement

Disengagement occurs when idiosyncratic differences in individuals become unidentifiable. This can degenerate further, eventually leading to a genetic drift or stalling where the population's homogeneity inhibits selection from generating new individuals (de Jong, Stanley, & Wiegand, 2007). Disengagement also has the adverse effect of nullifying the value of the fitness function, especially if the fitness function relied on relative delta between individuals to advise selection (de Jong et al., 2007).

2.6.3 Red Queen Effect

During the coevolutionary process, the increased propagation of a useful gene in a set of individuals tends to eventually nullify future benefits as competing individuals will evolve to suppress that useful trait. This is best known as the Red Queen effect (Cliff & Miller). The Red Queen effect innately makes measuring progress more difficult to define and observe (Funes & Pollack, 2000). Miconi (2009) offers the categorization of coevolutionary progress into three distinct groups. *Historical progress* where improvements are made against previous competitors, *local progress* where improvements are made against current competitors, and *global progress* where an individual shows significant performance over the whole competitive space both past and present. Miconi (2009) warns that progress in one realm is not necessarily linked to progress in another realm. *Hall of Fame* and *Dominance Tournament* techniques (*see Section 2.7.1*) insinuate that success in a historical realm often equates to success in a global realm Miconi (2009). More wins do not necessitate better performance (Funes & Pollack, 2000). The nuanced nature of coevolution makes inferring results difficult and rigorous analysis needs to be applied to data to gain useful and consistent insight into the coevolutionary process.

2.6.4 Over-specialization

Over-specialization can occur if individuals find a means to exploit a few strategies in order to gain adequate fitness at the expense of attempting to find a broad set of strategies (de Jong et al., 2007). The coevolutionary process may focus on one or two scenarios that it can benefit from, leaving the rest of the objective in either a low fitness state or completely untouched (de Jong et al., 2007). The individuals are said to have focused or specialized on a subset of objectives.

2.6.5 Fitness Intransitivity

When measuring fitness, head to head competitions in symmetrical environments allow for simple fitness evaluation. For instance, if A competes against B and wins, A is better suited to achieving the goal set out in the experimentation. Miconi and Channon (2006) highlight an inherent issue with intransitive competitions that make understanding the idea of a "better" individual more complex. If A defeats B, but A loses to C and C loses to B, it is hard to infer the fitness of any of the individuals in the competition as their progress and defeats have cancelled out. In order to adequately analyze the outcomes of A, B or C, they must all be considered together or against a common external parameter. Intransitivity is typical of popular games such as rock-paper-scissors (de Jong et al., 2007). No one strategy is best when considered on its own, but in contrast to a defined opponent, the fitness of a given strategy can be adequately compared.

2.7 Coevolutionary Algorithms (Topologies)

Coevolutionary topologies define how individuals can interact with one another in order to be able to justifiably identify fitter and unfit individuals in the context of a competition (Panait & Luke, 2002). For example, in a football tournament, the format of competition can be a league format where each team plays every other team and winners accrue points and losers do not. The team with the most point is deemed the winner. Another common competitive format is the knockout format, where teams are randomly placed into a tournament bracket where only winners progress until there is one last team remaining. Both these formats work in a football setting in determining the best team. Each format has its own characteristics, and depending on the goal, the effectiveness of each format can be critiqued when determining whether it is the best for a given scenario.

In a competitive coevolutionary landscape, topologies define how competing individuals fitness are to be gained and calculated (Panait & Luke, 2002). Topologies aim to minimize fitness attribution pathologies listed in *Section 2.6.5* by creating a logical and clear format. Two commonly used methodologies for evaluating competition superiority are *duel methodology* and the *renaissance-man methodology*. Duel-methodology states that if individual A defeats B a number of times, then A can be said to be superior to B. Renaissance-man dictates that for A to be considered superior than B, A must defeat more individuals than B on average even if B dominates over A in head-to-head competition (Panait & Luke, 2002). In some cases, a mixture of both superiority methodologies may be necessary. From these methodologies we can identify from the literature a set of appropriate the evolutionary topologies to utilize in our study. We outline four commonly used topologies that introduce diverse competition formats that allow for varying tradeoffs. These four topologies will serve as good benchmarks for comparison given the research problem. These topologies are: Hall of Fame (HoF), Round Robin (RR), K-Random Tournaments (KRT) and Single Elimination Tournament (SET),

2.7.1 Hall of Fame

Hall of Fame is a common coevolutionary topology that stores well adapted individuals from previous generations and introduces them at a later stage (Rosin & Belew, 1997). These individuals are then occasionally reintroduced to subsequent generations with the hope that future generations do not forget about historically fit solutions (Popovici, Bucci, Wiegand, & De Jong, 2012). The hope is as the EP progresses, later generations can use information obtained from previous generations to escape local maxima by introducing valuable diversity. An individual's fitness is therefore comprised of its competition with current and previous generations (Szubert, Jaskowski, & Krawiec, 2009). HoF guarantees a strong solution over time but places more focus on previous generations than the current generation as a result, current solutions are pressured to be better than opponents in previous generations and not present or future generations. In some cases, basic coevolutionary strategies can outperform HoF for complex settings due to this inherent issue

(Miconi & Channon, 2006). The concern presented by HoF is that its consideration for historically fit opponents, diverts attention away from the future and may result in unsatisfactory results (Stanley & Miikkulainen, 2002). Stanley and Miikkulainen (2002) note that the continued focus on the past may be counterproductive as the EP has naturally digressed away from past solutions to present ones for an inherent reason. As an EP progresses, it embeds new genes in its chromosome as a result of selection. In order to conservatively progress through the EP, HoF may be a viable coevolutionary topology to mitigate severe digression from past valid checkpoints. However, other strategies such as low mutation rates and conservative crossover methods such as single point crossover can be used if the concern is rapid disruption of the chromosome (Stanley & Miikkulainen, 2002). Miconi (2009) states that in some studies individuals coevolved using HoF as the evolutionary topology were outdone by individuals that did not, citing a potential problem with involving historical elites in future generations. The study revealed that HoF produced individuals that were more adapted at defeating their ancestors than competing against new individuals in future generations. A way to discount the need for "remembering" is if all elements of a given genome are still required in subsequent generations (de Jong et al., 2007).

HoF as an archival strategy has similarities to dominance tournament topology. Dominance tournament works by obtaining the fittest individual of a current generation only if it defeats all individuals retained in memory (Ficici & Pollack, 2003). It prevents a phenomenon called cycling where the evolutionary process perpetually moves through a similar set of solutions instead of proceeding with exploration (Eiben & Smith, 2015; Miconi & Channon, 2006). Dominance tournament has the benefit of reducing comparisons and ensuring winners are well defined (Stanley & Miikkulainen, 2002). In order to reduce the computational cost of sampling all individuals a technique known as opponent or *shared sampling* can be used to select a representative few individuals whose genetic traits are of either superior or greatly diverse characteristics (Rosin & Belew, 1995). This may mean selecting stronger individuals from previous generations instead of using weaker more homogenous individuals from a current generation. Competitor selection in HoF can be further optimized through the use of shared sampling allowing the EP to select individuals with the best competitive shared fitness within the current sample and selecting individuals from previous generations that outperformed the current best individual with regards to certain opponents (Lubberts & Miikkulainen, 2001). This enables a more representative and diverse solution set capable of challenging opponents to a greater degree consequently reducing the total number of individuals competing.

2.7.2 Round Robin

Round Robin allows every individual in a given generation to compete against every other individual in the same generation (Jaśkowski et al., 2008a). RR provides an adequate benchmark as games between individuals are evened out, but the number of games may make it computationally infeasible even for small population sizes (Gómez, Garcia, & Silva, 2005; Jaśkowski, Krawiec, & Wieloch, 2008b). Stanley and Miikkulainen (2004) advocate for use of carefully selected competitions to lessen the computational burden of evaluating all competitions. Round robin tournaments require $\frac{n(n-1)}{2}$ to be played (Jaśkowski et al., 2008a). RR has been used in cases such as the iterated prisoners dilemma as its methodology requires a duel based methodology between individuals (de Jong et al., 2007). RR contains similarities to other strategies such as last elite standing that pits the current champion of a population against each opposing individual within that population (Cliff & Miller, 1995; Miconi & Channon, 2006). Last elite standing is generational and there is uncertainty on whether an elite individual selected truly is the fittest, or only made it through as a result of lucky circumstances (Cliff & Miller) (Cliff & Miller, 1995).

2.7.3 K-Random Tournament (Opponents)

K-Random Opponents (KRT) allows individuals to play with k randomly selected individuals from the population (Hiew et al., 2017). KRT typically rejects self-play and avoids repeating of games to ensure maximal coverage of the population space (Tan, Teo, & Lau, 2008). The computational cost of KRT can be scaled up or down depending on the input value of k. In cases where RR topologies may be unnecessary from a computational and evolutionary standpoint, the value of k can be used to scale the number of individuals competitions. Only kn games are ever played in any generation (Jaśkowski et al., 2008a). In the literature various k values have been experimented with. Jaśkowski et al. (2008a) experimented with values from $1\ to\ 10$, while researchers like Tan et al. (2008) experimented with larger values ranging between 10 and 90 that incremented in steps of 10 from generation to generation, after which an RR strategy was implemented for their work with multi-objective three dimensional problems. KRT has shown use in the multi-objective optimization realm led by research by Tan et al. (2008) as well as in evolving the Blondie24 checkers player (Popovici et al., 2012).

2.7.4 Single Elimination Tournament

Single Elimination Tournament (SET) is a popular competition format that extends beyond coevolution. Most sports such as tennis or football utilize a SET strategy to allow individuals to climb up a competitive ladder, until the best individual is chosen (Stanton & Williams, 2011). Winners of SE tournaments are decided within $N - 1$ tournaments unlike in the RR topology where roughly N^2 tournaments are required in each generation (M. P. Kim, Suksompong, & Williams, 2017). Individuals in the tournament are first randomly paired in brackets where the winner of one bracket faces the winner of a different bracket (Jaśkowski et al., 2008a). Losers are eliminated from the tournament until there is a single winner that emerges. In the case of a tie, a winner is selected randomly (Panait & Luke, 2002). Individuals are randomly paired so as to mitigate the unwanted effects of tournament fixing, where careful seeding of individuals can ensure predictable outcomes (M. P. Kim et al., 2017).

Seeding plays a crucial role as the winner of a given tournament bracket could have arisen solely from seeding biases or fortune by placing a significantly fit opponent against unfit opponents (M. P. Kim et al., 2017). SET typically disallows self-play (Tan et al., 2008). From a superiority lens, SET may seem to promote duel-methodology, if A defeats B and B defeats C, then A must defeat C. This assumption is only valid under a strong transitivity assumption, in cases where this transitivity dynamic does not hold, results from SET can be hard to adequately interpret (Panait & Luke, 2002). SET has the problem that in a population of N individuals, some will play more games than others. Fitness calculation in SET should be scaled based on the number of games an individual has played as by definition, an individual who has played the most games should attain most fitness (Panait & Luke, 2002). SET has been used in games such as Tic-Tac-Toe (Tan et al., 2008).

3 RESEARCH METHODOLOGY

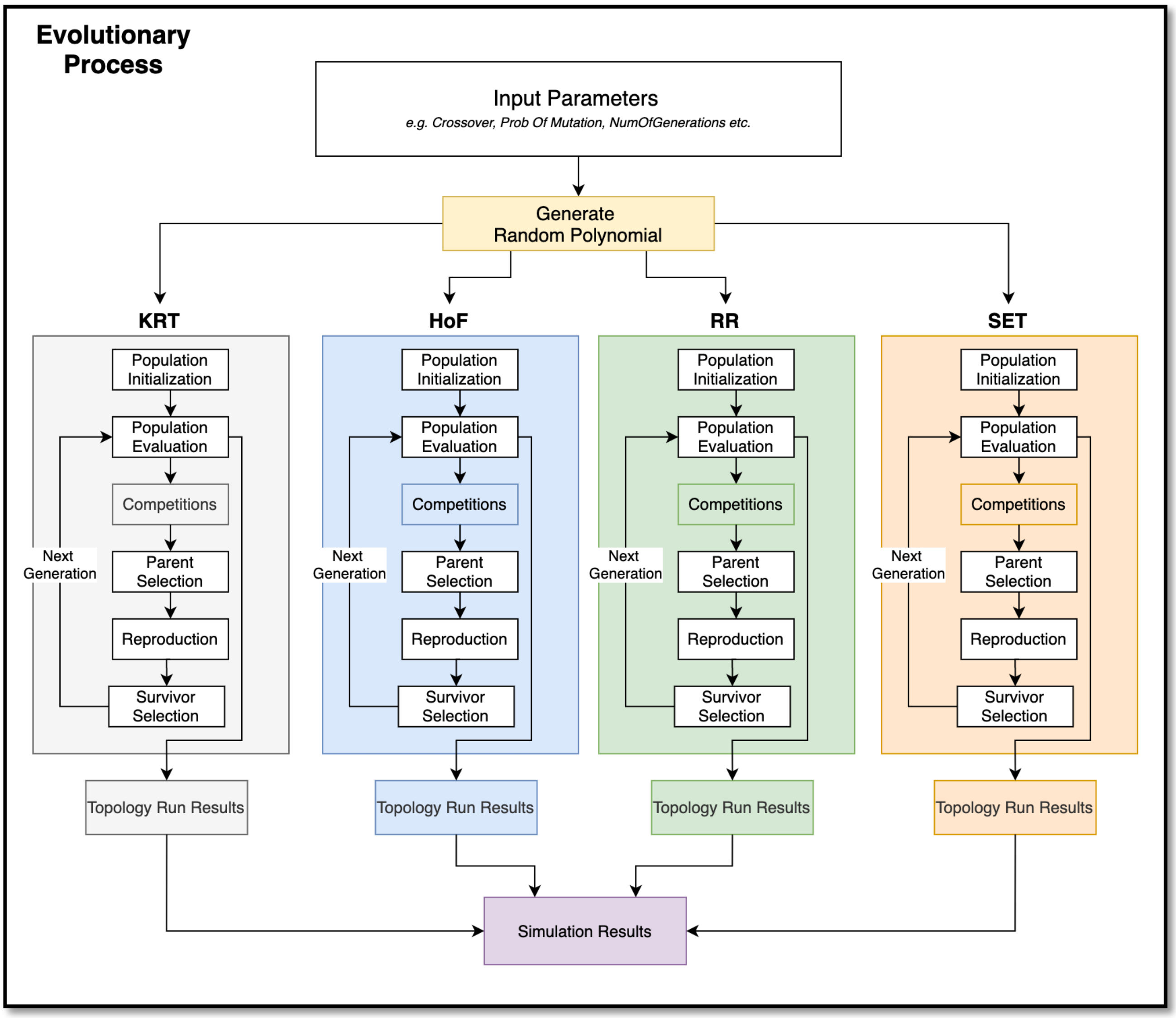


Figure 3-i: *Overview of an entire simulation. A simulation generates a random polynomial that acts as the input program (IP). This is copied to each topology where competition begins. The competition of each topology varies slightly. After competing individuals compete, individuals proceed to parent selection, reproduction and survivor selection. Individuals are then evaluated based on meeting certain criteria. If the max number of generations is reached, the topologies complete. Topology results are then generated and all results from the different topologies are combined into a single simulation result.*

The experimentation will utilize hybrid GA-P structures *(see Section 2.4)* to represent both bugs and tests. The chromosomes of the individuals will be list based MTM strategies that when expressed will manipulate the underlying phenotype represented as a mathematical tree structure. Coevolution *(see Section 2.5)* will be used to guide the EP via the 4 coevolutionary topologies *(see Section 2.7)*. Each topology will be run as a separate simulation and the results compared to discuss strengths and weaknesses of the various topologies for this kind of experimentation. This section details the entire structure of the coevolutionary process and its components.

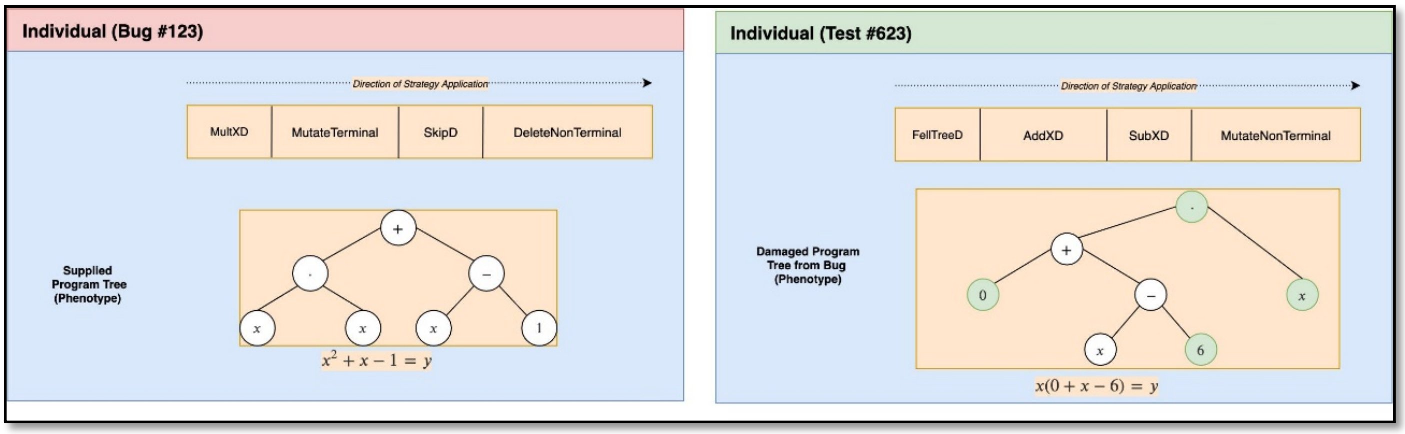


***Figure 3-ii**: Bug and Test Anatomy. Both bugs and tests have identical anatomical structure. A chromosome to store tree manipulation strategies, and a phenotype that stores the expressed chromosome. The bug on the left has received the input program as its phenotype*

3.1 Evolutionary Process Overview

The evolutionary process (EP) (sometimes referred to as the simulation in this research) outlines the events from the start of the simulation (run) until the end. *Figure 3-ii* illustrates an overview of a single simulation.

A simulation is given a set of predetermined input parameters that dictate certain aspects of the simulation such as the initial population size, the number of generations, the crossover technique to use etc. The simulation begins by generating a random polynomial to act as the input program (IP) in question. The input parameters dictate the complexity limits of the generated polynomial. The input program and input parameters are copied to each topology. Each topology begins by randomly generating a set of individuals. The *EachIndividualCount* parameter dictates how many individuals are to be generated for each competitive group (bugs and tests). The generated initial population will undergo evaluation, and if the termination criteria set by the input parameters are met, we can terminate. Otherwise each topology begins its unique competition phase. After completion of the competition phase, all competitors will have been attributed fitness depending on their performance. The resulting population will undergo selection, reproduction and finally survivor selection before the next generation is created. The EP will terminate when the maximum number of generations has been reached or when marginal progress in the simulation is being made (Eiben & Smith, 2015). More details regarding the input parameters are discussed in *Section 3.11*.

3.2 The Individuals (Solutions)

The EP will comprise of two groups of adversarial individuals, termed bugs and tests.

Bugs: The bugs primary objective is to take in the IP supplied as an input parameter and apply a set of MTM strategies *(See Section 3.8)* to maximize the error from the original function, therefore "damaging" the function. The bugs aim to find a set of strategies embedded in their chromosome that maximize the error from the original function.

Tests: The tests prime objective is to take the damaged function from the bugs and find a set of MTM strategies that remedy the program back to the original input program. The tests have no idea of what the input program was, all they are provided with is a spec of various domain to codomain mappings of the IP. Tests will need to find the most efficient set of MTM strategies to undo the damage from the bugs with only the spec as a guide.

Error! Reference source not found. shows the anatomical structure of a bug and a test. The list at the top of each individual represents a chromosome that contains MTM strategies (genes) used to manipulate the underlying tree. The tree is not provided to either individual until competition time. The bugs and tests only differ in how their fitness is evaluated.

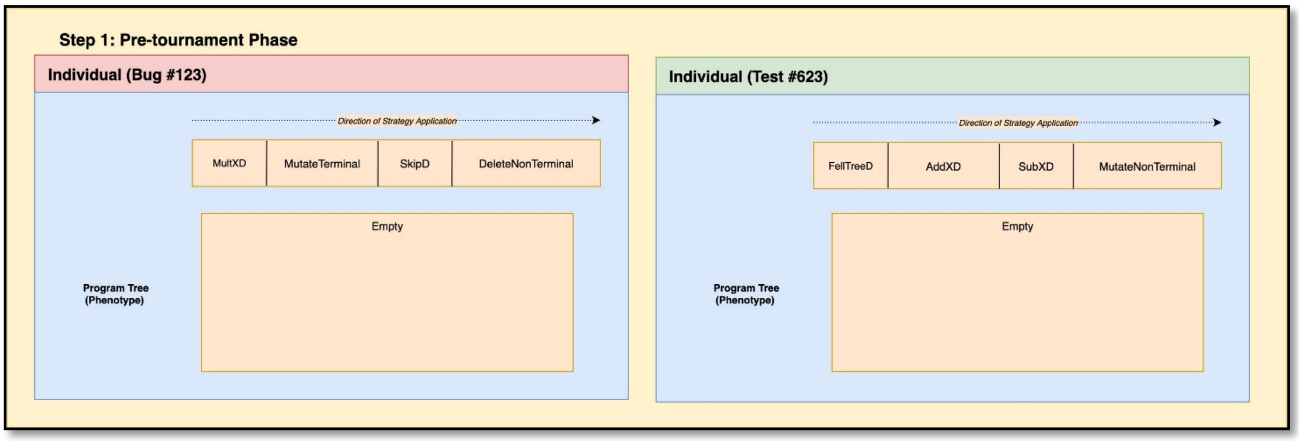


***Figure 3-iii**: Pre-competition phase: Before any competition begins, both tests and bugs contain no tree to manipulate.*

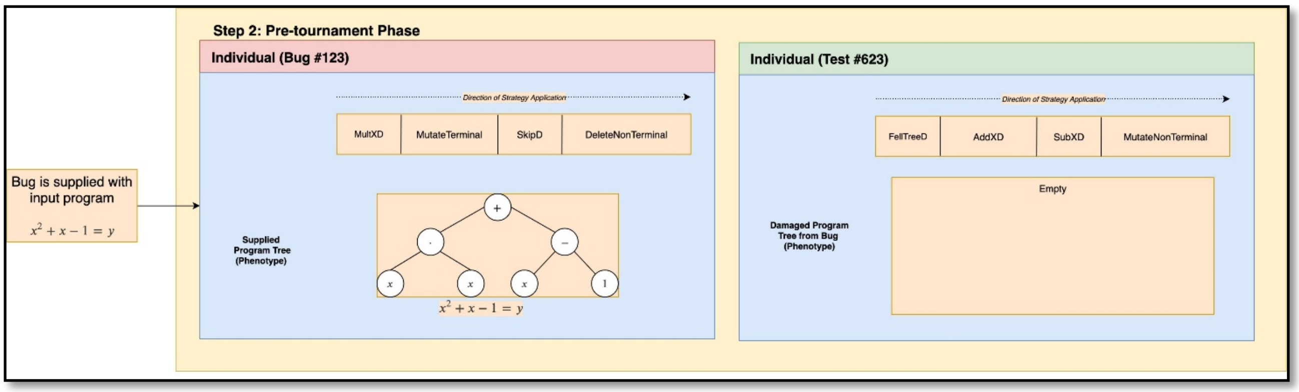
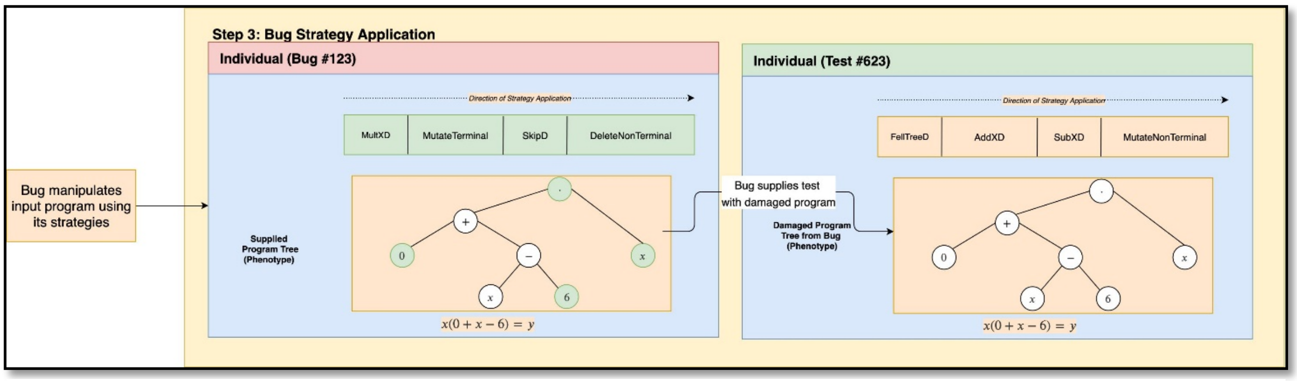

Bug is supplied with input program
$x^2 + x - 1 = y$
Step 2: Pre-tournament Phase
Individual (Bug #123)
Direction of Strategy Application
MultXD
MutateTerminal
SkipD
DeleteNonTerminal
Supplied Program Tree (Phenotype)
$x^2 + x - 1 = y$
Individual (Test #623)
Direction of Strategy Application
FellTreeD
AddXD
SubXD
MutateNonTerminal
Empty
Damaged Program Tree from Bug (Phenotype)

***Figure 3-iv**: Pre-tournament phase: Before the competition begins, the bug is supplied with a copy of the input program to damage. The test still contains no program as it awaits the damage program from the bug.*

Bugs are rewarded for MTM strategies that achieve maximal error from the input program, and the tests being rewarded for developing strategies that minimize the error caused by the bug.

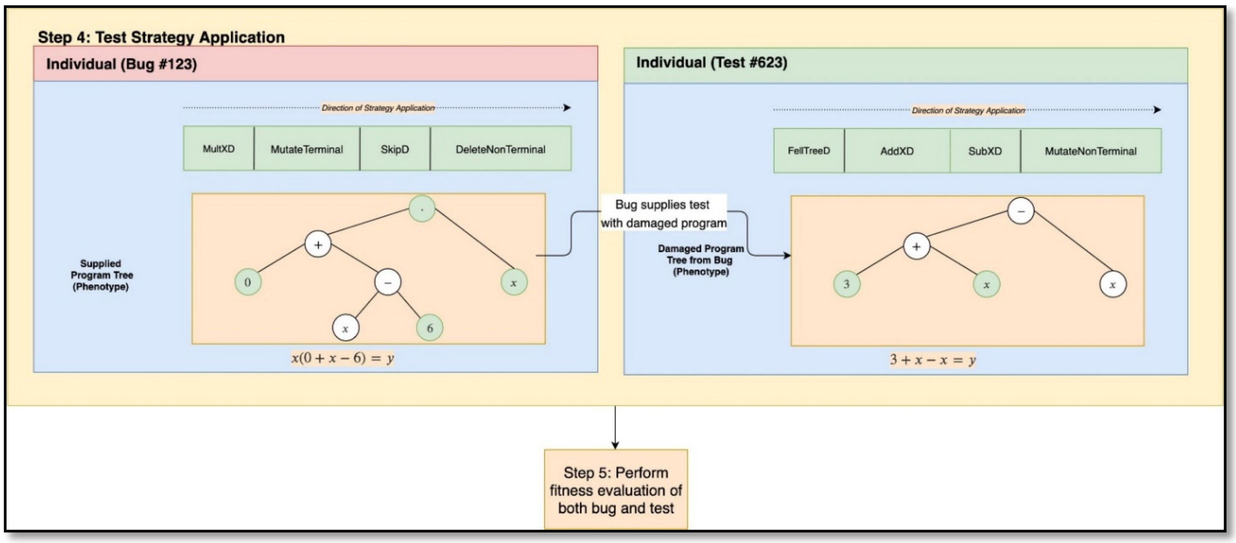
3.3 COMPETITION OVERVIEW

In the pre-tournament (pre-competition phase) shown in **Figure 3-iii**, both bugs and tests contain no program tree. The bug first receives the IP as shown in **Figure 3-iv**. The bug loads the IP into its program tree and begins to apply the MTM strategies embedded in its chromosome.



Step 3: Bug Strategy Application
Individual (Bug #123)
Individual (Test #623)
Direction of Strategy Application
MultXD
MutateTerminal
SkipD
DeleteNonTerminal
FellTreeD
AddXD
SubXD
MutateNonTerminal
Bug manipulates input program using its strategies
Supplied Program Tree (Phenotype)
Bug supplies test with damaged program
Damaged Program Tree from Bug (Phenotype)
$x(0 + x - 6) = y$
$x(0 + x - 6) = y$

Figure 3-v: *The bug manipulates the input program producing a new tree that is then passed on to the test. Notice the change the bug made to the input program. The test then embeds this new program as its phenotype, ready to apply its strategies.*



Step 4: Test Strategy Application
Individual (Bug #123)
Individual (Test #623)
Direction of Strategy Application
MultXD
MutateTerminal
SkipD
DeleteNonTerminal
FellTreeD
AddXD
SubXD
MutateNonTerminal
Supplied Program Tree (Phenotype)
Bug supplies test with damaged program
Damaged Program Tree from Bug (Phenotype)
$x(0 + x - 6) = y$
$3 + x - x = y$
Step 5: Perform fitness evaluation of both bug and test

Figure 3-vi: *Test strategy application. The test applies its strategy on the damaged program from the bug. Fitness is then calculated for both individuals. Fitness is awarded proportionately to how well each individual performed their objective*

The resulting tree is shown in *Figure 3-v*. After the bug applies its MTM strategies it supplies the competing test in this case *Test #623* with the program it has damaged. The test applies its MTM strategies to the damaged program as shown in *Figure 3-vi* in the hope of recreating the original IP.

3.4 The Input Program

The Input Program (IP) as mentioned in *Section 3.1* is randomly generated at the start of the simulation. For the purposes of this example, we shall use the polynomial function in *Equation 3* as the randomly generated IP. The input parameters shown in *Figure 3-i* will contain additional parameters on how to go about evaluating the IP. These additional parameters are the *range* and *seed* value.

$$x^2 + x - 1 \qquad (3)$$

Spec			
Range	Seed	Independent Variable (x)	Dependent Variable (y)
5	-2	-2.0	1.0
		-1.0	-1.0
		0.0	-1.0
		1.0	1.0
		2.0	5.0

***Table 3-i**: Spec of the input program shown in Equation 3. The range dictates over how many values is the spec evaluated, and the seed is the starting point of the evaluation. The range used in this table is just for illustration purposes. A larger range is used during the simulation.*

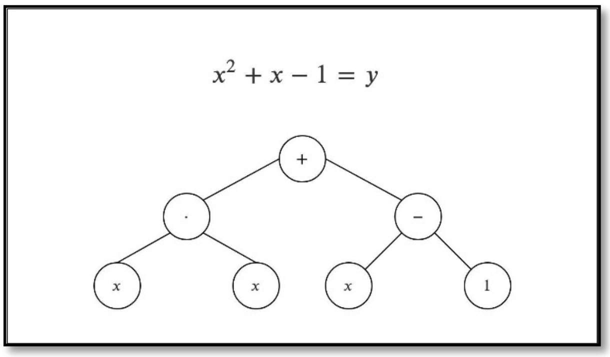


***Figure 3-vii**: Tree representation of the input program supplied in Equation 3*

The range is the number of continuous independent variables values in which the function will be evaluated beginning with the seed as shown in *Table 3-i*. The function evaluation across the range beginning at the seed creates an independent to dependent variable mapping which we refer to as the specification *(spec)*. This spec will be used to calculate how well either kind of individual performed.

In *Figure 3-vii* we see a tree representation of the sample input program shown in *Equation 3*. In-order Depth First Traversal is used to iterate through the items in the tree and generate a mathematical polynomial expression of the items in the tree.

3.5 FITNESS CALCULATION

3.5.1 Ratio-RMSE Fitness

Ratio fitness involves calculating fitness based on the ratio of how well a competing test resolved the errors introduced by the bug back to the spec.

$$\text{RMSE(b)} = (b, s) = \left(\frac{1}{n} \sum_{i=seed}^{n} (s_i - b_i)^2 \right)^{1/2} \qquad (4)$$

Equation 4 shows that the bug's error uses the standard RMSE. s and b and are the output mathematical expressions from the spec and bug respectively. s_i and b_i represent the dependent values obtained from running the spec and bug mathematical expressions against the independent variable x at position i. i begins at the seed value and is incremented until it reaches n . n represents the range of evaluation outlined in the spec. To summarize, the Ratio-RMSE fitness evaluates the summed squared difference between the dependent variables generated from evaluating the spec and bug mathematical expressions, normalized against the range of evaluation.

$$\mathrm{RMSE(t)} = (t, s) = \left(\frac{1}{n}\sum_{i}^{n}(s_i - t_i)^2\right)^{1/2} \tag{5}$$

Similarly, *in Equation 5* we see the standard RMSE for the test where t now represents a test.

$$ratioFit_{test} = \begin{cases} 1 - \dfrac{\mathrm{RMSE(t)}}{\mathrm{RMSE(b)}}, & \mathrm{RMSE(t)} < \mathrm{RMSE(b)} \\ 0, & \mathrm{RMSE(t)} \geq \mathrm{RMSE(b)} \end{cases} \tag{6}$$

Equation 6 shows how the test's fitness is calculated. The equation evaluates both individual's error, and if the test's error is less than that of the bug, meaning it managed to repair the program *(at least partially)*, the test gains fitness based on how well it reduced the error. If it can reduce the error to 0, it gains maximal fitness of 1, conversely, if the test fails to reduce the error created by the bug, it receives a minimal fitness of 0.

$$ratioFit_{bug} = 1 - ratioFit_{test} \tag{7}$$

The bugs fitness is simply the remainder of what the test failed to repair shown in *Equation 7*. Ratio-RMSE Fitness calculation is a direct application of the zero-sum nature of competitive coevolutionary fitness functions.

Ratio-RMSE Fitness has an unintended pathology and flaw that made us switch to an objective based fitness strategy. With Ratio-RMSE Fitness the test's fitness is only attributed based on the error created by the of the bug. Initial runs with Ratio-RMSE produced counter-intuitive results mentioned in *Section 2.6.1*. We observed after some number of generations, the bugs learned that they can achieve maximal fitness by adopting the role of the test which involved minimizing the error as much as possible. If the bug doesn't perform well, and produces minimal error, the test will have to work hard to come up with a set of strategies to further minimize the error. In some extreme examples, the bug chose to avoid damaging the input program completely by adopting multiple *SkipD* strategies *(see Section 3.8 for list of strategies)* in its genome leaving the test with the impossible task of fixing something that has not been damaged.

To minimize this unexpected behavior, we introduced thresholds that force both bugs and tests to play their respective games.

3.5.2 Threshold-RMSE Fitness

Threshold-RMSE Fitness introduces thresholds that both bug and test must exceed before they can begin to accrue fitness. In *Section 2.6.5* we highlighted the issue of fitness intransitivity, and one remedy was to establish an external parameter that we could gauge the relative fitness of competing individuals. In our case, this prevents bugs from attempting to outplay the tests by converging on a perfect solution *(the input program)* and prevents the tests from producing solutions that are marginally better than that of the bugs. Threshold-RMSE sets two main thresholds that scale with the IP.

Ideally, the threshold belonging to the test is placed at the point of truth (where error is zero), this ensures that tests gain maximal fitness only when the error is 0 *(fully repair the damaged program to the standards outlined in the spec)* and reduced fitness as they approach zero error. Similarly, a bug only begins to attain fitness once they positively cross their error threshold and

Spec					
Range	Seed	(x)	(y)	$C_b(3)$	$C_t(2)$
5	-2	-2	1	3	2
		-1	-1	-3	-2
		0	-1	-3	-2
		1	1	3	2
		2	5	15	10

Table 3-ii: *Spec shown with added threshold coefficients values for the bug $C_b(3)$ and test $C_t(2)$. Assume we are using the input program from Equation 3. The coefficients 3 for C_b and 2 for C_t are multiplied against the dependent-variable (y) to establish adequate thresholds.*

lose fitness if they fail to exceed their threshold. Threshold-RMSE fitness pins the individuals against an external metric (thresholds) that emphasizes the dichotomy between competing individuals. This way we are guaranteed to ensure that tests are evolving to beat their threshold producing very fit individuals, similarly bugs are doing their best to attain strategies that produce maximum error. This is similar to games where competing individuals are judged based on how each competitor is compared to the standard that they were to be measured against. If an individual exceeds their standard by a greater margin they triumph over their competitors. Threshold-RMSE decouples direct competition between the test and bug allowing for more specialized competition between both parties. Threshold-RMSE also has the positive effect of being a selection-pressure mechanism. For example, if you want to maximize the chance of creating bugs that create significant errors to the program, one could set a very high threshold for the bugs. This forces bugs that find a way to create such error to make it through to the selection process.

3.5.3 Calculating Appropriate Thresholds

Thresholds for competing individuals need to be setup at the start of the simulation. At the start of the simulation during the spec evaluation of the IP, two separate scalar values are provided through the input parameters for the tests and the bugs to use to calculate their thresholds. These scalar values or threshold coefficients tune the performance expectations of both tests and bugs in the simulation. For example, If the bug's threshold coefficient C_b is 3 and the test's threshold coefficient C_t is 2, we obtain the modified spec shown in *Table 3-ii*.

$$IP \rightarrow y = x^2 + x - 1$$

$$Bug \rightarrow b = -x^2 + 3x$$

$$Test \rightarrow t = x^2 + 1$$

Fitness Evaluation					
(x)	(y)	$C_b(y,3)$	$C_t(y,2)$	$b(x)$	$t(x)$
-2	1	3	2	-10	5
-1	-1	-3	-2	-4	2
0	-1	-3	-2	0	1
1	1	3	2	2	2
2	5	15	10	2	5
RMSE	**0.00**	**4.81**	**2.72**	**5.31**	**2.45**
Fitness				**0.09**	**0.10**

Table 3-iii: *Threshold-RMSE fitness evaluation for bug and test. The thresholds for both bugs and tests individuals are shown as* $C_b(y,3)$ *and* $C_t(y,2)$*. The y variable that comes from evaluating the IP, is multiplied against the constant value passed in the threshold allowing us to obtain the per-threshold values shown in the table.*

The values obtained for the threshold multipliers are now the relative thresholds for bug and test. In our case, if after the competition between bug and test, the bug obtained a mathematical expression of $-x^2 + 3x$ and the test attempted to repair the bugs damaged expression and got $x^2 + 1$, *Table 3-iii* shows how fitness will be calculated.

Table 3-iii shows the Threshold-RMSE fitness evaluation for the given bug and test mathematical expressions above. Note that in Threshold-RMSE fitness, the range is from -1 to 1 but these values mean slightly different things for bug and test depending on the threshold coefficient.

For each value of x, we calculate the corresponding threshold coefficient value for the bug and test by multiplying the threshold coefficient (in this case, 3 for bugs, 2 for tests) against the spec's dependent variable y. To calculate the RMSE of the bug's and test's threshold, we use the RMSE formula shown in *Equation 3* and compute the values for the bug's and test's threshold.

If the bug surpasses its threshold coefficient like in our case, the bug attained an RMSE of 5.31 and the threshold was 4.81. The bug will therefore gain positive fitness as its RMSE (5.31) did exceeded the threshold value (4.81) that was set for it. The test got an RMSE value of 2.45 which was less (marginally better, as the test threshold is an upper limit it should not exceed) than its threshold of 2.72, giving it slightly positive fitness. This form of fitness measurement works well in our case, however is may be less intuitive than the ratio-RMSE fitness strategy introduced in *Section 3.5.1*. In *Section 3.5.4* we provide general heuristics used to interpret fitness results.

$$tFit_{bug} = \begin{cases} \dfrac{RMSE(b)}{RMSE(C_b)} - 1, & RMSE(C_b) > RMSE(b) \\ 1 - \dfrac{RMSE(C_b)}{RMSE(b)}, & RMSE(C_b) \leq RMSE(b) \end{cases} \quad (8)$$

Equation 8: *Threshold-RMSE fitness formula for bugs*

$$tFit_{test} = \begin{cases} 1 - \dfrac{RMSE(t)}{RMSE(C_t)}, & RMSE(C_t) \geq RMSE(t) \\ \dfrac{RMSE(C_t)}{RMSE(t)} - 1, & RMSE(C_t) < RMSE(t) \end{cases} \quad (9)$$

Equation 9: *Threshold-RMSE fitness formula for tests*

These fitness functions in *Equation 8 and Equation 9* show the formulas for calculating threshold-RMSE fitness for both bugs and tests respectively. We see from both equations that fitness values can to range between -1 and 1 depending on how well the bug or test performed. The threshold coefficient can be scaled to fit the desired outcome of the EP, in our case we enforce a value of 10 for bugs meaning the bug's error should be at least 10 times greater than the value of the input program's dependent variable y for any given x. Similarly, if we set a threshold coefficient value of 1 for the test, the test can only gain maximal fitness if it exactly matches the input program for the given range.

Equation 8 states that the threshold-RMSE fitness of the bug ($tFit_{bug}$) is the ratio of the RMSE of the bugs damaged program $RMSE(b)$ divided by the RMSE of the bugs threshold $RMSE(C_b)$ subtracted by 1 if the RMSE of the bugs threshold is greater than the RMSE of the bugs damaged program. This ensures that if the bug failed to exceed its threshold, it would attain negative fitness. Conversely, a bug obtains positive fitness if the RMSE of the bugs threshold $RMSE(C_b)$ is less than that of the bugs damaged program $RMSE(b)$. This means the bug was able to exceed the threshold and apply a greater error than that stipulated by the threshold.

Equation 9 shows how the threshold-RMSE fitness for the test is calculated ($tFit_{test}$). If the RMSE of the test's fixed program $RMSE(t)$ is less than that set by the test's threshold $RMSE(C_t)$, then the test gains positive fitness. This means that for tests the threshold is an upper-bound that should not be exceeded. If exceeded, the test gains negative fitness as it is clear that it is introducing more error. However, if the test's fixed program reduces the error introduced by the bug such that the error is less than that allowed by its threshold, then the test gains positive fitness. Tests can gain a maximal fitness value of 1 if and only if it completely undoes any error introduced by the bug and applies a set of strategies that reduce that error to 0.

It is therefore important to note that bugs aim to exceed their threshold in order to gain positive fitness, whereas tests use their thresholds as cautionary upper bounds that if exceeded, result in negative fitness for the tests.

3.5.4 Interpreting Results using Threshold-RMSE

One drawback of the Threshold-RMSE is that bugs and tests fitness values are no longer tightly coupled to each other. As mentioned earlier, the threshold values set for the bug and tests are what the bugs and tests have to compete against respectively. This means that if the threshold for a test was 1.1 and that of a bug was 10, if a bug obtained a fitness score of 0.5 and a test obtained a fitness score of 0.1. It means that the bug improved against its set threshold by increasing the error on the input program such that the error introduced was about twice as large as that determined by the bugs threshold. The value of 0.1 for the test means that the test applied a set of strategies that were able to reduce that error introduced by the bug to within 1.1 times that of the original spec.

In this case it is fair to state both individuals gained positive fitness as the bug exceeded its threshold, and the test did not exceed its threshold. If, however we set the threshold for the test to 1.0, meaning the test's threshold had to match up exactly with the spec, the test has to apply a set of strategies that always fix the damaged program with zero error. Any deviation causes negative fitness for the test.

3.5.5 Handling Edge Cases & Penalization

The inclusion of divide as a viable operator that either tests or bugs can use brings about the question of what happens in a situation where an individual divides by zero. Keijzer (2003) outlined research where researchers would typically ignore or assign a specified fitness to individuals that attempted this. Although this may be viable in certain contexts, we decided to implement a more fine-grained approach to how we handle divide-by-zero scenarios.

If an individual's strategy generates a tree that divides by zero, a custom penalization method known as steady penalization is used. Steady penalization penalizes an individual depending on whether an individual's program tree explicitly divides by zero, for example $\frac{1}{0}$ or $\frac{3x}{2-2}$, or an individual only divides by zero when the spec is being evaluated and traverses an independent variable that causes the evaluated expression to divide by zero, for example $\frac{1}{x}, x = 0$ or $\frac{3x}{x-2}, x = 2$. In the former case, the individual is given the minimal fitness value of -2 which is less than the bottom threshold of -1. This is to force selection to avoid individuals with strategies that cause explicit divide by zero errors. In the latter scenario where the evaluation of the spec on a given independent variable causes the individual to divide by zero, 10% is deducted from the individual's fitness as a cautionary warning. This penalty is made small intentionally to prevent significant harm to the individual's chances of surviving selection as the expression is still valid for most cases.

3.6 POPULATION INITIALIZATION

Population initialization refers to the seeding of individuals at the start of the EP. Population initialization is identified as being an important part of any evolutionary algorithm as the rate at which subsequent individuals in later generations converge to a viable solution is greatly affected by the initial population (Rahnamayan, Tizhoosh, & Salama, 2007). Work by Stanley and Miikkulainen (2004) advocated for a technique known as complexification. Complexification ensures the initial random population is kept simple on a chromosomal level to give more room for exploration as the EP progresses. Similarly, Searson et al. (2010) suggested that the initial population be instantiated with a small number of tree elements to prevent initial bloat. Poli et al. (2008) maintains a neutral strategy known as Ramped-Half-and-Half initialization, which is common with GPs. Ramped-Half-and-Half initialization allows half the population to be initialized at capacity, and the other half with room to grow. Ramped-Half-and-Half can the drawback of creating large, algebraically expanded terms further contributing to bloat (Davidson et al., 2003). Given the hybrid nature of the individuals in our research, the primary computational cost of evaluating an individual pertains to evaluating the underlying mathematical program tree against the spec.

The hybrid GA-P approach may require experimentation with various population sizes, as GAs tend perform better with larger populations (Howard & D'Angelo, 1995). Howard and D'Angelo (1995) note that population sizes greater than 200 have marginal return on retrieving optimal solutions. A good balance would be to increase the number of generations if the population sizes are kept low. This strategy is advocated by Searson et al. (2010) who experimented with a relatively small population size of 20, but relatively large number of generations at 100. Small populations may fail to provide adequate genetic diversity that may enable the EP to converge to an optimal solution (Williams & Crossley, 1998). Code bloat has a detrimental effect on the validity of large generation evolutionary simulations. High bloat of introns has shown to limit the effectiveness of multi-generational evolutionary simulations, having marginal utility after 50 generations (Davidson et al., 2003).

For this study a moderate population number of 64 will be used, with many generations (300) allowing for adequate time for selection to take effect. In addition, strategies that attempt to generate trees to add to an individual's tree, will be limited to trees of depth one.

3.7 Generations

The number of generations within an EP will be determined by how well the tests are performing at consistently converging on strategies that fix the errors brought about by bugs. The EP is given freedom to run for a maximum of 300 generations before a solution will be evaluated. From a fitness performance perspective, if a test is found to contain the maximum fitness possible (1), the simulation should be able to terminate, however in our experimentation, we shall allow the EP to continue to observe how individuals interact across longer generations. If the average fitness of all tests in 50 subsequent generations exceeds a given threshold, the EP can terminate prematurely as a good set of strategies have been located in tests that can adequately remedy the buggy programs.

3.8 Strategies

The genes in any individual within the EP are a random mix of deterministic and non-deterministic strategies used to manipulate an individual's program tree. Each individual has a list of MTM strategies that make up their chromosome. When an individual's chromosome is expressed, the strategies are applied to the underlying mathematical program tree generating a new tree as shown in *Figure 3-v*. The new tree can be evaluated for fitness using Threshold-RMSE-Fitness to attribute the individual carrying that chromosome appropriate fitness. A comprehensive description of each strategy is shown in *Table 9-i* in the Appendix.

The strategies that contain *'X'* ensure that the independent variable x will be involved in the tree manipulation. All strategies are strongly typed and therefore manipulate the tree and leave it in a valid mathematical state. Every result after a strategy is applied can be expressed as a mathematical operation.

3.8.1 Deterministic and Non-Deterministic Strategies

Unlike traditional GAs that utilize binary strings as their chromosomes that are deterministic in their expression, we utilize a mix of deterministic and non-deterministic strategies to enable a variety of exploitative and explorative options for individuals. Strategies suffixed with *'D'* represent deterministic strategies, for example the strategy *AddXD* will take the current tree and add x to it. An individual can contain many of the same strategies as part of its chromosome. Deterministic strategies allow for greater exploitation than exploration. If a bug chooses to increase its RMSE it may attempt to utilize the *MultXD* strategy to multiply its tree by the independent variable x giving a predictably larger RMSE.

Non-deterministic strategies are strategies suffixed with *'R'*, and when applied independently to the same tree, *may* give slightly different outcomes due to their stochastic nature. For instance, the application of the strategy *MutateTerminalR* may change an equation such as $\frac{x^3}{2}$ to $\frac{x^3}{8}$ or $\frac{3^3}{2}$ as the mutation of a given terminal will randomly select from a list of available terminals. Non-deterministic strategies are meant for more exploration than exploitation, this stochastic nature parallels the effects of internal mutation.

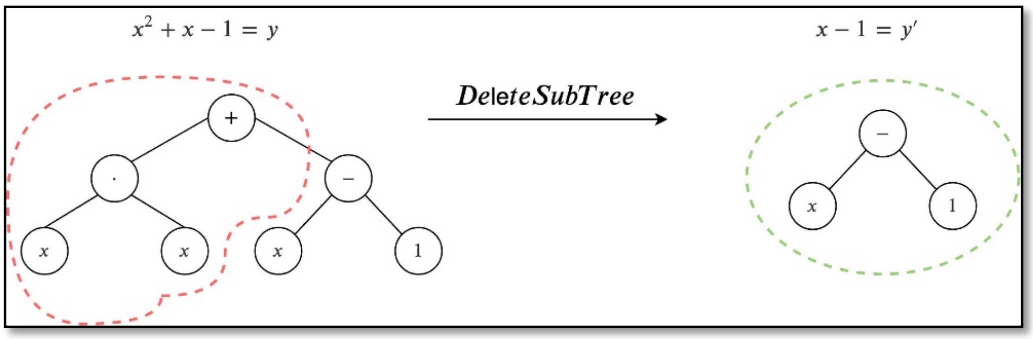


Figure 3-viii: *DeleteSubTree (DeleteNonTerminal) - This operation deletes a portion of the sub-tree leaving the remaining portion in a valid mathematical state.*

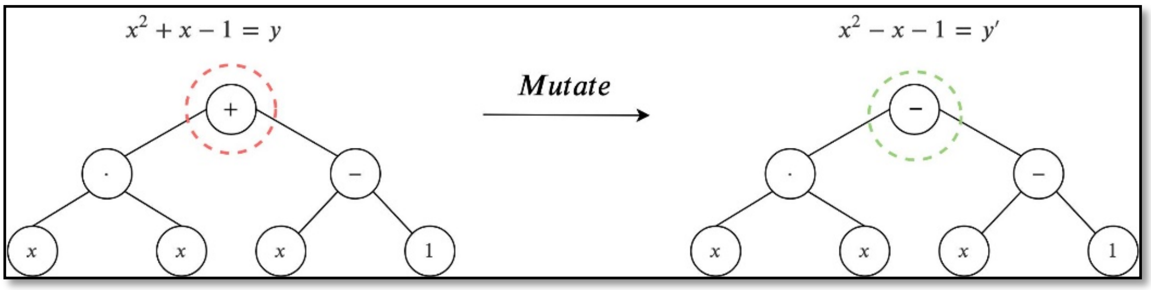


Figure 3-ix*: MutateNonTerminalR Strategy - The MutateNonTerminalR strategy functions on a non-terminal and switch the mathematical operation*

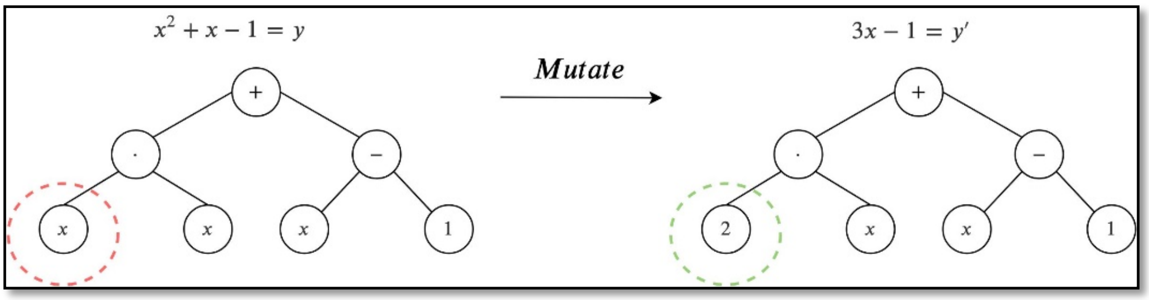


Figure 3-x*: MutateTerminalR Strategy - The mutate terminal strategy operates on a terminal and switch the value*

3.8.2 Illustration of How Mathematical Tree Manipulation Strategies Work

In *Figure 3-viii,* a portion of the program y is selected *(shown by the red dashed circle)* to be deleted. The *DeleteNonTerminalR* MTM strategy is used to truncate sections of a tree. The resulting tree adheres to STGP principles keeping it mathematically sound outlined in *Section 2.2.3*. Since a node for deletion is selected at random, a deletion may take out significant portions of a tree causing unintended consequences as solutions that are converging to a high or low RMSE tree representation *(depending on the objective)* may a delete a valuable subtree potentially decimating all progress. To ensure mathematical soundness, all delete operations that leave dangling branches *(branches with no leaf-node)* on binary non-terminals will replace the deleted portion with the value 0, or 1 depending if the operation intends to divide by the terminal (to avoid divide by 0).

The *MutateNonTerminalR (see Figure 3-ix)* and *MutateTerminalR (see Figure 3-x)* strategies tend to be less disruptive with regards to tree structure as they leave the structure of the tree relatively unchanged and only focus on the value of the non-terminal and terminal respectively.

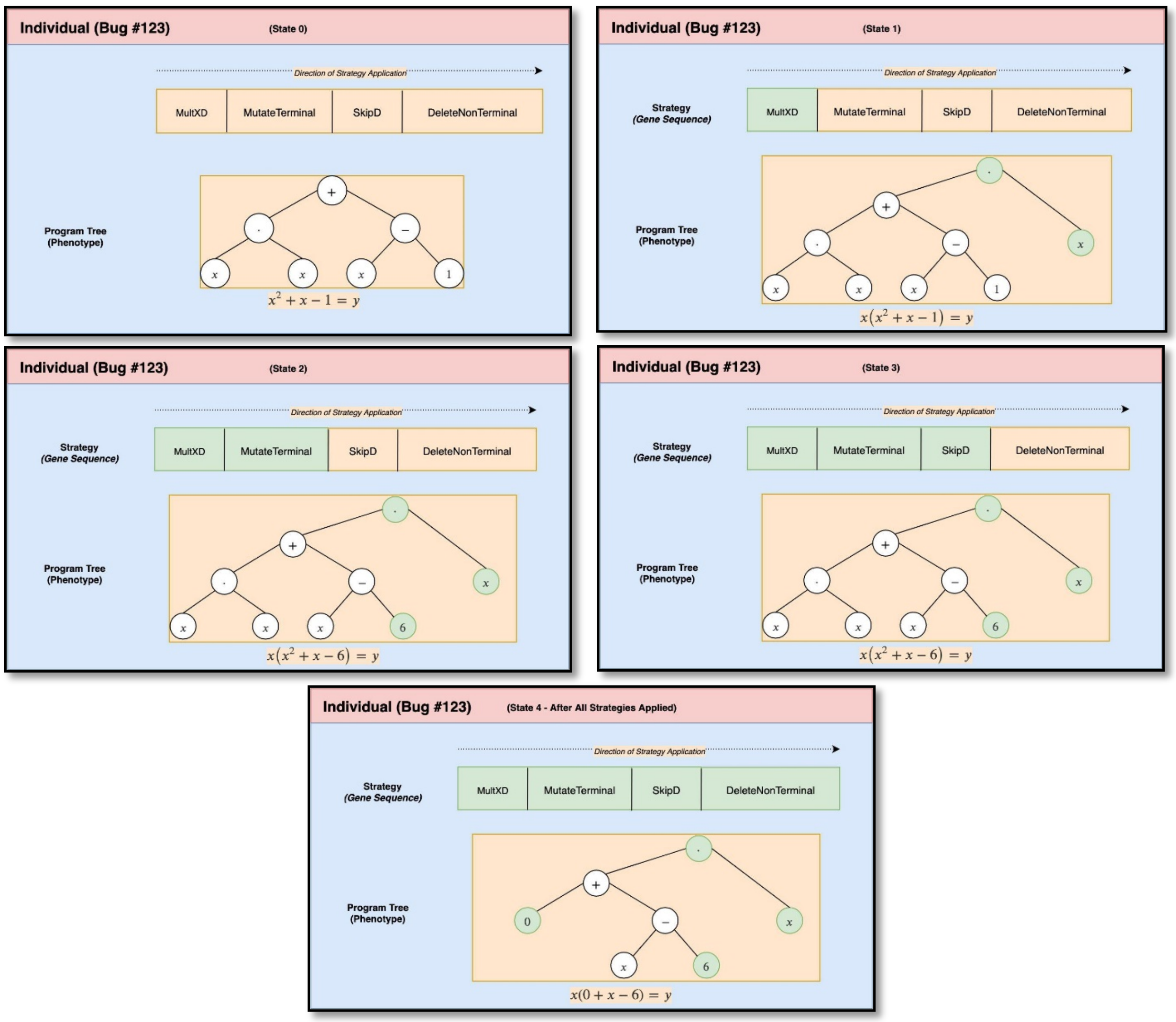


Figure 3-xi: *Strategy application - State-0 to State-4 of strategy application - How a chromosome containing strategies is applied to an individual's program tree. The illustration shall use a bug with ID 123.*

3.8.3 Strategy Application

We are now aware of the various MTM strategies available. Strategy application delves into how chromosomes are expressed. The chromosome in an individual is represented as the gene sequence shown in *Figure 3-xi*. This sequence contains a list of MTM strategies (genes) that are applied to the program tree in order from left to right. *Figure 3-xi* shows an example of a bug *(Bug #123)* that is ready to apply its strategies to an input tree obtained from *Equation 3* shown at State 0. At State 1, the bug applies its first strategy *MultXD*. Note the change to the program tree. The next strategy is *MutateTerminalR* strategy shown in State 2. The bug then applies the *SkipD* strategy. *SkipD* performs no action to the tree at State 3, allowing the individual to skip a transformation. Lastly at State 4, the final strategy *DeleteNonTerminalR* is applied. The resulting program tree generated from applying all strategies is:

$$x(x - 6) = y$$

This program is then handed to the competing test as an input program, and the test takes its turn by applying its respective strategies to this damaged program with the aim of minimizing the RMSE between the original input program and its own tree. Within the EP, there is a fixed number of strategies any individual can carry set by an input parameter. This value will be set to 15.

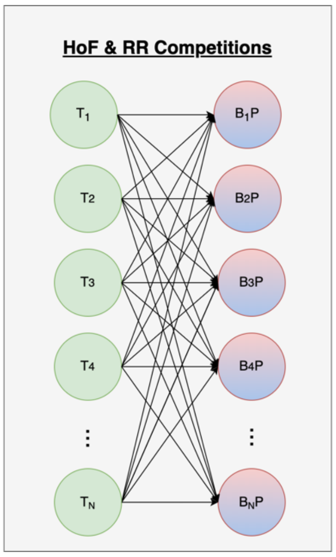


Figure 3-xii: *HoF and RR adversarial competitions: In HoF and RR each individual plays against every other individual. Tests here are shown in green, and bugs are shown in the red-blue gradient. The 'P' in the bugs name represents the program that has been damaged by the bug.*

3.9 COEVOLUTIONARY TOPOLOGIES

In *Section 2.7* we described a set of coevolutionary topologies presented in the literature. For our experimentation we shall analyze and if need be, modify the stated algorithms to suit our research requirements. For this research four coevolutionary topologies shall be used to compare their efficacy. These algorithms are Hall of Fame (HoF), K-Random Tournaments (KRT), Round Robin (RR), and Single Elimination Tournament (SET). We also show the computational complexity of each topology. The complexity defines the number of competitions played per generation for each individual and will aid in our comparison of when we discuss our results.

3.9.1 Hall of Fame

The HoF strategy mentioned in S*ection 2.7.1* requires the EP to archive the fittest individuals from each generation and reinsert them at a later generation. In HoF we chose to use a similar approach with regards to competitions as with that found in RR. In *Figure 3-xii* we see an illustration of how each test (green) will compete against each bug (red-blue gradient). Their fitness will be recorded in every match. HoF will archive the fittest kind of individual from each generation and reinsert them after M number of generations. Archived individuals will be reinserted to the population randomly replacing existing individuals. This will allow the EP to remember previous useful strategies and prevent them from being lost as the EP process evolves. If M is large it will inhibit progress and lead to a phenomenon called cycling where the evolutionary process perpetually moves through a similar set of similar solutions instead of proceeding with exploration (Eiben & Smith, 2015; Miconi & Channon).

Complexity Analysis: $N * N \ = N^2$

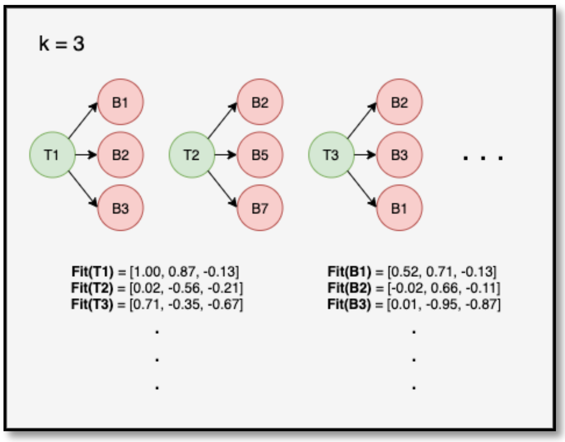

k = 3
T1
B1
B2
B3
T2
B2
B5
B7
T3
B2
B3
B1
. . .
Fit(T1) = [1.00, 0.87, -0.13]
Fit(T2) = [0.02, -0.56, -0.21]
Fit(T3) = [0.71, -0.35, -0.67]
Fit(B1) = [0.52, 0.71, -0.13]
Fit(B2) = [-0.02, 0.66, -0.11]
Fit(B3) = [0.01, -0.95, -0.87]

Figure 3-xiii: *KRT Competition Illustration: A KRT topology with k=3 will ensure an individual plays a maximum of 3 tournaments with randomly selected adversaries.*

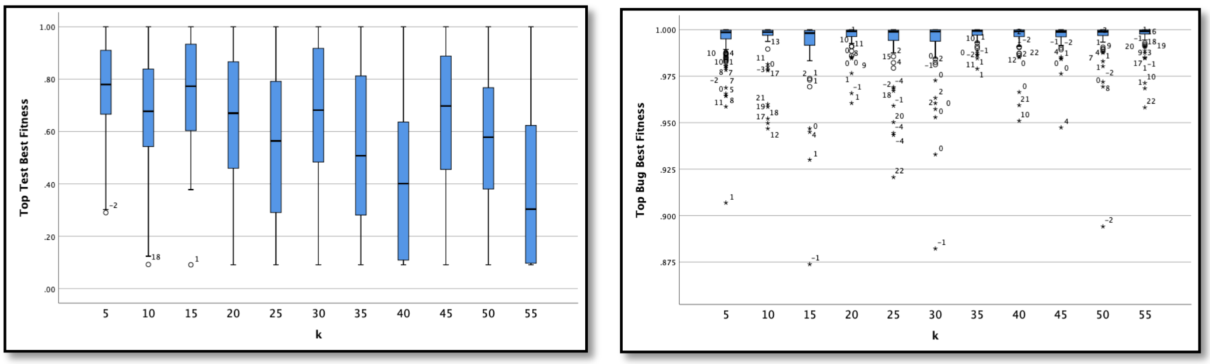


Figure 3-xiv*: Best Fitness values of the best bug and test vs the KRT k-Value (x-axis),*

3.9.2 K-Random Tournaments

K-Random tournaments (KRT) pits a set of competing individuals against a random set of other individuals of size k. KRT has significant performance benefits over RR, as a given individual only needs to play k games in a single tournament instead of N (where N is the total number of opponents in a generation). At the end of the generation, every individual has played k times, and a fair fitness evaluation can be performed. An illustration is shown in *Figure 3-xiii*. In this example T1 plays against B1, B2, B3, T2 plays against B2, B5 and B7 and so on. The fitness of each competition is recorded for both competing individuals. In our simulation we ensure that individuals do not play against the same adversary more than once. This simplifies the fitness calculation, and ensures every individual has a chance to play.

Panait and Luke (2002) warns against extremes for k citing bad performance and high computational costs if k is too large. We experimented with 11 different values of k across 1000 different simulations containing randomly generated polynomials. *Figure 3-xiv* shows the best bug and tests best fitness values when running against various k values shown on the x-axis. We observe a general decrease in best fitness for tests as k increases, however for bugs, we observe minimal variance across different k values. We can show that a low k value of 5 provides the best fitness for both individuals and consequently the best computational performance as fewer tournaments are played in each generation, as shown by the complexity analysis below.

Complexity Analysis: The complexity of KRT is $N * k$. As $k \rightarrow N$, complexity increases and average quality of individuals (primarily tests) decreases.

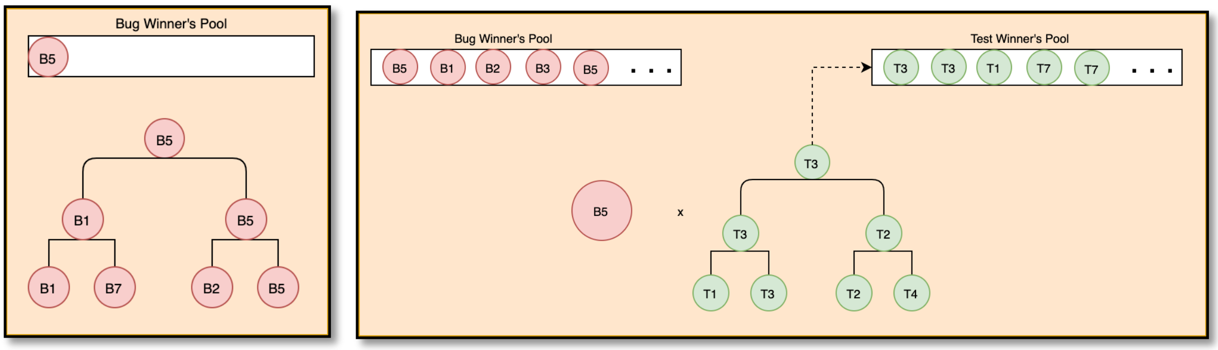


Figure 3-xv: *SET competition phase: (Left): A set of bugs competing against it each other to see which bug can best maximize RMSE on the input program. Winners of each round get added to the Bug Winner Pool list.*
(Right): The tests run their competitions after the bugs have filled up the Bug Winner Pool. Tests compete in a similar bracketed fashion against each other, however the goal of the tests is to take the supplied bug, in this case B5 and find the best test that can minimize the RMSE of the damaged program. The winner of the test gets added into the Test Winner Pool.

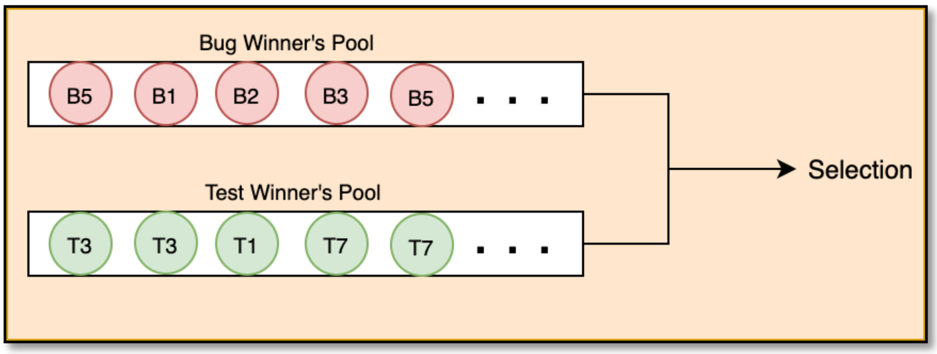


Figure 3-xvi: *Bug and Test Winner Pools with individuals ready to proceed to the next stage in the simulation, selection.*

3.9.3 Round Robin

The round robin (RR) strategy is the simplest to implement, as each bug in a given generation will compete against each test in the same generation as shown in *Figure 3-xii*. We labelled this competitive phase in the experimentation an epoch. In any generation there are N^2 number of epochs, where N is the number individuals per side. In our case there are the same number of bugs as there are tests. In a given epoch, the bug would go first and infect the input program with a variety of strategies *(See Figure 3-xi)*. Each strategy applied by the bug would modify the tree. Once all the strategies are applied the resulting tree is copied and passed on to the test in the same epoch. The test would then apply a set of its own unique strategies with the aim of repairing the damage to the input tree, by minimizing its RMSE. After applying all its strategies to the tree, fitness evaluation would be performed on both test and bug. The individuals would then undergo selection, and the resulting population would proceed to the next generation.

Complexity Analysis: The complexity of RR is $N * N = N^2$ as every individual plays every other individual.

3.9.4 Single Elimination Tournament

Single Elimination Tournament (SET) creates tournament brackets that pin like individuals against each other to see which individual is best at performing their respective tasks. In *Figure 3-xv (left)* we observe a set of bugs that have been randomly placed into a bracket to compete against each other. In our case, B1 competes against B7, B2 against B5. The bugs compete by trying to maximize RMSE on the IP. The bug that does this the best progresses into the next round. In our case, the overall winner of that tournament is B5. The winner is then placed into a winner's pool, and the tournament will be played another $SETCount$ times, $SETCount$ in this case being an

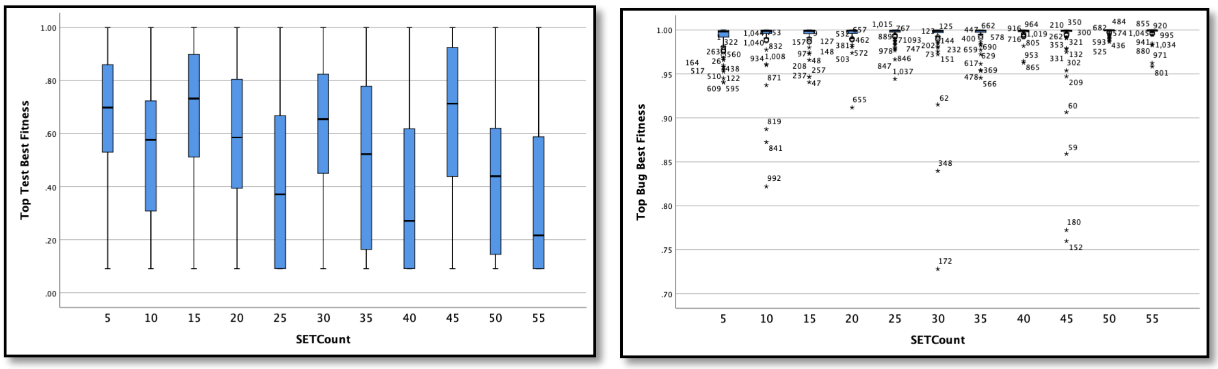


Figure 3-xvii: *Best Bug and Test Fitness vs SETCount*

input parameter that determines the number of tournaments that will be played. Once $SETCount$ number of competitions have been played, $SETCount$ winners will emerge. There is a possibility that the same individual might win multiple tournaments. This is a desired outcome as we want to ensure we are finding the fittest individual. If an individual wins multiple tournaments, they are cloned and added to the winner's pool.

The winners from the bug tournament are then transferred to the next phase where tests will compete against other tests to determine which test has evolved the best strategy to defeat each of the bugs. The tests tournaments will be run for each bug in the winners list. Once complete there will be an equal number of winner tests and bugs. We then randomly select non-winners from the bug population and add them to the bug winners list in the event that $SETCount < N$, where N is the total bug population size, and do the same for tests. The random addition of non-winner individuals also helps increase diversity from the population and can help prevent against the pathology of over-specialization. Both tests winners and bug winners' lists should contain the same number of individuals. These individuals can now proceed to the selection phase of the EP.

To allow for easier pairing, the number of competing individuals on each side must be 2^x to allow for even tournament seeding. SET has potential to be computationally expensive as only a single individual emerges each time a tournament is played. We experimented with $SETCount$ values ranging from 5 tournaments to 55 tournaments across 1000 different simulations each containing randomly generated polynomials. The results shown in *Figure 3-xvii* are similar to that of the k value of KRT. Smaller tournament numbers fair better from a fitness perspective in contrast to larger tournaments. For tests, the $SETCount$ value with the largest number of fittest tests is 15. For bugs, varying the $SETCount$ did not seem to have a discernable difference. As a result, we chose to use an $SETCount$ of 15. This means that 15 individuals will be selected from the bracket-based tournaments. The remaining individuals will include randomly selected individuals from the population allowing us to maintain a level of diversity across the respective populations.

Complexity Analysis: SET has a complexity of $2\big((N+1) * (SETCount)\big)$, where $N$is the total number of individuals of one kind in the simulation, and $SETCount$ is the number of tournaments individuals of one kind will play. We double this complexity to encompass both individuals.

3.9.5 Topology Complexity Evaluation

From a complexity standpoint, KRT given a moderate k value will contain the fewest number of competitions between in a given generation. As k tends to N, the number of competitions played in a generation scales to that of RR and HoF. Through experimentation across various samples of k, we note that having a relatively small number of k competitions can perform adequately both from a fitness and computational performance perspective. SET has a different tournament structure than the rest due to its knockout-like format that needs to be played. Large values of $SETCount$ have been shown to increase the computational complexity and also increase the selection pressure, as individuals with the best fitness will continue to win their respective tournaments and be added to the selection pool, leading highly specialized individual's that may be fit for a few generations, but any changes in the coevolutionary landscape could lead to a sharp decline in their ability to gain maximal fitness across multiple generations.

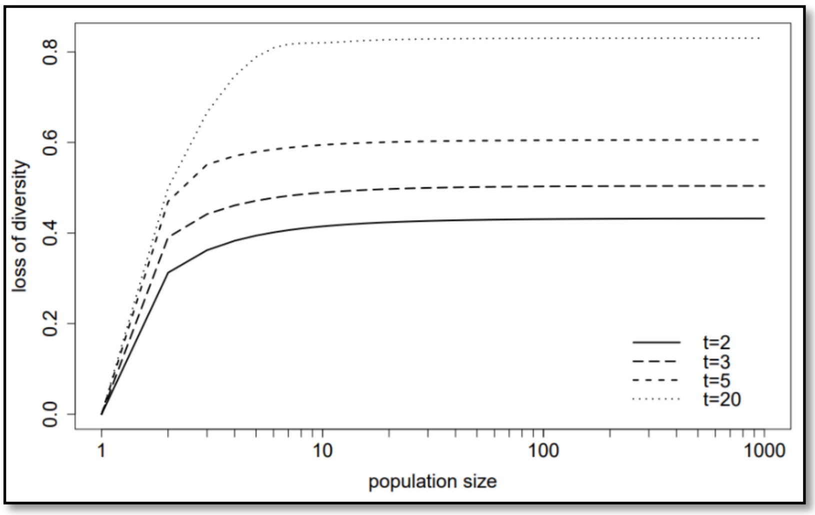


Figure 3-xviii: *(Sokolov & Whitley, 2005, p2) showcases the rate of loss of diversity against population size for different tournament sizes. A tournament size between 2 and 4 delays the loss of diversity and places a hard limit on total loss of diversity in a population.*

3.10 Selection

Selection guides the creation and destruction of individuals in the fitness landscape based on certain parent and survivor selection parameters (Sokolov & Whitley, 2005). Fitter individuals are generally allowed to progress so long as enough genetic diversity is maintained into subsequent generations (Sokolov & Whitley, 2005).

3.10.1 Parent Selection

The parent selection techniques typical for most GAs are Elitism and Tournament selection. Elitism, is a strategy where only the best solution progresses into the next generation, allowing for a steady state approach to the EP. Elitism runs the risk of quickly losing genetic diversity, however (Gonçalves et al., 2005; Poon et al., 1995) note that introducing high mutation rates may mitigate this issue. Tournament selection introduces a sampling bias where a tournament of size T contains randomly selected individuals (Sokolov & Whitley, 2005). N number of tournaments with random individuals are then played *(N represents the number of individuals)* The individual with greatest fitness in each tournament progresses (Sokolov & Whitley, 2005). The bias in this selection method prevents the EP on coalescing on a particular kind of chromosome leading to degeneration of diversity within the EP. Tournament selection has the added advantage of being simple to implement with little to no computational overhead (Howard & D'Angelo, 1995). This research will utilize tournament selection as the primary parent selection method, in order to perpetually maintain a good amount of diversity and increase chances of converging on a global maximum. A tournament size of 3 will be used in the experimentation to delay loss of diversity by minimizing selection pressure as shown in *Figure 3-xviii.*

3.10.2 Reproduction

After individuals compete, they must undergo a process where fit individuals mate to produce offspring that are potentially propagated to future generations (Chai et al., 1996). The primary recombination operator in EAs is a two parent crossover (Gustafson, Burke, & Krasnogor, 2005).

Crossover

Uniform crossover will be utilized on a chromosomal level to exchange strategies among parent individuals. We experimented with uniform and single-point crossover and found uniform crossover to perform the best in providing more diverse offspring due to the stochastic nature of how offspring are generated from uniform crossover. Uniform crossover maximized the chances of creating fit offspring that could escape local maxima more often giving the EP another chance at exploration. When looking to find a test that could evolve a strategy that could completely fix the

Parameters	Value(s)
Input Programs	$1556\ random\ polynomials$
Input Program Polynomial Degree (IPPD) Range	-11, 27
Evaluation Range	20
Seed	$x = -10$
Coevolution Topologies	RR, $\text{HoF}_{[M=5\ generations]}$, $\text{KRT}_{[K=5]}$, $\text{SET}_{[SETCount=15]}$
Fitness Thresholds	*Bug = 10,* Test = *1.1*
Strategy Count	15
MaxGenerations	300
EachPopulationSize	64
Parent Selection	$\text{Tournament}_{[T=3]}$
ProbOfMutation	0.1
Crossover	Uniform
SurvivorPercentage	0.5

Table 3-iv: *Complete Parameter List for all the simulations.*

buggy program, uniform crossover gave the best chance at attaining maximal fitness due to its ability to more uniformly intersperse parent genes across children, allowing for greater diversity across the search space.

Mutation

Tate and Smith (1993) advocates for higher mutation rates in GAs where chromosome encodings are not binary strings, which is the case presented in this research. Mutation is a strategy that aims to reclaim genetic diversity and is often rejected as a primary means of exploration (Arcuri & Yao, 2008b). It only has a subordinate role on the convergence of populations, only allowing them to transcend local optima (Lee & Antonsson, 2000). In our experimentation we shall use a constant value of 0.10. This value is not excessively large and leaves room for crossover to be the primary search mechanism.

3.10.3 Survivor Selection

In our research, survivor selection was controlled by the *survivorPercentage* parameter, that controlled the number of parents that would survive after being combined. The experimentation chose a half and half approach to survivor selection. The parents would be sorted in terms of their average fitness. The top $\frac{N}{2}$ parents (where N represents the population size) would be added to the survivor list. The offspring of the parents would then be randomly selected to fill the remaining $\frac{N}{2}$ slots. This ensured half the ongoing population to contain the fittest parents, and the remaining half to contain randomly selected individuals from the reproduction step.

3.11 EVOLUTIONARY PARAMETERS

Table 3-iv summarizes the 13 core evolutionary input parameters we set for all the simulations.

4 Experimentation

The entire experimentation will comprise of 1556 simulations. In each simulation, random programs will be generated for the bugs and tests to contend with across the four different topologies. The results from each topology will be recorded, and the next simulation with a different randomly generated program will begin.

4.1 Strategy Selection Predictions

Given the strategies outlined in *Section 3.8,* we see it fit as researchers to predict how bugs and tests may interact with certain IPs. We give 3 predictions based on the expected behavior of the best bugs and tests on IPs of polynomial degree (PD) -7, 1 and 10.

4.1.1 Sample Input Program-1 (IPPD = -7)

$$\frac{384}{x^7} + \frac{72}{x^3} + \frac{24}{x^2} + 16x + \frac{16}{x} - 40 \tag{10}$$

Program 1 shown by *Equation 10*, is a complex polynomial with its most significant degree being -7.

Researchers Prediction:

- ***Bug:*** In order to maximize RMSE, the bug may choose to continuously apply a *MultXD* or *DivXD* operation to quickly to maximize its RMSE from the spec. A fit bug may even choose to implement the *FellTreeD* operation early on to reduce the tree to the value 0. This automatically gives that bug the ability to nullify the -7 PD. At this point the bug may choose to exploit *MultXD* operations to further increase its RMSE from the IP.
- ***Test:*** If the test is handed a fit bug, such as the one presented above. It may choose to perform *FellTreeD* operation to nullify any *MultXD* work done by the bug. At which point the test can minimize RMSE by applying subsequent *DivXD* strategies until the PD normalizes around that value.

4.1.2 Sample Input Program 2 (IPPD = 1)

$$x \tag{11}$$

Program 2 shown by *Equation 11,* serves as a simple input program representing the independent variable x.

Researchers Prediction:

- ***Bug:*** The work of the bug seems straightforward in this regard. The bug should ideally develop multiple subsequent *MultXD* MTM strategies, to quickly maximize its RMSE from the spec.
- ***Test:*** Although this program is quite simple, it may prove a challenge for tests primarily if bugs underperform. If a bug underperforms and only changes the PD from 1 to 2 or something small. The test needs to have strategies in it that are carefully arranged to slightly undo the change. Since we fix the number of strategies to 15, the test should be careful to not overdo its operations. The test should also attempt to avoid strategies that introduce bloat. Any strategy that can limit tree growth such as *SkipD* or an extreme *FellTreeD* coupled with a *MultXD* strategy can assist the tree. Ideally the test should aim for a small tree, and this is where the challenge may lie for our test.

Terminology	Definition
Best Individual	This represents the best performing individual by average of all the fitness obtained through all the competitions they played in the simulation.
Run/Simulation	These words are used interchangeably to refer to the entire process outlined in *Figure 3-i*.
IP	IP is an abbreviation for input program. This is the mathematical function that is randomly generated and supplied at the start of the simulation for bugs to modify and tests to fix.
PD	This abbreviation stands for the Polynomial Degree of a program. PD values can belong to bugs, tests or then input program (IPPD).

Table 4-i: *General experimentation terminologies to assist with statistical interpretation of data*

4.1.3 Sample Input Program 3 (IPPD = 10)

$$\frac{4\,x^{10}}{63} - \frac{48\,x^{9}}{7} + x^{7} - 4\,x^{5} - 4 \tag{12}$$

Program 3 shown in *Equation 12*, tests large positive PD values for the IP.

Researchers Prediction:

Bug: The bug would continue to multiply values of x to the input program, to maximize its delta from the spec. The bug can also choose to use a *FellTreeD* to immediately diminish the PD count by 10.

Test: The best test would be best served with strategies that ensure the PD remains high. The fittest bugs are likely to select *MultXD* operations that further increase the PD, tests will need to know the best number of *DivXD* operations to apply and be wary of operations that may truncate the tree significantly. Such operations if placed at the end of the chromosome could be costly. We expect that fit tests in this simulation to have no *FellTreeD*, or any kind of *NonTerminalMutations* that can truncate the tree at the end of the chromosome.

4.2 SIMULATION OUTPUTS (VARIABLES)

Each simulation will output a set of variables that will be the focus of our analysis. These variables are shown in *Table 4-ii*. *Table 4-i* outlines a set of terminologies and their definitions that the research uses when analyzing the data.

	Variable	Definition
1	**Input Program Polynomial Degree (IPPD)**	This is a positive or negative integer the describes the **most dominant power** of x . **Relevance:** This variable will be act as a the primary independent variable whose relationship to the other variables in this table will be analyzed. **Example:** A mathematical expression such as $\frac{x^7}{x} - 3$ has an $IPPD$ value of 6, where as an expression of $\frac{3}{x}$ has a PD value of -1.
2	**Input Program Variable Count (IPVC)**	This is an integer that keeps track of the **number** of x variables in a given input program. **Relevance:** This variable will be act as a secondary independent variable whose relationship to the other variables in this table will be analyzed. **Example:** A mathematical expression such as $\frac{x^7}{x} - 3$ has an $IPVC$ value of 8 as there are 8 variables of x, where as an expression of $\frac{3}{x}$ has a $IPVC$ value of 1.
3	**Best Individuals in Simulation vs Polynomial Degree**	This variable contains information on the PD of the best individual's I output program in topology T in the simulation. **Relevance:** Since the degree of a polynomial is a useful indicator on the superficial performance for a given individual, we can track how well individuals performed by analyzing their best solutions PD against the IPPD. **Example:** If an input program that has a PD of 4, a test that fairs well would also have a PD of 4, whereas a good bug would maximize its distance from 4 either positively or negatively depending on the threshold set. Note that this is a useful high-level metric to observe for the best individuals and not the fitness of the individuals.
4	**Best Fitness in Simulation**	This variable represents the best individual I in a simulation for the given topology T. It is worth noting that the metric here looks at the individual with the best average fitness across all generations for a given simulation. Each topology will have a best individual, therefore each simulation will have a total of 8 best individuals. 4 best bugs from each simulation, and 4 best tests from each simulation. **Relevance:** Measuring the best individual's fitness is a core variable to observe how well the EP evolved individual's that were capable of identifying useful strategies to maximize their end goal.
5	**Average Fitness in Simulation**	This variable represents the average of a kind of individual $I's$ fitness in a simulation for topology T. **Relevance:** Unlike the "best fitness in simulation" output variable, this variable does not focus on one best individual, but rather all the individuals of a certain kind that existed in the topology during the simulation. It measures the average fitness of all individuals across all generations giving us a glimpse into how bugs and tests faired in general in their respective topologies.
6	**Best Age**	This variable represents the best kind of individual's I age when recorded. When an individual successfully passes selection and moves on to a subsequent generation, their age is incremented by 1. **Relevance:** This variable helps us observe how old fit individuals become before they either fall out of fitness due to increased selection pressure, or increased competition from adversaries. If the fittest individuals are relatively young it may point to signs of high competition and selection pressure.
7	**Best Birth Gen**	This variable represents in what generation the best individual I was born when running in topology T. **Relevance:** This variable useful when analyzing at what stage (generation) are the fittest or most fit individual born. This variable may highlight whether the fittest individuals are born early, in the middle or late in the EP.
8	**Best Individual Strategies**	This variable represents the strategy that the best individual had in a given topology during a simulation. **Relevance:** One of the core objectives of the thesis is to identify the set of strategies that fit individuals generate. An analysis on the form and structure of the best individual's strategies could give insight on the creation of generalizable strategies to maximize/minimize error given an input program.

***Table 4-ii**: Output variables to analyze from the experimentation.*

5 Data Analysis & Discussions

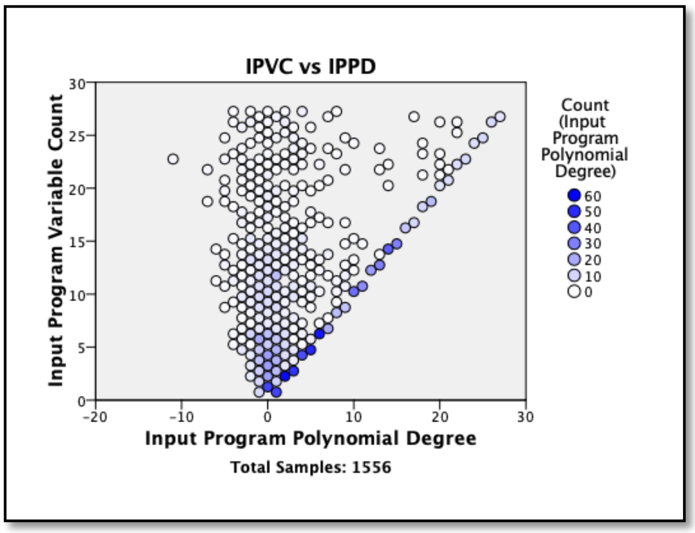
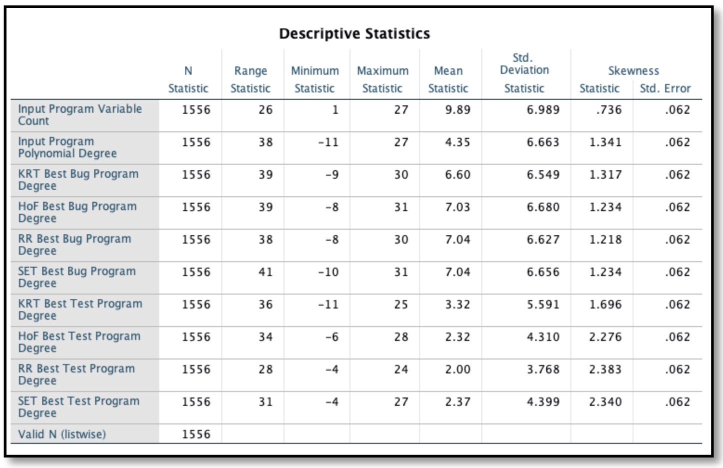


Figure 5-i: *The relationship between the input program variable count (IPVC) and the input program polynomial degree (IPPD) across 1556 simulations.*

5.1.1 IPPD vs IPVC

The comparison between polynomial degree and variable count is worth noting as their relationship may not be immediately clear. *Figure 5-i* showcases a hex binned scatter plot of IPPD vs IPVC. This illustration uses a gradient blue to map the frequency of the IPPD values and where they mostly clustered. In situations where the IPVC value is high and the IPPD value is low, we can assume the program had a larger number of $\div x$ (divide-by-x) operations. In sections where IPVC and IPPD values are high, we can infer a program had a large number of $\times x$ (multiply-by-x) operations. In our distribution, we see darker shades of blue cluster around the origin and spread out linearly as both IPPD and IPVC values increase. We also see lighter shades of blue extending upward in the diagram, indicating an increase in the frequency of IPPD variables that had a higher number of variables, but no net increase in polynomial degree. This occurs with IPs that have a balancing mix of $\times x$ and $\div x$ operations. As one can see from the diagram, IPPD can never be greater than the IPVC.

As an independent variable, either IPPD or IPVC could be used as a supplementary scalar independent variable in addition to observing the behavior of individual's in the various topologies. We opt for IPPD as it better informs the reader regarding the mathematical structure of the underlying polynomial and gives an accurate sense of the powers of x we are dealing with, unlike IPVC which only states the number of x variables in the given program. IPPD will therefore be used to indicate program complexity and show how the fitness of individuals across the various topologies scale given the complexity of the IP.



Descriptive Statistics

	N	Range	Minimum	Maximum	Mean	Std. Deviation	Skewness	
	Statistic	Statistic	Statistic	Statistic	Statistic	Statistic	Statistic	Std. Error
Input Program Variable Count	1556	26	1	27	9.89	6.989	.736	.062
Input Program Polynomial Degree	1556	38	−11	27	4.35	6.663	1.341	.062
KRT Best Bug Program Degree	1556	39	−9	30	6.60	6.549	1.317	.062
HoF Best Bug Program Degree	1556	39	−8	31	7.03	6.680	1.234	.062
RR Best Bug Program Degree	1556	38	−8	30	7.04	6.627	1.218	.062
SET Best Bug Program Degree	1556	41	−10	31	7.04	6.656	1.234	.062
KRT Best Test Program Degree	1556	36	−11	25	3.32	5.591	1.696	.062
HoF Best Test Program Degree	1556	34	−6	28	2.32	4.310	2.276	.062
RR Best Test Program Degree	1556	28	−4	24	2.00	3.768	2.383	.062
SET Best Test Program Degree	1556	31	−4	27	2.37	4.399	2.340	.062
Valid N (listwise)	1556							

Figure 5-ii: *Descriptive statistics for best bugs and tests in all topologies across all simulations. The IPVC and IPPD variables are added for reference.*

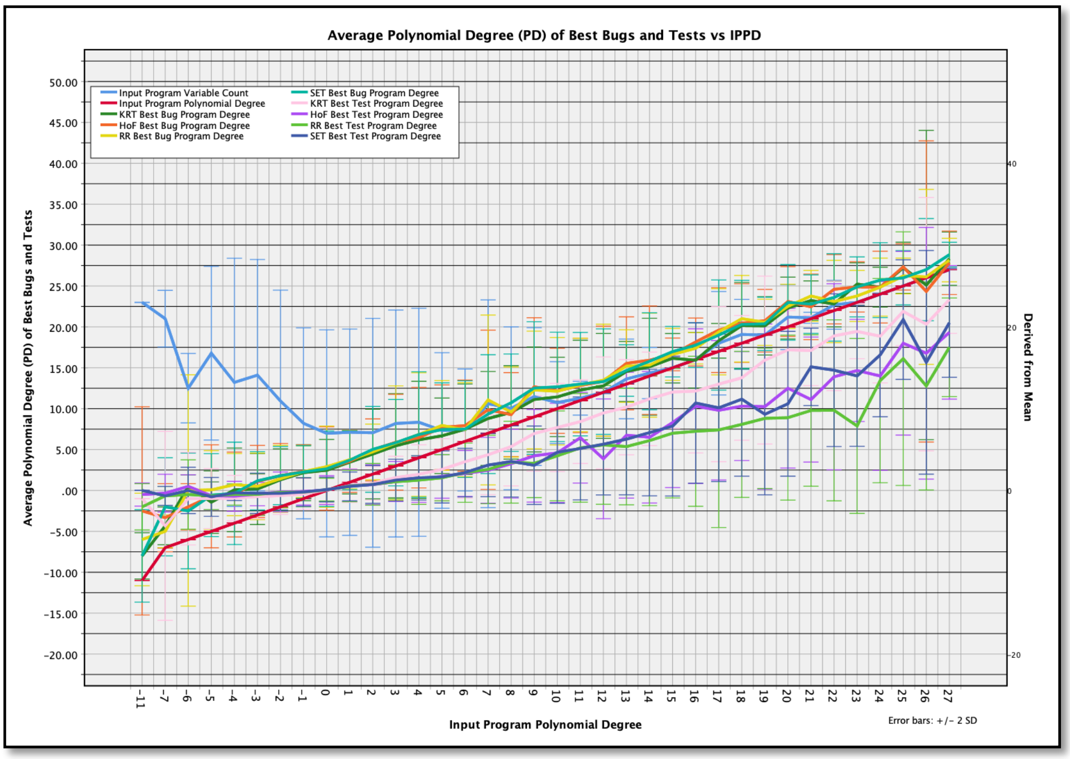


Figure 5-iii: *The average PD of the best bugs in all topologies against the IPPD values across all simulations. The IPVC (red) and IPPD (blue) variables are also plotted for reference.*

5.1.2 Topology Comparisons of Best Bug and Test's Polynomial Degree vs IPPD

Bugs:

Figure 5-iii showcases the descriptive statistics for the best bugs and tests in all topologies across the 1556 simulations with the IPPD and IPVC values are added for reference. When comparing the mean values for the best bugs in the different topologies, we see SET (7.04), RR (7.04), HoF (7.03) and KRT (6.60) all have values with similar means. Only KRT has a slightly lower mean than the rest. The standard deviations (SD) hover between KRT (6.55), RR (6.62), SET (6.66) and HoF (6.68). When observing the skewness of the distribution among the topologies, we do notice small variances from the original IPPD skewness (1.34) with KRT (1.317) being the closest to the IPPD followed by HoF (1.23), SET (1.23), RR (1.22). These statistical values indicate that all the tested topologies in the simulation had little difference in the resulting polynomials they generated. In *Figure 5-iii* we can visually see how KRT, HoF, RR and SET cluster around one another, and scale as the IPPD values scale. One interesting point to note is the 2SD ($\pm 2\ standard\ deviation$) plot in *Figure 5-iii***Figure 5-iii**. HoF (orange) tends to have larger deviations than the other topologies, primarily at the IPPD extremes. This is different to that of KRT, potentially indicating that HoF did better in creating solutions that maximized the RMSE from the IP. This would be possible as HoF plays a larger number of tournaments than KRT, allowing it to engage in competition that could eventually select for strategies that create this difference against the IP.

Given the IPPDs range (-11.00 to 27.00), we see that the topologies behave very similar. When we look at the minimum and maximum statistics, we also see marginal differences between all topologies. The largest range was exhibited by SET (41.00, from -10.00 to 31.00) with KRT (39.00, from –9.00 to 30.00) and HoF (-39.00, from -8.00 to 31.00) having the same ranges, and RR (38.00, from -8.00 to 30.00) having the smallest range. For all these topologies, we do see their maximums extend past the IPPD maximum of 27.00 shown by some of the IPPD values, however none of their minimums extend below the IPPD minimum (-11.00). This could indicate a preference for solutions that select for the *MultXD* strategy. Note how the different topologies all coalesce slightly above the red IPPD line perhaps suggesting bugs preferred higher order polynomials than lower ones. The biggest difference between bugs can be seen to be when IPPD values are at -11.00.

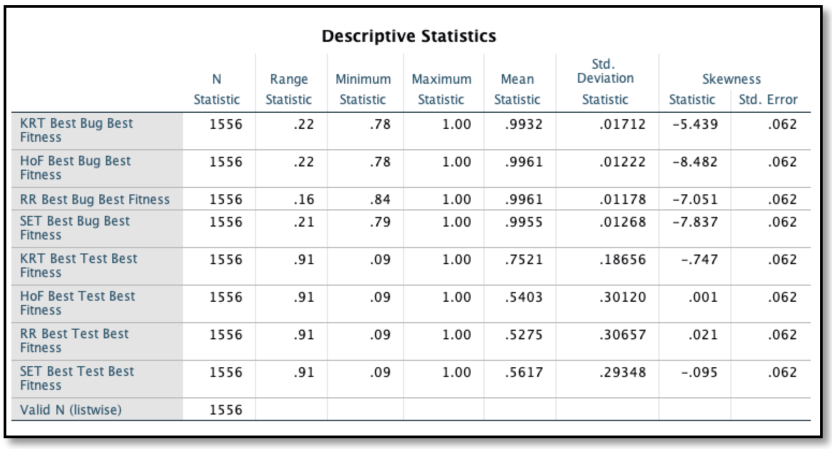
It is safe to summarize that the topologies themselves showed minor differences when it came to generating buggy programs as can be seen in *Figure 5-iii.* The best buggy programs across the topologies were not that different with regards to PD.

Tests:

When observing *Figure 5-iii,* the clearest difference between bugs and tests is the positioning of the lines with respect to the red IPPD line. Bugs chose to positively increase their PD and stay above the red IPPD line (value for IPPD values greater than zero), tests opted to stay below the IPPD values with RR (2.00) having the greatest delta to that of the mean of the IPPD (4.35). This can also be seen in *Figure 5-iii* as green RR line is the furthest below the red IPPD line. From the descriptive statistics shown in *Figure 5-ii*, we can see that the mean for bugs ranged between KRT (6.60) and SET (7.04). For tests however, the means range from RR (2.00) followed by Hof (2.32), SET (2.37) and KRT (3.32) which are much lower than those of the bugs. This could be compensatory behavior exhibited by tests in an attempt to undo the work done by the bugs. Also note the differences are quite small between the averages, with KRT having the largest gap between the topologies.

KRT PDs maintain the smallest delta to the IPPD value across nearly all IPPD values. Since IPPD can be used as an indirect proxy to fitness, here KRT shows potential in creating individuals that more closely align to those supplied at input. In the illustration in *Figure 5-iii*, KRT outperforms all other topologies for IPPD values greater than 2. However, at the upper end, KRT (25.00) struggles to maintain the upper value of some of the IPPD simulations (27.00). Where bugs were successful in increasing PD over the IPPD, tests seemed to struggle. This could indicate that tests were over-compensating.

When observing the SD values of the topologies, we note that tests had higher SD values than that of the bugs with KRT (5.59) having the highest SD, followed by SET (4.34), HoF (4.31) then RR (3.77). KRT clearly outperformed the rest of the topologies in generating tests whose PD values aligned closely with those of the IPPD. RR performed the poorest, and what is interesting to note is that RR beyond the IPPD value of 12.00 begins to gradually fall off from the other topologies. The lower SD in RR and lower PD mean, may indicate continual degenerations into local maxima as IPPD increased. Here is where we see the difference between HoF and RR, the reintroduction of previously fit solutions may have benefitted HoF with regards to generating solutions with PDs that match the IPPD value.



Descriptive Statistics

	N	Range	Minimum	Maximum	Mean	Std. Deviation	Skewness	
	Statistic	Statistic	Statistic	Statistic	Statistic	Statistic	Statistic	Std. Error
KRT Best Bug Best Fitness	1556	.22	.78	1.00	.9932	.01712	-5.439	.062
HoF Best Bug Best Fitness	1556	.22	.78	1.00	.9961	.01222	-8.482	.062
RR Best Bug Best Fitness	1556	.16	.84	1.00	.9961	.01178	-7.051	.062
SET Best Bug Best Fitness	1556	.21	.79	1.00	.9955	.01268	-7.837	.062
KRT Best Test Best Fitness	1556	.91	.09	1.00	.7521	.18656	-.747	.062
HoF Best Test Best Fitness	1556	.91	.09	1.00	.5403	.30120	.001	.062
RR Best Test Best Fitness	1556	.91	.09	1.00	.5275	.30657	.021	.062
SET Best Test Best Fitness	1556	.91	.09	1.00	.5617	.29348	-.095	.062
Valid N (listwise)	1556							

***Figure 5-iv:** Descriptive statistics for best bugs and tests fitness in all topologies across all simulations.*

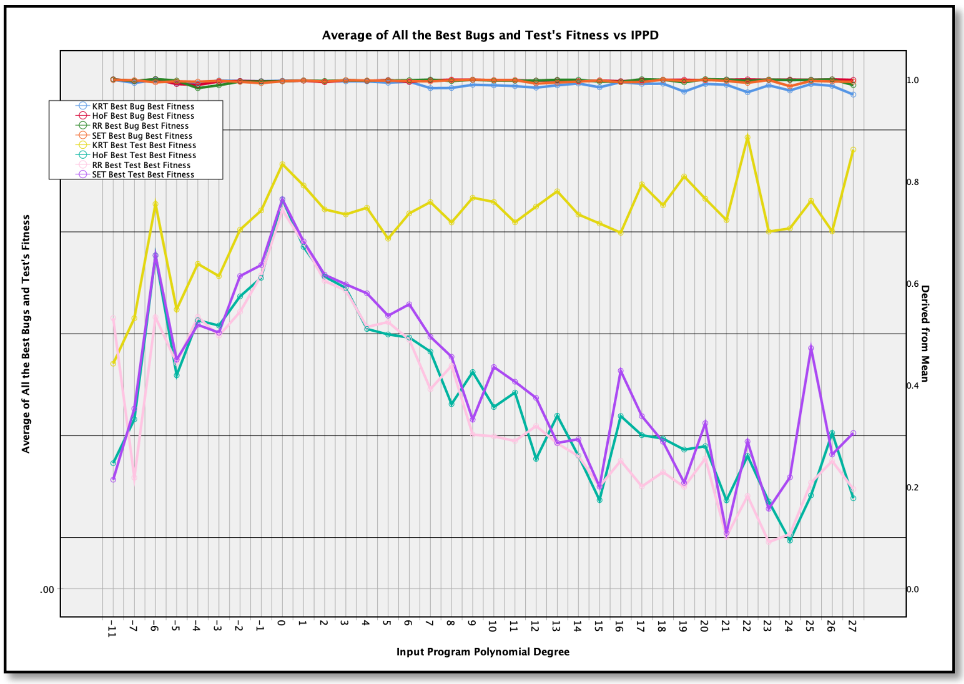


***Figure 5-v**: Average of all the best bugs and test's fitness in all simulations vs IPPD*

5.1.3 Topology Comparisons of Best Bug and Test's Fitness Polynomial Degree

In this section we compare the fitness of the best bugs and tests in all simulations. Keep in mind the thresholds for bugs (10.00) and tests (1.10)

Bugs

The descriptive statistics in *Figure 5-iv* show best fitness values attained by tests and bugs in all topologies across all simulations. It is immediately clear when looking at the range and mean statistics that the top bugs throughout all the topologies outperform the tests. This is also visible in the line chart in *Figure 5-v*. In all topologies of the bug the mean statistic is about 0.99 indicating that all topologies were able to find fit bugs that maximized RMSE against the PD.

The SD values range between 0.01 and 0.02 which is quite small showing consistent high success rates for bugs. The bug fitness values are highly skewed negatively further indicating the large cluster of the best bugs with high fitness values. It may be tempting to assume that the best bugs outclassed the best tests, however, remember that threshold-RMSE fitness pins an individual against its threshold. Given that the *BugThreshold* was set to 10 (a minimum of a 10x error for each x value evaluated on the IP), we see that the best bugs on average were able to surpass this threshold. When observing the minimum and maximum values RR (min: 0.84, max: 1.00) had the smallest range and was therefore the most consistent at producing the best bug that attained the best fitness. RR also had the smallest SD when looking at its performance with regards to IPPD in the previous section. The subsequent topologies were closely packed with SET (min: 0.79, max: 100) followed by KRT(min: 0.78, max: 1.00) and HoF(min: 0.78, max:1.00).

Tests

The performance of the best tests was not as promising against their threshold (1.10), even though it may be argued that the test's threshold is quite close to the IP (1.00) with it only being 1.1 times off the original IP. Note that if a test got a fitness of 0.00, it means they were only able to attain the limit set by their thresholds. If a test got a fitness better than 0.00, it means they were able to improve against their threshold and proceed towards the perfect solution (1.00).

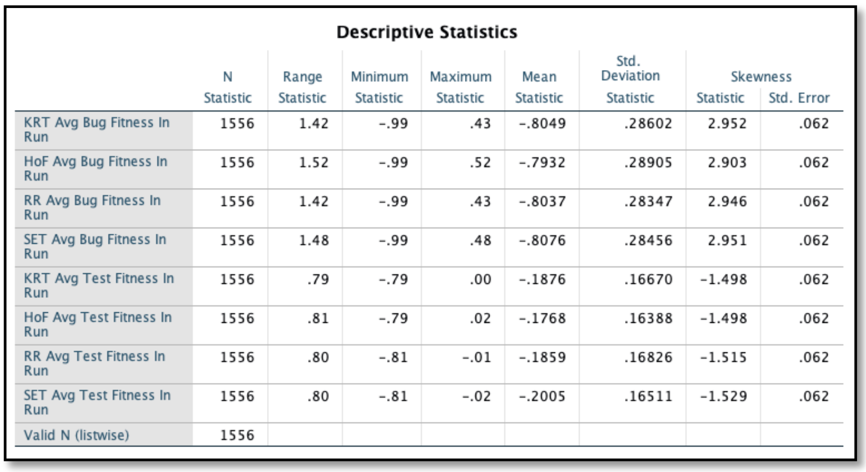
Out of the best tests, the maximum statistic received by all test topologies was still a perfect score. This means each topology was guaranteed to produce at least one perfect test. The mean of the best performing test was KRT (0.75). There is a substantial gap to 2nd placed SET (0.56) followed by HoF (0.54) and lastly RR (0.53). RR seemed to favor bugs quite well, whereas in the tests we see it falling to last place. This was expected after our findings in the previous section showed RR struggling to keep up with the other topologies *(See Figure 5-iii)*.

One point to note from *Figure 5-v,* is the performance degradation by HoF, SET and RR for the tests at a PD value of about 0.00. Both negative and positive ends of the PD spectrum seem to hurt the tests, primarily HoF, RR and SET. KRT on the other hand actually improved as IPPD increased.

The analysis of best tests against IPPD in *Figure 5-iii* seems to have given a valid prediction on what we should have expected with regards to the fitness of the best tests. From a test standpoint, KRT is not only the best performer from a fitness perspective, but also the best performer computationally

Summary

To summarize, the best bugs seem to have little problems exceeding their threshold (10.00), with RR being the best topology at having the highest probability of generating the fittest bug. On the test side, the average of the best tests indicates a bit of a struggle to completely fix the program even though all the tests on average exceeded their threshold (1.1). With at least some topologies in some simulations gaining full fitness. Bugs seemed immune to the increase in program complexity indicated by IPPD, whereas tests began to struggle considerably once programs increased in complexity. KRT being the only unaffected topology. The fact that bugs outperformed tests can also be simply explained by the fact that it is generally easier to damage a program than fix it, as there are fewer valid paths to fixing a damaged program than there are to damaging one.



Descriptive Statistics

	N	Range	Minimum	Maximum	Mean	Std. Deviation	Skewness	
	Statistic	Statistic	Statistic	Statistic	Statistic	Statistic	Statistic	Std. Error
KRT Avg Bug Fitness In Run	1556	1.42	-.99	.43	-.8049	.28602	2.952	.062
HoF Avg Bug Fitness In Run	1556	1.52	-.99	.52	-.7932	.28905	2.903	.062
RR Avg Bug Fitness In Run	1556	1.42	-.99	.43	-.8037	.28347	2.946	.062
SET Avg Bug Fitness In Run	1556	1.48	-.99	.48	-.8076	.28456	2.951	.062
KRT Avg Test Fitness In Run	1556	.79	-.79	.00	-.1876	.16670	-1.498	.062
HoF Avg Test Fitness In Run	1556	.81	-.79	.02	-.1768	.16388	-1.498	.062
RR Avg Test Fitness In Run	1556	.80	-.81	-.01	-.1859	.16826	-1.515	.062
SET Avg Test Fitness In Run	1556	.80	-.81	-.02	-.2005	.16511	-1.529	.062
Valid N (listwise)	1556							

***Figure 5-vi**: Descriptive statistics for best bugs and tests fitness in all topologies across all simulations.*

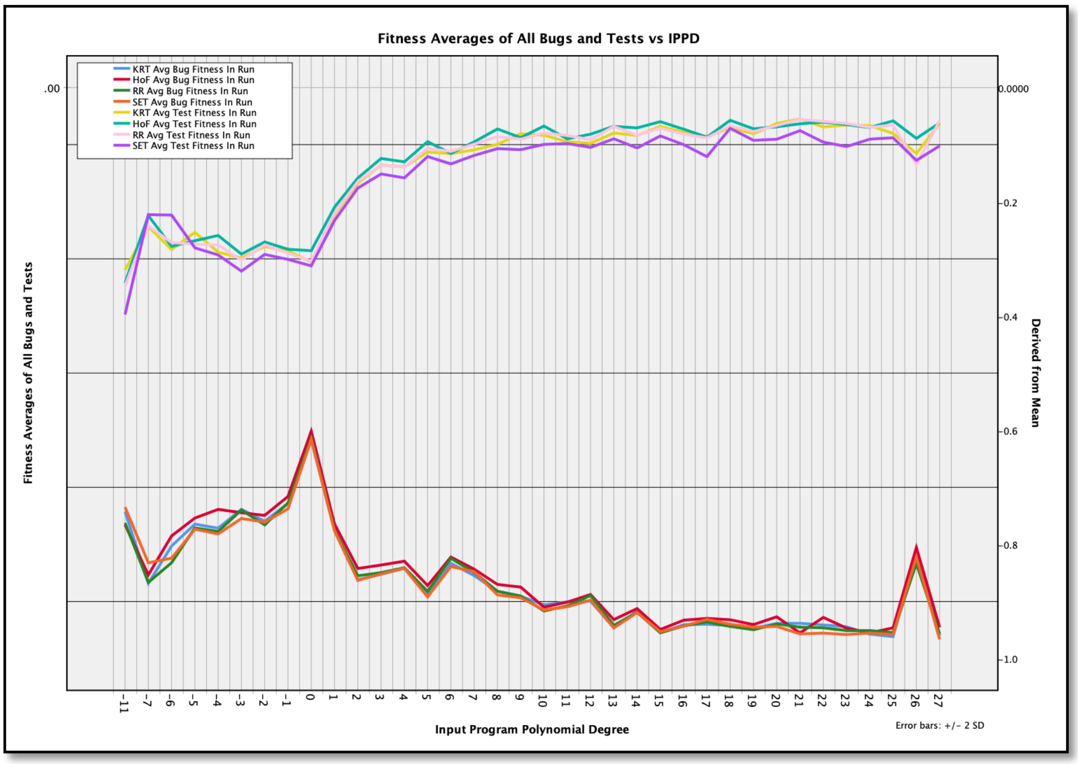


***Figure 5-vii**: Average of all bugs and test's fitness for all topologies in all simulations vs IPPD*

5.1.4 Topology Comparisons of Average Bug and Test's Fitness Polynomial Degree

Still on the topic of fitness evaluation, we take a closer look at the average performance of all bugs and tests in all topologies across all simulation. This metric does not look at the **best** individuals as was evaluated in *Section 5.1.3*, but rather the **average of all individuals** that existed in a given topology in a given simulation. This metric will give a better indication of the performance of the other individuals that took part in the EP.

Bugs

Figure 5-vi details the descriptive statistics of the averages of all bugs and tests in all topologies across all simulations. Given the performance of the best bugs in *Section 5.1.3* one would assume the average performance of the bugs would also be high, or at least high enough to beat the tests. The results shown in *Figure 5-vi* differ from that hypothesis quite severely. The mean value for the averages of all bugs is exceedingly low in all topologies. The mean values across the 4 topologies are quite clustered together, with HoF (-0.79) having a slight advantage over 2nd placed RR (-0.80), 3rd placed KRT (-0.81) and SET (-0.81). This here shows the difference in performance between the best and the average of the bugs as being quite significant. When we observe the minimum and maximum statistics, we see that the minimum statistic for the average of all bugs in all topologies all hit lows of -0.99. We do see the best maximum statistic value for all bugs being held by HoF (0.52), followed by SET (0.48), then KRT and RR tied with (0.43). The SD of the different topologies attest to this wide range. The skewness for all topologies hovers around 2.92 indicating the fitness values across the topologies is skewed more so to lower fitness values across the distribution. In *Figure 5-vii* we note that as the IPPD increases the bugs get progressively worse as seen by the familiar inflection point at IPPD=0. This is may indicate that on average most bugs struggled as the programs became more complex. There is also an erroneous bump in average fitness for all topologies when IPPD=26.

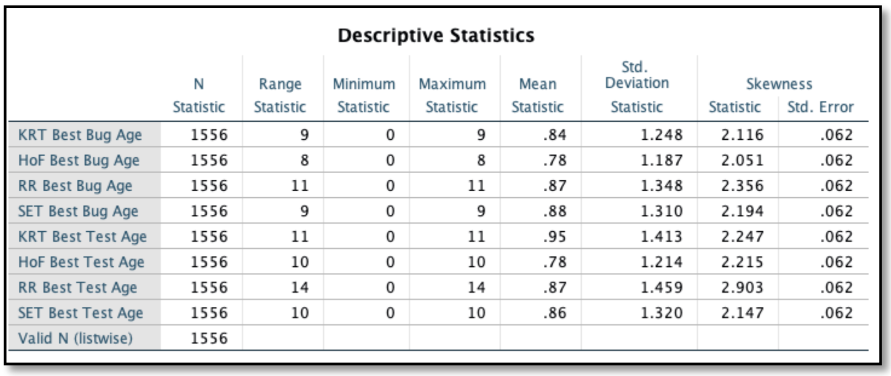
Tests

The tests exhibit a much smaller range to the best versions of themselves, than those shown by the bugs. The range is almost half, 1.50 for bugs, 0.80 for tests. The average of all tests in all topologies have much better minimum statistics than that of the bugs. Bugs all had minimum statistics of -0.99, where tests minimum statistics range between -0.79 (KRT) and -0.81 (HoF). The average of all tests, however, did not show higher maximum statistics than that of bugs. Where the best maximum statistic of the bug is HoF (0.52), the best maximum statistic of the test is also HoF (0.02) but with a much smaller value. What we see with the maximum statistic for HoF (0.02) and KRT (0.00) are the only topologies that managed to only make the threshold (1.1) and not improve on it. RR (-0.01) and SET (-0.02) both attained maximum statistics that were also quite close to that of HoF (0.02) and KRT (0.00) but failed to hit it. The SD values for the tests are lower than that of the bugs at about 0.16. The skewness statistic for the tests ranges between KRT (-1.50) and SET (-1.53) showing that the mode of the distribution clustered ahead of the mean and median. In *Figure 5-vii* tests just like bugs seem to have an inflection point at IPPD=0. However, unlike bugs, the average of all tests improves as the IPPD increases.

Summary

In summary, the average of all bugs and tests show that bugs encompassed a wider range, than that of tests on average. Although the average of all tests performs better. When analyzing the performance of bugs and tests against IPPD, we should remember that bugs interact with the IP first. It is possible that bugs can perform a mediocre job at damaging the IP, giving the tests the capacity to capitalize on the bugs mistakes. When looking at the negative IPPD values for bugs this indicates that the bugs fail to exceed its threshold, the more negative the value the more likely that the bug actually has no net effect on the IP. This may allow tests with weaker strategies to perform better and may explain why the significant drop in the average of all bugs performance, meant that tests could reap the benefits of the bugs lack of ability.

From what we have seen, bugs average performance varies significantly from best bug to average bug for the given thresholds. Tests tend to be more consistent but generally perform poorer than bugs.



Descriptive Statistics

	N	Range	Minimum	Maximum	Mean	Std. Deviation	Skewness	
	Statistic	Statistic	Statistic	Statistic	Statistic	Statistic	Statistic	Std. Error
KRT Best Bug Age	1556	9	0	9	.84	1.248	2.116	.062
HoF Best Bug Age	1556	8	0	8	.78	1.187	2.051	.062
RR Best Bug Age	1556	11	0	11	.87	1.348	2.356	.062
SET Best Bug Age	1556	9	0	9	.88	1.310	2.194	.062
KRT Best Test Age	1556	11	0	11	.95	1.413	2.247	.062
HoF Best Test Age	1556	10	0	10	.78	1.214	2.215	.062
RR Best Test Age	1556	14	0	14	.87	1.459	2.903	.062
SET Best Test Age	1556	10	0	10	.86	1.320	2.147	.062
Valid N (listwise)	1556							

Figure 5-viii: *Descriptive statistics for best bugs and tests age in all topologies across all simulations.*

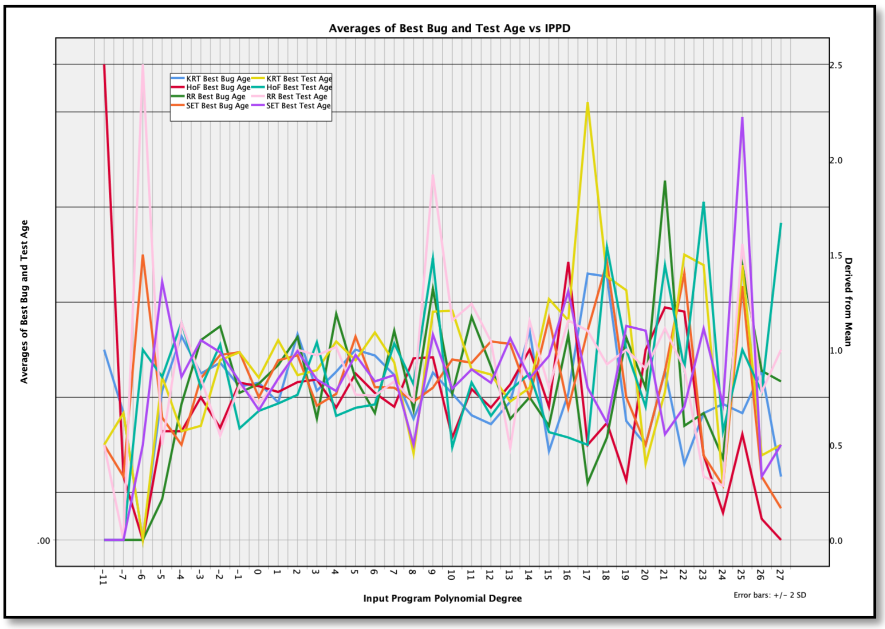


Figure 5-ix: *Average of all bugs and test's age for all topologies in all simulations vs IPPD*

5.1.5 Topology Comparisons of Best Bug and Test's Age Polynomial Degree

After viewing the performance of the average bug and test, we return our attention to the best bugs and tests and examine some of their other characteristics such as age, and birth generation.

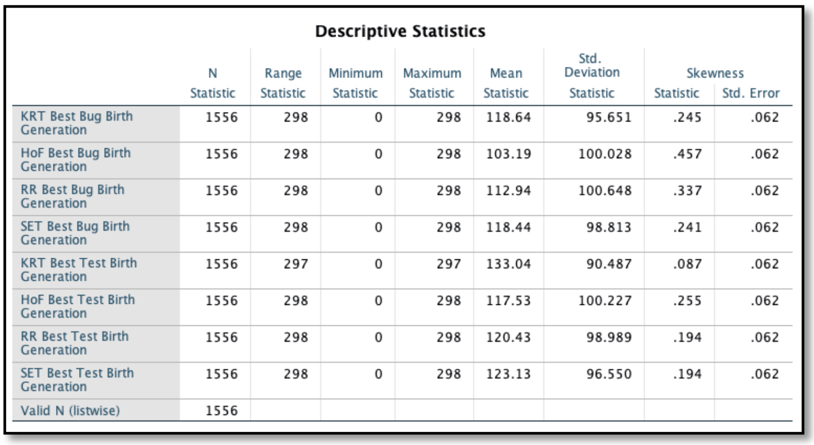
Age statistics are a useful metric in seeing how long individuals last in a simulation before selection takes over or they are no longer competitive. The age of the best individual is measured by the number of generations an individual has survived before failing to be selected for subsequent generations or even falling to 2nd place in their generation. We only look at the best individual and the age they were when they were last the best individual in their generation for their kind.

Small age numbers may indicate disruptive selection processes, increased competition amongst like individuals or competing individuals or even unlucky mutations that discard useful strategies. A moderately fit test that has survived a generation can be randomly placed against a set of very fit bugs in a subsequent generation and lose out in the selection process due to its strategies being inadequate to remedy the damaged programs. The randomness of competitions may help or hurt individuals trying to maximize their genes over time.

Figure 5-viii shows the descriptive statistics of the best bug and tests in all topologies across all simulations. For bugs, we see ranges that span from 0.00 for all topologies up to a maximum of exhibited by RR (11.00), KRT (9.00), HoF (8.00) and SET (9.00). This shows that some simulations had the oldest bug at about 11.00 generations old. There is no simulation where the youngest best bug was at least 1 generation old *(see minimum statistic)*. What is quite notable is the mean statistic for the different bug topologies ranging from HoF (0.78) to KRT (0.84) to RR (0.87) and SET (0.88). This means for all the simulation the average age of the best bugs was less than 1 generation old. This can be attributed to the factors listed in the previous paragraph. We see the same for tests where averages for tests in all topologies range from HoF (0.78) to KRT (0.95) with SET (0.86) and RR (0.87) being in between. On the maximums, we see RR (14) have the highest age, followed by KRT (11.00), HoF (10.00) and SET (10.00). Longevity is not a hallmark for this competitive coevolutionary landscape. This will almost certainly indicate that either reproduction techniques may have been disruptive, or that the strategies themselves were only good against a few sets of individuals. For the individuals that made it to 14 generations we see the upper limit on how long an individual can last before the coevolutionary landscape evolves enough such that the top individual loses its place at being the top.

It is worth noting that these values are for the best individual only. The individual who was the best for N number of generations may still be part of the selection pool but may have been dethroned by another fitter individual. The skewness statistics for all bugs and test is positive with an average value of 2.40, showing that both best bug and tests were more likely on the younger side of the distribution.

In analyzing the effect of IPPD against the ages of the individuals in *Figure 5-ix*, the general trend shows that as PD positively scales past 8 we see average age begin to climb but not by much. Similarly, on the negative PD side, averages begin to grow slightly capping out at 2 on both ends. This can perhaps be explained by the fact that bugs that can damage more complex IPs or tests that can fix complex IP programs must have traits that are very specialized allowing them to proceed to subsequent generation as their chromosomes are extremely unique in dealing with higher order polynomials. However as mentioned earlier the fitness landscape evolves quite rapidly and these highly specialized individuals end up dying out.



Descriptive Statistics

	N	Range	Minimum	Maximum	Mean	Std. Deviation	Skewness	
	Statistic	Statistic	Statistic	Statistic	Statistic	Statistic	Statistic	Std. Error
KRT Best Bug Birth Generation	1556	298	0	298	118.64	95.651	.245	.062
HoF Best Bug Birth Generation	1556	298	0	298	103.19	100.028	.457	.062
RR Best Bug Birth Generation	1556	298	0	298	112.94	100.648	.337	.062
SET Best Bug Birth Generation	1556	298	0	298	118.44	98.813	.241	.062
KRT Best Test Birth Generation	1556	297	0	297	133.04	90.487	.087	.062
HoF Best Test Birth Generation	1556	298	0	298	117.53	100.227	.255	.062
RR Best Test Birth Generation	1556	298	0	298	120.43	98.989	.194	.062
SET Best Test Birth Generation	1556	298	0	298	123.13	96.550	.194	.062
Valid N (listwise)	1556							

Figure 5-x: *Descriptive statistics for best bugs and tests generation of birth in all topologies across all simulations.*

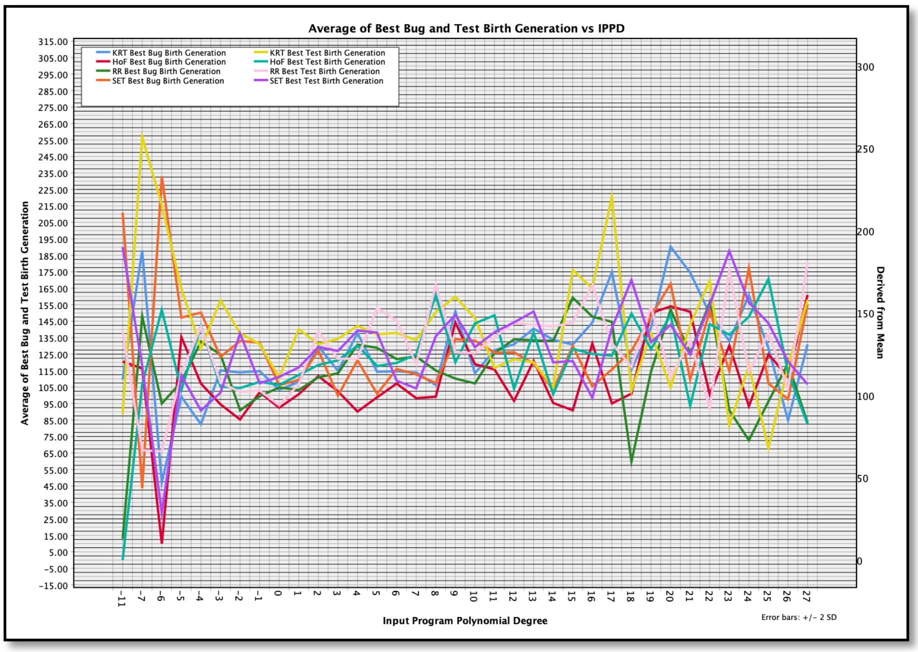


Figure 5-xi: *Average of all bugs and test's generation of birth for all topologies in all simulations vs IPPD*

5.1.6 Topology Comparisons of Best Bug and Test's Birth Generation Polynomial Degree

Birth generation is another metric that is worth observing as we gain insight into when in the EP the fittest individuals are born. This can give us a sense of whether the EP gradually over time creates fitter individuals, or fitter individuals are more likely to be randomly stumbled upon. Our analysis in on best individual ages in *Section 5.1.5* shows that the best bugs and tests share a very short lifespan. This allows us to hypothesize that the birth generation of individuals may be less deterministic.

Figure 5-x shows the descriptive statistics of the birth generation data for the best bugs and tests in all topologies across all simulations. Given that the simulation ran for 300 generations, we see minimum and maximum statistics that range over almost the entire generational cycle. With nearly all individuals and topologies having a range of 298 with the exception of best test KRT (297). We can therefore accept our hypothesis that the birth generation of the top individual can occur almost anywhere in the simulation. The standard deviation for all individuals across all topologies ranges between Bug KRT (95) and Bug RR (100) which is quite a significant SD. The minimum statistics for all individuals across all topologies is 0, indicating that it is possible to have the best individual be born after the first generation completes (the experimentation only performs terminates after at least the first generation completes).

When we look at the relationship of birth generations and the IPPD in *Figure 5-xi* we see significant deviations from the mean (125.00) between IPPD values of -11.00 to -3.00. Here we observe a wider average range of individuals being born. This trend also occurs at the higher values of IPPD. We can therefore conclude that birth generations for the fittest individuals is highly indeterministic and can occur at any generation in the simulation. This is a testament to the highly chaotic nature of the competition between the adversarial individuals.

Input Program	$\frac{10}{x^{11}} + \frac{2}{x^{10}} - \frac{16}{x^8} + \frac{9}{x^5} + \frac{18}{x^4} - \frac{8}{x} + 1$
PD	*-11*
Simulation	*1124*

	KRT Best Bug | 0.999	***KRT Best Test | 0.373***
	$36\,x - 17\,x^2 + 6\,x^4 - 33\,x^5 + 4\,x^6 - 2$	$\frac{8 - x^2}{x}$
	Strategies	
1	*SkipD*	*MutateNonTerminalR*
2	*DeleteNonTerminalR*	*DivCD*
3	*AddXD*	*AddXD*
4	*FellTreeD*	*DeleteNonTerminalR*
5	*MutateTerminalR*	*DeleteTerminalR*
6	*AppendRandomOperationR*	*SubCD*
7	*MultCD*	*AddCD*
8	*AddXD*	*MutateNonTerminalR*
9	*DeleteNonTerminalR*	*AppendRandomOperationR*
10	*MultCD*	*SubCD*
11	*SubCD*	*DivXD*
12	*SubXD*	*DeleteTerminalR*
13	*MutateNonTerminalR*	*DivCD*
14	*AppendRandomOperationR*	*MultCD*
15	*MutateNonTerminalR*	*DeleteNonTerminalR*

Table 5-i: *Sample simulation (7) with IPPD of -11.00. The best bug (0.999) in this case came from the KRT topology and the best test (0.373) also belonged to the KRT topology. The strategies are implemented in order from 1 to 15.*

Sample Raw Output Program	**Cleaned Output Program**
$\left(\left(\left(\left(\left((0-9)*x\right)*x\right)*8\right)*x\right)*8\right)$	$-576\,x^3$

Table 5-ii: *Raw output format vs cleaned output format.*

5.1.7 Analysis of Best Bug and Test's Strategies Across Different Polynomial Degree

Here we analyze a set of 3 sample solutions and their strategies and observe the outputs of the best tests and bugs as well as their strategies across various IPPDs. We use our predictions gained in *Section 4.1* to contrast the expectation in our findings. We have selected samples with similar IPPD values from those selected in *Section 4.1.* We look at IPPD values of *-11, 1, 14.* Note that the output programs shown in the following tables from both bug and test are presented in a clean mathematical notation and not in their raw programmed form as seen in ***Error! Reference source not found.***.

Sample Simulation Output Program 1: (IPPD = -11)

In *Table 5-i* we observe the best bug and test for an IPPD= -11.00. Both bugs and tests generate strategies that are to be applied to their underlying trees in order from 1 to 15. The results from *Figure 5-v* showed that the best tests struggled quite considerably at lower IPPDs hence why the best test for this simulation only has a fitness of 0.373. The input program when scaled across the seed and range gives very small numbers. Since RMSE focuses on the numeric delta between numbers, we can see why the bugs flourished in this environment as any attempt to increase the magnitude of the error generated will give high fitness. The bugs generate a polynomial of order 6 which is 17 polynomial degrees (PDs) from the IP. The evaluation across the spec range therefore gives the bug a significant amount of fitness (0.999). When we inspect the strategy of the bug, we see in position 4, a key strategy that may have helped the bug considerably. The *FellTreeD* strategy sets the IP to 0. Which is already 11 PDs away from the start. The bug goes on to perform additive operations from 5-15 (assuming the non-deterministic operations were adding to the tree) with only 11, and 12 being operations that may decrease the overall value of the polynomial when evaluated.

The tests here struggled to remedy most of the work done by fit bugs. There is no guarantee that the KRT test shown in *Table 5-i* actually faced the bug shown here, as they may have come from different generations and also KRT competition selection is random. What we do know is that for the test to return to the IP, it must find a way to either apply *DivX* operations to lower the IPPD or if the program supplied to the test already has a low IPPD (assuming the bug it face performed poorly), the test must preserve the low IPPD in order to gain fitness. As we can see from the strategies in *Table 5-i* the best KRT test implements conservative strategies that perform minor *DivX* operations, and the removal of extraneous parts with some *DeleteTerminal* operations etc. Note we only see one multiply operation and that is to multiply with a constant. It is fair to say that the test attempted to locate strategies that kept it small, just like the IP.

Here is where we perhaps see a limiting factor of the fitness evaluation technique we selected (threshold-RMSE). In a situation where you have very high positive or high negative PDs evaluated over large values of x, some values may overshadow the other values. This is a feature of polynomials however as the x values with higher degrees carry significantly more weight, even though the rest of the polynomial may express unique behavior when evaluated under other values of x. For negative PDs we note that RMSE becomes almost entirely ineffective as negative PDs such as x^{-11} coalesce around 0, for several values of x where as even having a positive PD such as $-10x$ can have a greater RMSE when evaluated across several values of x.

In *Section* 4.1.1 we hypothesized that the bug would opt for multiple *MultXD* operations or an early *FellTreeD* followed by *MultXD*. We do see this exhibited by our best bugs in the simulation. We predicted that tests would attempt to directly reverse what bugs chose to do, however it seemed like the best tests were satisfied with just having a PD value close to 0, as attempting to find strategies to further lower the PD value was less meaningful.

Input Program	8x + 5
PD	*1*
Simulation	*7*

	Best Bug$_{(SET)}$ | 0.999	**Best Test$_{(SET)}$ | 0.818**
	$70560\,x^3 + 2856\,x^2 - 16\,x + 6$	$\frac{2}{9}(37\,x + 23)(for\ x! = 0)$
	Strategies	
1	*SkipD*	*MultCD*
2	*DivXD*	*AddCD*
3	*SubCD*	*MutateNonTerminalR*
4	*AddXD*	*DivCD*
5	*SubXD*	*DivCD*
6	*DivCD*	*SkipD*
7	*FellTreeD*	*DeleteNonTerminalR*
8	*DeleteNonTerminalR*	*MultCD*
9	*SubCD*	*MultCD*
10	*AddCD*	*SubCD*
11	*AddXD*	*DivCD*
12	*DivXD*	*DivCD*
13	*DeleteNonTerminalR*	*SubXD*
14	*SkipD*	*MultXD*
15	*MutateNonTerminalR*	*MultXD*

Table 5-iii*: Sample simulation (7) with IPPD of 1. The best bug (0.999) and test (0.818) came from the SET topology. The strategies are implemented in order from 1 to 15*

Sample Simulation Output Program 2: (IPPD = 1)

Table 5-iii shows an IP with PD of 1. The best bug (0.999) and test (0.975) were from the SET topology. Once again, the bug was able to greatly exceed their threshold. Given the IP, we see the generated program from the bug achieve a max PD of 3, from the IPPD of 1. We also see other coefficients with a PD of 2. The bug made great use of the large coefficient (70560) for the variable with the largest PD. This helped considerably in the maximization of the RMSE. The *FellTreeD* in position 7 may not have been appropriate here as it may have disrupted progress made by the other strategies earlier in chromosome.

Depending on the program that was handed over to the test, the test seems to have adopted a set of strategies that cancel each other out. Note positions 4, 5 imply a division with some constant, 8, 9 seem to multiply back some constant 11, 12 also divide with some constant, and 14 and 15 multiply some constant. There's a clear back and forth set of strategies embedded in its chromosomes. This is a peculiar find, as it may suggest that the EP had generated bugs that barely had any error on the IP. This finding is consistent with our findings in *Section 5.1.4* where we saw that the average bug actually performed significantly worse than the best bug. This test may have been handed strategies from the EP that had little net effect when applied. Therefore, since the bugs were weak, tests needed to only maintain what the weak bugs had done to gain maximum fitness.

The fact that the PD is low may be more forgiving to bugs and tests that make mistakes, as it is easier to create and correct mistakes when the PD is low as the errors do not grow exponentially.

In *Section 4.1.2* we predicted that for low IPPD values the bugs would exploit the *MultXD* strategy as much as possible. We see this to a certain extend in this example. However, we do not that the bug also made use of the large coefficients to further maximize RMSE. We predicted that

Input Program	$\frac{1792\,x^{14}}{15} + \frac{224\,x^{6}}{3} - \frac{8\,x^{3}}{15} - 56$
PD	**14**
Simulation	**814**

	SET Best Bug = 0.999	**RR Best Test = 0.844**
	$\frac{1204224\,x^{17}}{5} + 150528\,x^{9} - \frac{5376\,x^{6}}{5} - 48\,x^{4} - 18816\,x^{3} + 48\,x^{2} + 239\,x$	$1806336\,x^{8} + \frac{8064\,x^{5}}{5} + 1080\,x^{3} - 5361\,x^{2} - 3\,x - 6\ (for\ x! = 0)$
1	*SubCD*	*AddKD*
2	*SubCD*	*MutateNonTerminalR*
3	*MutateTerminalR*	*SubCD*
4	*SubKD*	*MutateNonTerminalR*
5	*AddCD*	*AppendRandomOperationR*
6	*MultCD*	*SubKD*
7	*AppendRandomOperationR*	*SkipD*
8	*MutateNonTerminalR*	*SkipD*
9	*MultCD*	*SkipD*
10	*MutateTerminalR*	*MutateTerminalR*
11	*MultKD*	*SkipD*
12	*SubCD*	*DivKD*
13	*SkipD*	*MultCD*
14	*SubCD*	*SkipD*
15	*MutateNonTerminalR*	*AddKD*

Table 5-iv: *Sample simulation (814) with IPPD of 14. The best bug belonged to the SET topology and test in this case both belonged to the RR topology. Best bug with best fitness of 0.970 and the best test with the best fitness of 0.844.*

for tests to succeed, maintaining a small tree was important, we see this here as well with tests performing operations that cancel each other out.

Sample Simulation Output Program 3: (IPPD = 14)

The final PD we are going to evaluate is in *Table 5-iv*. This IPPD has a high value of 14. If we focus on the most prominent PD for the bug, we see a PD of 16. The test struggles to hit the PD of 14 only achieving a PD of 8. It does however make up for this somewhat by having a coefficient of order 10^{6}. When the bug is handed the IP, it makes use of additive strategies (strategies that add nodes to the tree), such as *SubCD*, *AddCD*, *MultCD*, *AppendRandomOperationR*. Even though strategies such as *SubCD* may decrease the overall RMSE, it does add a node to the tree that can later be mutated into an operation such as a multiply operation, that increases the RMSE. This goes back to the idea of how GP bloat when exploited in the right manner can be used as a means to further grow the tree. The bug therefore selects strategies that build on to the tree even though the RMSE may temporarily fall in the short term. It does implement the expected *MultKD* in position 11 but note the two *MutateNonTerminalR* in positions 8 and 15 that are likely to change to non-terminals that are already mostly subtractions into either additions or multiplications further allowing the bug's RMSE to grow. There is not a single destructive (operations that truncate the tree) on the bug such as *DeleteTerminalR*, *DeleteNonTerminalR* or *FellTreeD* showing that the bugs had evolved to getting rid of these genes. When we look at the predictions made in Section 4.1.3 we anticipated that bugs would exploit the *MultXD* strategy, or if the PD was great enough, use an extremely destructive strategy such as FellTreeD to move in the opposite direction. What we see in the simulation however is that bugs avoid any destructive strategies and opt for strategies that only grow the tree in the hopes of maximizing RMSE.

The test fails to meet the same PD as the IPPD but makes up for it with the large coefficient. Assuming the test got a bug with a fairly large PD, we see the tests opt for more destructive strategies than the bug in an attempt to reduce the PD. We saw a glimpse of this in *Figure 5-iii* where the best tests across all topologies consistently had lower PD values than the IPPD, and best bug's PD. Tests made use of destructive strategies such as the *MutateNonTerminalR* strategy that has significant capacity to truncate the tree to any length depending on the non-terminal selected. We also see the tests opting to invoke multiple the SkipD operations, this is a peculiar strategy to take primarily when placed in the middle of the chromosome. This could suggest the test wanted to prevent taking further action. This test in particular has 4 SkipD operations, perhaps in attempt to prevent over-compensating by trying to bring down the PD of the bugs. In *Section 4.1.3* we anticipated that the tests would avoid placing any kind of extremely destructive strategies such as *FellTreeD* operations anywhere at the end of the chromosome as that would significantly harm fitness. We do see tests take a more measured approach with tree truncation by using the *MutateNonTerminalR* strategy instead.

6 CONCLUSION

In this study we coevolved a group of adversarial "bugs" and "tests" with the respective objectives of maximizing the error (damage) of an input mathematical function and minimizing the error of the damaged function (repair). We devised a unique fitness measurement technique known as threshold ratio mean squared error (threshold-RMSE), that supplied separate targets for both bugs and tests to meet in order to gain fitness. We observed that more intuitive coevolutionary fitness techniques that used a zero-sum fitness measurement did not work well in this environment as some individuals would elect to play the role of other individuals to maximize their fitness and therefore disrupt and further complicate the interpretation of results. The thresholds for both bugs and tests can be used by the researcher as an additional selection pressure parameter that forces bugs or tests to attain a certain level of performance in order to move forward in the selection phase.

In addition to selecting the appropriate fitness measurement for this competitive landscape, we utilized K-Random Tournaments (KRT), Hall of Fame (HoF), Round Robin (RR) and Single Elimination Tournament (SET) as topologies in our coevolutionary landscape and compared their effects on 1556 randomly generated single-variable polynomials that had degrees ranging from -11 to 27. Each topology was tried in each of the 1556 simulations in order to allow for a direct and accurate comparisons.

We observed that KRT with a moderately low k value of 5 was the least computationally intensive and provided the best average results for generating tests. KRT also performed well when generating bugs but the other topologies seemed to fair just as well. In general, large competition environments such as RR, HoF, competitions with high k values for KRT, or high number of tournaments in SET, showed minimal benefit. Input program complexity determined by the polynomial degree (PD) also played a role in determining the performance of both tests and bugs across the different topologies. As the polynomial degree (PD) rose or fell below zero, the probability rate of generating the best tests in the HoF, RR and SET topology began to fall. KRT was unaffected by changes in PD and showed to improve somewhat as the PD increased. The best bugs, however, could be generated at all PD.

We discovered that in almost all topologies, the coevolutionary landscape is quite chaotic. On rare occasions, the fittest individuals would last no more than 14 generations (exhibited in HoF). The best individuals would last on average around 1 generation before factors such as competition between like individuals, competition against antagonistic individuals, as well as parent and survivor selection would take effect and dethrone the current best individual. The chaotic environment unsurprisingly generated an environment where it was difficult to predict when the fittest bug or test would emerge from a given simulation with the average standard deviation in a 300-generation simulation being about 100 generations across all topologies and individuals. It was therefore difficult to predict if the desired coevolutionary arms-race where both bugs and tests become mutually fit together.

With regards to the best mathematical tree manipulation (MTM) strategies housed by each individual. We noticed each kind of individual (bug and test) opted for different strategies depending on several factors. One factor we measured was the strategies top tests and bugs opted for when the input program polynomial degree (IPPD) was scaled. For negative IPPD values, the bugs we observed chose to eliminate the program entirely resetting the PD to 0, then start increasing the PD using strategies that add to the tree. One of the best tests we analyzed opted to find strategies that remained around PD values of 0, as it may have not been worthwhile to risk searching for specialized DivX (divide by x) strategies that are the only strategies to decrease PD value. This hurt the tests as tests did not perform well when the IPPD was negative. For small IPPD values such as 1, both individuals performed well, the bug we sampled opted to perform operations that multiply to the tree or increase the coefficient of the largest variable, tests elected to fill their chromosomes with strategies that cancelled each other out in the hopes of maintain a low PD. For large IPPD values such as 14, the top bug we sampled exceeded its threshold gaining maximal fitness. The PD generated by the bug was 17. The bug opted for additive strategies that increase the number of nodes in the tree. The bug also found ways to mutate certain non-terminals later in its chromosome to then exploit further RMSE. The fittest bugs shied away from any tree truncating operations as a tree with more nodes was likely to maximize its RMSE. One of the best tests sampled opted for a more measured approach with regards to decreasing the large RMSE introduced by the bugs. Tests chose careful mutations of non-terminals in order to carefully truncate the size of the buggy tree, in hopes of returning back to the origin al IPPD. We also see tests elect to use strategies that have no net impact on the tree such as the Skip strategy. What we had seen earlier with larger IPPDs is that tests would overcompensate by greatly reducing the PD of the bug such that it surpassed the IPPD. The best tests we sampled utilized these skip strategies to limit the amount of alteration to the bug to avoid overcompensating.

All in all, the competitive coevolutionary landscape in nearly all the simulations was very dynamic with several factors playing crucial roles in determining the performance of the varying individuals. Tests not only had to develop strategies against the best bugs, but they also had to know what to do against weaker bugs so as to not apply over compensatory strategies. It is therefore difficult to find one strategy that fits all approach as the main purpose of a coevolutionary landscape is adaptation to adversarial pressure. We saw this in play with both bugs and tests given the strategies they chose to implement.

7 PROJECT LIMITATIONS & FUTURE WORK

The main limitations of the work which can be starting points for future work mainly involve strategy selection and count, and fitness measurement limitations.

7.1 STRATEGY SELECTION AND COUNT

Determinism played a factor in influencing results and anticipating the outcome of individuals. Future research could focus on more granular strategies or perhaps even strategies that the individuals come up with instead of a hard set of strategies that both individuals have to utilize. It would also be of use to have a flexible set of strategies with a more lenient upper limit to allow bugs and tests to best express what they are to do. The only complication with such a technique would be having to scale crossover and other mating strategies for individuals with variable list size chromosomes. We attempted to mitigate this with the introduction of the skip strategy to allow individuals to not be compelled to have to use a manipulation operation on the tree. We saw this used scarcely though more so by the sample test with higher polynomial degrees.

7.2 FITNESS LIMITATIONS

7.2.1 Threshold Fitness

A limitation posed by threshold fitness is that it decouples a natural competition between adversarial individuals. Individuals although competing head to head are only attributed fitness given their performance against an external parameter. The choice of threshold fitness was required given our limitations with the more intuitive ratio fitness. Further experimentation can look into the value of what thresholds are appropriate in what context. In some experiments, a bug may not need to actually damage a program that much, how would the tests respond if the perturbation is marginal? The role of the threshold can act as an independent variable for both adversarial individuals that could be explored more in future work.

7.2.2 RMSE Fitness

RMSE as a measure to show the magnitude difference between functions has significant limitations primarily for polynomials with negative PD values. Expression such as $-10x$ when contrasted to other expressions such as x^{-11} have greater RMSE for several values of x even though x^{-11}has a higher magnitude PD. The bugs sampled showed little interest in using the *DivXD* strategy as any *DivXD* that pushed the IP into negative values of PD would be of little benefit to bugs. PD values less than 0 may not be best suited to RMSE based fitness.

In addition, if the IPPD is very large, bugs seem to have quite a significant advantage as even minor alterations to their chromosome such as *MutateTerminalR*, *FellTreeD*, *MultXD* become valid strategies that create significant deltas to the IP; this places added pressure on tests to come up with specialized solutions to counteract this. Bugs have more room to be awarded for mediocre strategies. This is perhaps why we failed to see the best tests continually hit high 0.99 ranges across all topologies.

Future work should look into more suited measures of deltas between polynomials that could more accurately inform fitness functions.

8 BIBLIOGRAPHY

Arbuckle, D. J., & Requicha, A. A. G. (2010). Self-assembly and self-repair of arbitrary shapes by a swarm of reactive robots: algorithms and simulations. *Autonomous Robots, 28*(2), 197-211.

Arcuri, A., & Yao, X. (2008a, 2008). *A novel co-evolutionary approach to automatic software bug fixing.* Paper presented at the Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on.

Arcuri, A., & Yao, X. (2008b). Search based software testing of object-oriented containers. *Information Sciences, 178*(15), 3075-3095. doi:10.1016/j.ins.2007.11.024

Arcuri, A., & Yao, X. (2014). Co-evolutionary automatic programming for software development. *Information Sciences, 259*, 412-432. doi:10.1016/j.ins.2009.12.019

Augusto, D. A., & Barbosa, H. J. (2000). *Symbolic regression via genetic programming.* Paper presented at the Proceedings. Vol. 1. Sixth Brazilian Symposium on Neural Networks.

Barbosa, H. J., & Lemonge, A. C. (2002). *An Adaptive Penalty Scheme In Genetic Algorithms For Constrained Optimiazation Problems.* Paper presented at the GECCO.

Barmpalexis, P., Kachrimanis, K., Tsakonas, A., & Georgarakis, E. (2011). Symbolic regression via genetic programming in the optimization of a controlled release pharmaceutical formulation. *Chemometrics Intelligent Laboratory Systems, 107*(1), 75-82.

Bentley, P. J., & Wakefield, J. P. (1996). *Hierarchical crossover in genetic algorithms.* Paper presented at the Proceedings of the 1st On-line Workshop on Soft Computing (WSC1).

Brun, Y., Serugendo, G. D. M., Gacek, C., Giese, H., Kienle, H., Litoiu, M., . . . Shaw, M. (2009). Engineering self-adaptive systems through feedback loops. In *Software engineering for self-adaptive systems* (pp. 48-70): Springer.

Brynjolfsson, E., Rock, D., & Syverson, C. (2018). Artificial intelligence and the modern productivity paradox: A clash of expectations and statistics. In *The Economics of Artificial Intelligence: An Agenda*: University of Chicago Press.

Cardona, A. B., Togelius, J., & Nelson, M. J. (2013, 2013). *Competitive coevolution in ms. pac-man.* Paper presented at the Evolutionary Computation (CEC), 2013 IEEE Congress on.

Castelli, M., Silva, S., & Vanneschi, L. (2015). A C framework for geometric semantic genetic programming. *Genetic Programming and Evolvable Machines, 16*(1), 73-81.

Cha, S.-H., & Tappert, C. C. (2009). A genetic algorithm for constructing compact binary decision trees. *Journal of pattern recognition research, 4*(1), 1-13.

Chai, B.-B., Huang, T., Zhuang, X., Zhao, Y., & Sklansky, J. (1996). Piecewise linear classifiers using binary tree structure and genetic algorithm. *Pattern Recognition, 29*(11), 1905-1917.

Cliff, D., & Miller, G. F. (1995, 1995). *Tracking the red queen: Measurements of adaptive progress in co-evolutionary simulations.* Paper presented at the European Conference on Artificial Life.

Davidson, J. W., Savic, D. A., & Walters, G. A. (2003). Symbolic and numerical regression: experiments and applications. *Information Sciences, 150*(1-2), 95-117.

de Jong, E., Stanley, K., & Wiegand, R. P. (2007). *Introductory tutorial on coevolution.* Paper presented at the Proceedings of the 9th annual conference companion on Genetic and evolutionary computation.

Deb, K., & Agrawal, R. B. J. C. s. (1995). Simulated binary crossover for continuous search space. *9*(2), 115-148.

Eiben, A. E., & Smith, J. E. (2015). *Introduction to Evolutionary Computing* (2nd ed. 2015 ed.). Berlin, Heidelberg: Springer Berlin Heidelberg.

Ficici, S. G., & Pollack, J. B. (2004). *Solution Concepts in Coevolutionary Algorithms.* Brandeis University Waltham, MA, (Dissertation/Thesis)

Funes, P., & Pollack, J. (2000). Measuring progress in coevolutionary competition. *From Animals to Animats, 6*, 450-459.

Gómez, J., Garcia, A., & Silva, C. (2005). *Cofre: A fuzzy rule coevolutionary approach for multiclass classification problems.* Paper presented at the 2005 IEEE Congress on Evolutionary Computation.

Gonçalves, J. F., de Magalhães Mendes, J. J., & Resende, M. c. G. (2005). A hybrid genetic algorithm for the job shop scheduling problem. *European journal of operational research, 167*(1), 77-95.

Gustafson, S., Burke, E. K., & Krasnogor, N. (2005). *On improving genetic programming for symbolic regression.* Paper presented at the 2005 IEEE Congress on Evolutionary Computation.

Hiew, B. Y., Tan, S. C., & Lim, W. S. (2017). Development of a Co-evolutionary Radial Basis Function Neural Classifier by a k-Random Opponents Topology. In *Emerging Trends in Neuro Engineering and Neural Computation* (pp. 207-217): Springer.

Hoai, N. X., McKay, R. I., Essam, D., & Chau, R. (2002). *Solving the symbolic regression problem with tree-adjunct grammar guided genetic programming: The comparative results.* Paper presented at the Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600).

Howard, L. M., & D'Angelo, D. J. (1995). The GA-P: A genetic algorithm and genetic programming hybrid. *IEEE expert, 10*(3), 11-15.

Hu, X.-B., & Di Paolo, E. (2009). An efficient genetic algorithm with uniform crossover for air traffic control. *Computers & Operations Research, 36*(1), 245-259.

Icke, I., & Bongard, J. C. (2013). *Improving genetic programming based symbolic regression using deterministic machine learning.* Paper presented at the 2013 IEEE Congress on Evolutionary Computation.

Jaśkowski, W., Krawiec, K., & Wieloch, B. (2008a). *Fitnessless coevolution.* Paper presented at the Proceedings of the 10th annual conference on Genetic and evolutionary computation.

Jaśkowski, W., Krawiec, K., & Wieloch, B. (2008b). *Winning ant wars: Evolving a human-competitive game strategy using fitnessless selection.* Paper presented at the European Conference on Genetic Programming.

Kao, Y.-T., & Zahara, E. (2008). A hybrid genetic algorithm and particle swarm optimization for multimodal functions. *Applied Soft Computing, 8*(2), 849-857.

Karaboga, D., & Basturk, B. (2008). On the performance of artificial bee colony (ABC) algorithm. *Applied Soft Computing, 8*(1), 687-697.

Karaboga, D., Ozturk, C., Karaboga, N., & Gorkemli, B. J. I. S. (2012). Artificial bee colony programming for symbolic regression. *209*, 1-15.

Kaya, Y., & Uyar, M. (2011). A novel crossover operator for genetic algorithms: ring crossover. *arXiv preprint arXiv*.

Keijzer, M. (2003). *Improving symbolic regression with interval arithmetic and linear scaling.* Paper presented at the European Conference on Genetic Programming.

Keijzer, M. (2004). Scaled symbolic regression. *5*(3), 259-269.

Kim, D. H., Abraham, A., & Cho, J. H. (2007). A hybrid genetic algorithm and bacterial foraging approach for global optimization. *Information Sciences, 177*(18), 3918-3937.

Kim, M. P., Suksompong, W., & Williams, V. V. (2017). Who can win a single-elimination tournament? *SIAM Journal on Discrete Mathematics, 31*(3), 1751-1764.

Koza, J. R. (1994). Genetic programming as a means for programming computers by natural selection. *Statistics and computing, 4*(2), 87-112.

Lee, C., & Antonsson, E. (2000). *Variable Length Genomes for Evolutionary Algorithms.* Paper presented at the GECCO.

Luke, S., & Panait, L. (2006). A comparison of bloat control methods for genetic programming. *Evolutionary computation, 14*(3), 309-344.

Luke, S., & Spector, L. (1997). A comparison of crossover and mutation in genetic programming. *Genetic Programming, 97*, 240-248.

McConaghy, T. (2011). FFX: Fast, scalable, deterministic symbolic regression technology. In *Genetic Programming Theory and Practice IX* (pp. 235-260): Springer.

McKay, B., Willis, M. J., & Barton, G. W. (1995). *Using a tree structured genetic algorithm to perform symbolic regression.* Paper presented at the First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications.

Miconi, T. (2009). *Why coevolution doesn't "work": superiority and progress in coevolution.* Paper presented at the European Conference on Genetic Programming.

Miconi, T., & Channon, A. (2006). *The n-strikes-out algorithm: A steady-state algorithm for coevolution.* Paper presented at the 2006 IEEE CONGRESS ON EVOLUTIONARY COMPUTATION, VOLS 1-6.

Panait, L., & Luke, S. (2002). *A comparative study of two competitive fitness functions.* Paper presented at the Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002).

Poli, R., Langdon, W. B., McPhee, N. F., & Koza, J. R. (2008). *A field guide to genetic programming*: Lulu. com.

Poli, R., & Page, J. (2000). Solving high-order boolean parity problems with smooth uniform crossover, sub-machine code GP and demes. *Genetic programming evolvable machines, 1*(1-2), 37-56.

Poon, P. W., & Carter, J. N. (1995). Genetic algorithm crossover operators for ordering applications. *Computers Operations Research, 22*(1), 135-147.

Poon, P. W., Carter, J. N. J. C., & Research, O. (1995). Genetic algorithm crossover operators for ordering applications. *22*(1), 135-147.

Popovici, E., Bucci, A., Wiegand, R. P., & De Jong, E. D. (2012). Coevolutionary principles. In *Handbook of natural computing* (pp. 987-1033): Springer.

Prügel-Bennetf, A. (2001). The mixing rate of different crossover operators. In *Foundations of Genetic Algorithms 6* (pp. 261-274): Elsevier.

Rahnamayan, S., Tizhoosh, H. R., & Salama, M. M. (2007). A novel population initialization method for accelerating evolutionary algorithms. *Computers & Mathematics with Applications, 53*(10), 1605-1614.

Ribeiro, J. C. B. (2008, 2008). *Search-based test case generation for object-oriented java software using strongly-typed genetic programming.* Paper presented at the Proceedings of the 10th annual conference companion on Genetic and evolutionary computation.

Rosin, C. D., & Belew, R. K. (1995, 1995). *Methods for Competitive Co-Evolution: Finding Opponents Worth Beating.* Paper presented at the ICGA.

Sagarna, R., Arcuri, A., & Yao, X. (2007, 2007). *Estimation of distribution algorithms for testing object oriented software.* Paper presented at the IEEE Congress on Evolutionary Computation.

Searson, D. P., Leahy, D. E., & Willis, M. J. (2010). *GPTIPS: an open source genetic programming toolbox for multigene symbolic regression.* Paper presented at the Proceedings of the International multiconference of engineers and computer scientists.

Semenkin, E., & Semenkina, M. (2012). *Self-configuring genetic algorithm with modified uniform crossover operator.* Paper presented at the International Conference in Swarm Intelligence.

Shayan, E., & Chittilappilly, A. (2004). Genetic algorithm for facilities layout problems based on slicing tree structure. *International Journal of Production Research, 42*(19), 4055-4067.

Sims, K. (1994). Evolving 3D morphology and behavior by competition. *Artificial Life, 1*(4), 353-372.

Smits, G. F., & Kotanchek, M. (2005). Pareto-front exploitation in symbolic regression. In *Genetic programming theory and practice II* (pp. 283-299): Springer.

Sokolov, A., & Whitley, D. (2005). *Unbiased tournament selection.* Paper presented at the GECCO.

Spears, W. M., & De Jong, K. D. (1995). *On the virtues of parameterized uniform crossover*. Retrieved from

Stanley, K., & Miikkulainen, R. (2002). *Continual coevolution through complexification.* Paper presented at the Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation.

Stanley, K., & Miikkulainen, R. (2004). Competitive coevolution through evolutionary complexification. *Journal of artificial intelligence research, 21*, 63-100.

Stanton, I., & Williams, V. V. (2011). *Manipulating stochastically generated single-elimination tournaments for nearly all players.* Paper presented at the International Workshop on Internet and Network Economics.

Szubert, M., Jaskowski, W., & Krawiec, K. (2009). *Coevolutionary temporal difference learning for Othello.* Paper presented at the 2009 IEEE Symposium on Computational Intelligence and Games.

Tan, T. G., Teo, J., & Lau, H. K. (2008). *Augmenting SPEA2 with K-Random competitive coevolution for enhanced evolutionary multi-objective optimization.* Paper presented at the 2008 International Symposium on Information Technology.

Tate, D. M., & Smith, A. E. (1993). *Expected Allele Coverage and the Role of Mutation in Genetic Algorithms.* Paper presented at the ICGA.

Vladislavleva, E. J., Smits, G. F., & Den Hertog, D. J. I. T. o. E. C. (2008). Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. *13*(2), 333-349.

Wang, C., Pastore, F., Goknil, A., Briand, L., & Iqbal, Z. (2015, 2015). *Automatic generation of system test cases from use case specifications.* Paper presented at the Proceedings of the 2015 International Symposium on Software Testing and Analysis.

Wappler, S., & Wegener, J. (2006, 2006). *Evolutionary unit testing of object-oriented software using a hybrid evolutionary algorithm.* Paper presented at the Evolutionary Computation, 2006. CEC 2006. IEEE Congress on.

Weimer, W., Nguyen, T., Le Goues, C., & Forrest, S. (2009, 2009). *Automatically finding patches using genetic programming.* Paper presented at the Proceedings of the 31st International Conference on Software Engineering.

Williams, E. A., & Crossley, W. A. (1998). Empirically-derived population size and mutation rate guidelines for a genetic algorithm with uniform crossover. In *Soft computing in engineering design and manufacturing* (pp. 163-172): Springer.

Wright, A. H. (1991). Genetic algorithms for real parameter optimization. In *Foundations of genetic algorithms* (Vol. 1, pp. 205-218): Elsevier.

Zelinka, I., Oplatkova, Z., & Nolle, L. (2005). Analytic programming–Symbolic regression by means of arbitrary evolutionary algorithms. *International Journal of Simulation: Systems, Science Technology, 6*(9), 44-56.

Zelinka, I., Oplatkova, Z., Nolle, L. J. I. J. o. S., Systems, Science, & Technology. (2005). Analytic programming–Symbolic regression by means of arbitrary evolutionary algorithms. *6*(9), 44-56.

9 APPENDIX

Strategy	Determinism	Description
AppendRandomOperation	R	*AppendRandomOperation* involves adding a section of a randomly generated tree to the given tree. This new program to be added to the original program may consist of one or more nodes that adhere to the STGP principles we outlined earlier. An "Add Program" strategy could replace a non-terminal with a new randomly generated program.
DeleteTerminal	R	Converts a random terminal to value 0. If the tree only contains the root, it sets the root to 0.
DeleteNonTerminal	R	Converts a random non-terminal to value 0. This operation always truncates the size of the tree depending on the non-terminal selected. Given a tree such as $(((x * 2)/x) + 3)$, if we select the $/$ (divide) non-terminal as the non-terminal to delete, the $(x * 2)$ portion will be set 0. The resulting expression will be $((0/x) + 3)$.
MutateTerminal	R	Changes a terminal value to another randomly generated terminal value. For example, $2 \rightarrow 8$ or $6 \rightarrow x$.
MutateNonTerminal	R	Changes a non-terminal value to another randomly generated non-terminal value. For example, $\div \rightarrow +$ or $\times \rightarrow -$.
Skip	D	Skip strategy is a fully deterministic strategy that can be employed by either type of individual to forfeit applying an actual strategy that modifies an underlying tree. A scenario where skip would make sense is if the test upon encountering its current solution is optimal (global optima) realizes further mutations may throw the individual off the global optima.
FellTree	D	Destroys the entire tree, leaving a 0 as the root terminal.
MultX	D	Takes the input tree and multiplies it by x.
AddX	D	Takes the input tree and adds x.
SubX	D	Takes the input tree and subtracts it by x.
DivX	D	Takes the input tree and divides it by x.
MultC	D	Takes the input tree and multiplies by a randomly generated constant.
AddC	D	Takes the input tree and adds it to a randomly generated constant.
SubC	D	Takes the input tree and subtracts a randomly generated constant.
DivC	D	Takes the input tree and divides it by a randomly generated constant.

Table 9-i: *List of Strategies - A comprehensive list of strategies and their determinism and descriptions. In the determinism column, D represents deterministic strategies, where R represents random or non-deterministic strategies.*