ABSTRACT

Title of Dissertation: Issues in Scaling Genetic Programming:
Breeding Strategies, Tree Generation, and Code Bloat

Sean Luke, Doctor of Philosophy, 2000

Dissertation directed by: Professor James Hendler
Department of Computer Science

Genetic Programming is an evolutionary computation technique which searches for those computer programs that best solve a given problem. As genetic programming is applied to increasingly difficult problems, its effectiveness is hampered by the tendency of candidate program solutions to grow in size independent of any corresponding increases in quality. This bloat in solutions slows the search process, interferes with genetic programming's searching, and ultimately consumes all available memory. The challenge for scaling up genetic programming is to find the best solutions possible before bloat puts a stop to evolution. This can be tackled either by finding better solutions more rapidly, or by taking measures to delay bloat as long as possible.

This thesis discusses issues both in speeding the search process and in delaying bloat in order to scale genetic programming to tackle harder problems. It describes evolutionary computation and genetic programming, and details the application of genetic programming to cooperative robot soccer and to language induction. The thesis then compares genetic programming breeding strategies, showing the conditions under which each strategy produces better individuals with less bloating. It then analyzes the tree growth properties of the standard tree generation algorithms used, and proposes new, fast algorithms which give the user better control over tree size. Lastly, it presents evidence which directly contradicts existing bloat theories, and gives a more general theory of code growth, showing that the issue is more complicated than it first appears.

# Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat

**Sean Luke**

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2000

Advisory Committee:

Professor James Hendler, Chair and Advisor
Professor James Reggia
Professor Lee Spector
Professor Chau-Wen Tseng
Professor Dana Nau
Professor Stephen Mount

This version of this text has been reformatted
to reduce the total number of pages required to print.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Historically, much of artificial intelligence has focused on the central topic of *heuristic search*, where a computer program hunts for answers to some problem, using heuristics to suggest possibly fertile areas for searching. For many such problems, these heuristics light a straight, easy path to the best solutions. For other problems there is no straight path, but at least it's clear how to go about finding the best solutions, even if in the process you run into some dead ends. Then there are problems where there isn't even a path to follow to find a good answer but, to paraphrase Justice Potter Stevens, "you know one when you see it."

Problems in this last set are tough. In the worst case, many of these problems can be solved only through random, brute-force search. But fortunately, many problems of interest in this set have a feature which can be exploited to outperform random search. This feature is a positive correlation between the form that candidate solutions take and their resultant quality as solutions. That is, for many problems, similar solutions often perform similarly. The family of search procedures whose heuristic takes advantage of this feature are collectively known as *stochastic search algorithms*. Such a heuristic is a fairly dim torch, and as such stochastic search algorithms are not guaranteed to find the optimal solutions. But they can often outperform random search on many problems of interest, and are useful tools for attacking those many problems where no stronger heuristic is available.

This thesis focuses on a specific subset of stochastic search approaches called *evolutionary computation* (EC), and especially one evolutionary computation technique known as *genetic programming* (GP). However, many of the results of the thesis have a wider applicability to other stochastic search algorithms, including simulated annealing, tabu search, and hillclimbing.

Evolutionary computation takes its inspiration from natural selection and genetics, applying a kind of "directed selection" to the search for problem solutions. EC begins with a population of randomly-generated individuals (candidate solutions). It then tests these individuals and assesses their quality. The better ones are then selected to breed and create new individuals which in turn are tested, selected, and bred. This cycle continues until a sufficiently good solution is found for the problem, or until time or other resources are exhausted.

## 1.1 Genetic Programming and Bloat

Most stochastic search techniques traditionally operate over parameterized solution spaces, that is, where candidate solutions are represented with vectors of a fixed length, and usually there is a finite number of them. Within evolutionary computation, such techniques include genetic algorithms (GA) and evolutionary strategies (ES). However, an increasing number of interesting and difficult problems have solutions which are best described using arbitrary-length representations. Searching in these problems is made more complex by the fact that they effectively have an infinite, although usually countable, number of candidate solutions. Some examples of areas with problems for which variable-length representations of solutions would be useful include the creation of finite-state automata, production rule sets, neural networks, and computer programs.

Genetic programming is the most common form of variable-length evolutionary computation. The goal of genetic programming is usually to find a good-performing computer program for a given task. Such a program can be of any size and form, within the constraints of the problem at hand. For a genetic programming system to find such a computer program, the experimenter need provide only two things: a description of how to form candidate programs, and a function which assesses candidate programs and assigns grades to them.

Recent difficulties in scaling genetic programming have highlighted a roadblock faced when dealing with variable-length representations: the candidate solutions being considered tend to grow bigger and bigger, without bound, and independent of increases in quality. For example, it is not uncommon for genetic programming to start off considering candidate solutions only five units long, and very soon wind up considering candidates many thousands of units long, with little to show for the extra effort.

In the genetic programming world this problem is known as *bloat*. Bloat presents a serious problem in scaling GP to larger and more difficult problems. First, bloat consumes computing resources, making the search process slower and slower, and eventually forcing it to stop when all available resources have been exhausted. Second, bloated candidate solutions are often more difficult to modify in meaningful ways, hampering the ability of GP to breed and discover better solutions. Third, bloating can slow the grade-assessment process. In a very real sense, bloating makes genetic programming a race against time, to find the best solution possible before bloat puts an effective stop to the search.

As genetic programming tackles harder, real-world problems, the size and complexity of their respective problem spaces grows as well, and so too the amount of time necessary to search them. This makes such problems easy prey for bloat, which sets in long before GP can find sufficiently good solutions. Experimenters must either take measures to counteract bloat, sacrifice runtime in favor of ever larger populations (moving GP in the direction of random search), or attempt to find good solutions faster, before bloat sets in.

## 1.2 Contributions

This thesis presents three significant contributions to the body of GP work:

- It presents a very large and comprehensive comparison of the common ways genetic programming can modify candidate programs to create new ones (subtree crossover and subtree mutation). This comparison covers four problem domains and two hundred common parameter settings, and reveals those areas where each breeding technique yields higher fitness and lower bloat. These areas are problem-specific, but do follow interesting trends. At 3.3 billion individual evaluations, this is the largest single comparison experiment in the genetic programming literature.

- It formally analyzes (GROW), the traditional algorithm used in genetic programming to create and mutate the trees which form GP individuals. The results of this analysis show that the GROW algorithm has very serious bloating characteristics, necessitating ad-hoc techniques to control the size of its generated trees. The thesis shows conditions under which subtree mutation can cause bloating in the population even with no driving selective force. It then introduces and analyzes two new algorithms, PTC1 and PTC2, which give the user control over tree size (which can counter bloat) and uniform tree node distribution.

- It gives data results which directly contradict the prevailing theories explaining the causes of bloat in genetic programming. This is the first such data of its kind. Supported by this and other data, the thesis proposes a more general theory of bloat applicable to most common genetic programming domains. These results shed light on why existing techniques work (or don't), giving further insight to the search for a cure for bloat.

Additionally, this thesis has two other items important to the GP community:

- It presents two case studies in genetic programming, both with nontrivial and difficult real-world problems to solve. In one of these case studies, evolving virtual soccer robot programs, the results of the evolutionary computation run performed surprisingly well against hand-coded programs in an international soccer robotics competition. Both case studies illustrate the problem of bloat in complex problem domains.

- It introduces a new genetic programming and evolutionary computation system, ECJ, designed to make possible complex genetic programming experiments, including ones discussed later in this thesis. ECJ provides a modular experimental framework and a rich set of built-in evolutionary computation features. As of July 2000, ECJ is the most sophisticated genetic programming system available in the public domain.

## 1.3 Thesis Overview

The remainder of this thesis proceeds as follows. The first four chapters introduce genetic programming and its issues and implementations, and provide a review of the existing literature. Chapter 2 gives an introduction to evolutionary computation. Chapter 3 describes genetic programming, its variants, and example test problems. Chapter 4 reviews existing issues in genetic programming and their solutions in the literature.

The next two chapters discuss nontrivial applications of genetic programming, illustrating the diverse areas in which the technique is now being applied, and also highlighting the problem of bloat. Chapter 5 discusses my application of genetic programming to robotic soccer, a very difficult and challenging domain. Chapter 6 introduces a novel variant of genetic programming of my devising, *edge encoding*, used to find sparse networks such as electrical circuits or finite-state automata.

The last four chapters present a detailed analysis of the dynamics of bloat in genetic programming, and a comparison of techniques to achieve the best results with the minimum amount of bloat. Chapter 7 compares mutation and crossover, the two most popular breeding methods for genetic programming, and discovers when and where each yields the best fitness and lowest bloat results. Chapter 8 gives a formal analysis of genetic programming's tree-creation algorithm and its contributions to bloat, and suggests alternative approaches. Chapter 9 shows that the prevailing wisdom about tree bloat (that it is caused by so-called "introns") is wrong, and proposes a new theory in its stead. Lastly, Chapter 10 discusses conclusions and future work.

Appendix A lists the most common kinds of introns and other structures and breeding events fingered as culprits in tree bloat. Appendix B introduces a novel genetic programming system, ECJ [Luke 2000], designed as a flexible experimental platform, and with which many of the experiments in this thesis were performed. Appendix C provides the full set of figures and tables for statistics discussed in Chapter 9.

## Chapter 2

# Evolutionary Computation

Evolutionary Computation (EC) searches for good solutions to problems by trying large numbers of candidate solutions, selecting the "better" ones, modifying them, and producing new candidate solutions to test. EC is a direct descendent of the artificial intelligence community: the earliest major work in evolutionary computation [Fogel, Owens, and Walsh 1966] evolved finite state automata as a novel machine learning mechanism.

Because it is inspired by natural selection and genetics, evolutionary computation borrows much of its vernacular from genetics, cellular biology, and evolutionary theory. In EC, a candidate solution is known as an *individual*. The pool of current individuals in the system is collectively known as the *population*. This population may, depending on the nature of the problem being solved, be broken into several *subpopulations*. The actual encoding of an individual's solution is known as its *genome* (occasionally *chromosome*). The solution's representation when undergoing modification is known as the individual's *genotype*. The way the solution operates when tested in the problem environment is known as the individual's *phenotype*. When individuals are modified to produce new individuals, they are said to be *breeding*. During testing an individual receives a grade, known as its *fitness*, which indicates how good a solution it is. The period in which the individual is evaluated and assigned a fitness is known as *fitness assessment*. When a population has been entirely replaced by children, the new population is known as the next *generation*. The whole process of finding an optimal solution is known as *evolving* a solution.

There are two common high-level, abstract procedures used in evolutionary computation. The traditional approach is *generational* EC. An increasingly popular newcomer is *steady-state* EC.

## 2.1 Generational Evolutionary Computation

Generational EC is straightforward. Initially, a population of individuals is created at random. Then the fitness of the individuals in the population is assessed. The better individuals are selected to breed, forming a new population. The new population replaces the original population, and the process repeats at the fitness assessment step. The cycle continues until an ideal individual is discovered or resources are exhausted. The algorithm returns the highest-fit individual it discovered during its run.

**Algorithm 1** Simple Generational Evolutionary Computation

Population $P \leftarrow$ Initialize($n$)
Individual $best \leftarrow$ nil
Repeat $m$ times:
    For each individual $p_i \in P$,
        AssessFitness($p_i$)
        If the fitness of $p_i$ is assessed as optimal,
            Halt and return $p_i$
        If $best$ is nil or the fitness of $p_i$ is better than the fitness of $best$,
            $best \leftarrow p_i$

    Population $Q \leftarrow \emptyset$
    Until $\|Q\| = \|P\|$
        $Q \leftarrow Q \cup$ Breed($P, \|P\| - \|Q\|$)
    $P \leftarrow Q$
Return $best$

Generational EC relies on three user-provided functions and two user-provided parameters. $n$ determines the size of the population. $m$ determines the number of *generations*, that is, the number of times the population is assessed and bred until the algorithm gives up. Initialize($n$) produces a set of $n$ initial individuals, usually generated at random. AssessFitness($p_i$) assigns a fitness (a grade) to the individual $p_i$, which reflects its performance at solving the problem at hand. Breed($P, \|P\| - \|Q\|$) selects individuals from $P$ based on their fitness, copies them, then modifies the copies to produce new candidate solutions. Breed returns at most $\|P\| - \|Q\|$ new individuals (but typically only 1 or 2), which are then added to the next-generation population $Q$.

## 2.2 Steady-State Evolutionary Computation

Another popular approach, Steady-State EC, requires the same three user-provided functions as the generational approach. However, steady-state EC breeds and replaces only a few individuals at a time in a population, never replacing the whole population in one fell swoop. This permits children to compete directly with parents.

**Algorithm 2** Simple Steady-State Evolutionary Computation

Population $P \leftarrow$ Initialize($n$)
Individual $best \leftarrow$ nil
For each individual $p_i \in P$,
    AssessFitness($p_i$)
    If the fitness of $p_i$ is assessed as optimal,
        Halt and return $p_i$
    If $best$ is nil or the fitness of $p_i$ is better than the fitness of $best$,
        $best \leftarrow p_i$
Repeat $m$ times:
    Population $Q \leftarrow \emptyset$
    Until $\|Q\| = r$
        $Q \leftarrow Q \cup$ Breed($P, r - \|Q\|$)
    For each individual $q_i \in Q$,
        AssessFitness($q_i$)
        If the fitness of $q_i$ is assessed as optimal,
            Halt and return $q_i$
        If the fitness of $q_i$ is better than the fitness of $best$,
•            $best \leftarrow q_i$
    Repeat $\|Q\|$ times:
        Individual $s \leftarrow$ SelectForRemoval($P$)
        $P \leftarrow P \setminus \{q\}$
○    $P \leftarrow P \cup Q$
Return $best$

One variant form of the algorithm moves the line marked $\circ$ to just below the line marked $\bullet$. This has the effect of causing new individuals to immediately compete for space in the population as soon as they are created.

The number of individuals replaced is determined by an additional user-provided parameter $r < \|P\|$. An additional user-provided function, SelectForRemoval($P$), selects a single individual, usually one that has a poor fitness, for elimination from the population. Note that if $r = \|P\|$, this algorithm is effectively generational EC. $m$ is more fine-grained in steady-state EC than in generational EC, since one iteration may produce just a single individual rather than a whole population. Thus in steady-state EC, a "generation" is usually defined as the number of iterations necessary to evaluate $\|P\|$ individuals.

## 2.3 Other Approaches

The algorithms described above share two constraints in common. First, there is only one population evolved at a time. Second, individuals are assessed independently of one another.

One way to loosen these constraints is to use individuals as part of the assessment of other individuals. For example, to evolve checker players, individuals in the population might receive a fitness assessment based on how well they perform, not against some benchmark standard, but against other individuals in the population acting as opponents. Hopefully, as the population improves, the opponents gradually grow more difficult. Loosening this constraint results in a family of evolutionary computation algorithms called *competitive fitness* [Angeline and Pollack 1993].

Another loosening of constraints is to evolve more than one population at a time. This is a common approach to parallelizing evolutionary computation. The popular way to do this is known as the *island model* [Cohoon *et al.* 1987], where evolution occurs in multiple parallel subpopulations, known as *demes*, a notion borrowed from biology [Wright 1964]. In the island model, demes evolve mostly separately, with occasional "migrations" of highly-fit individuals between the demes.

A final common use for multiple populations is a specific form of competitive fitness known as *coevolution*. In coevolution, individuals in populations are assessed through competition against other populations, but breeding is restricted to within each population. One application of coevolution might be to evolve predators and prey. One population of predators is assessed based on their success at hunting down a second population of prey. The prey in turn are assessed based on their success at avoiding the predators. See [Angeline and Pollack 1993] for a thorough review of coevolution techniques.

## 2.4 Fitness Assessment

The user-provided function AssessFitness($p_i$) is responsible for testing the individual and assigning it a grade (its fitness). In most forms of evolutionary computation, an individual's assessed fitness is a single real-valued parameter which reflects its success at solving the problem at hand. This is an entirely user-determined value. In this thesis, AssessFitness($p_i$) assigns to individual $p_i$ a *standardized fitness* $f_i^s$, defined as a fitness parameter in the range $[0, +\infty)$, where 0 represents the optimum and $\infty$ is worse than the worst possible fitness. The individual's *adjusted fitness* $f_i^a$, defined as $f_i^a = \frac{1}{1+f_i^s}$, maps the fitness into the interval $(0, 1]$, where 0 is worse than the worst fitness and 1 is the optimum.

## 2.5 Selection and Breeding

Once individuals have had their fitnesses assessed, they may be selected and bred to form the next generation in the evolution cycle, through repeated application of Breed(...). This function usually selects one or two individuals from the old population, copies them, modifies them, and returns the modified copies for addition to the new population. Evolutionary computation is replete with a variety of ways to perform Breed(...), but it

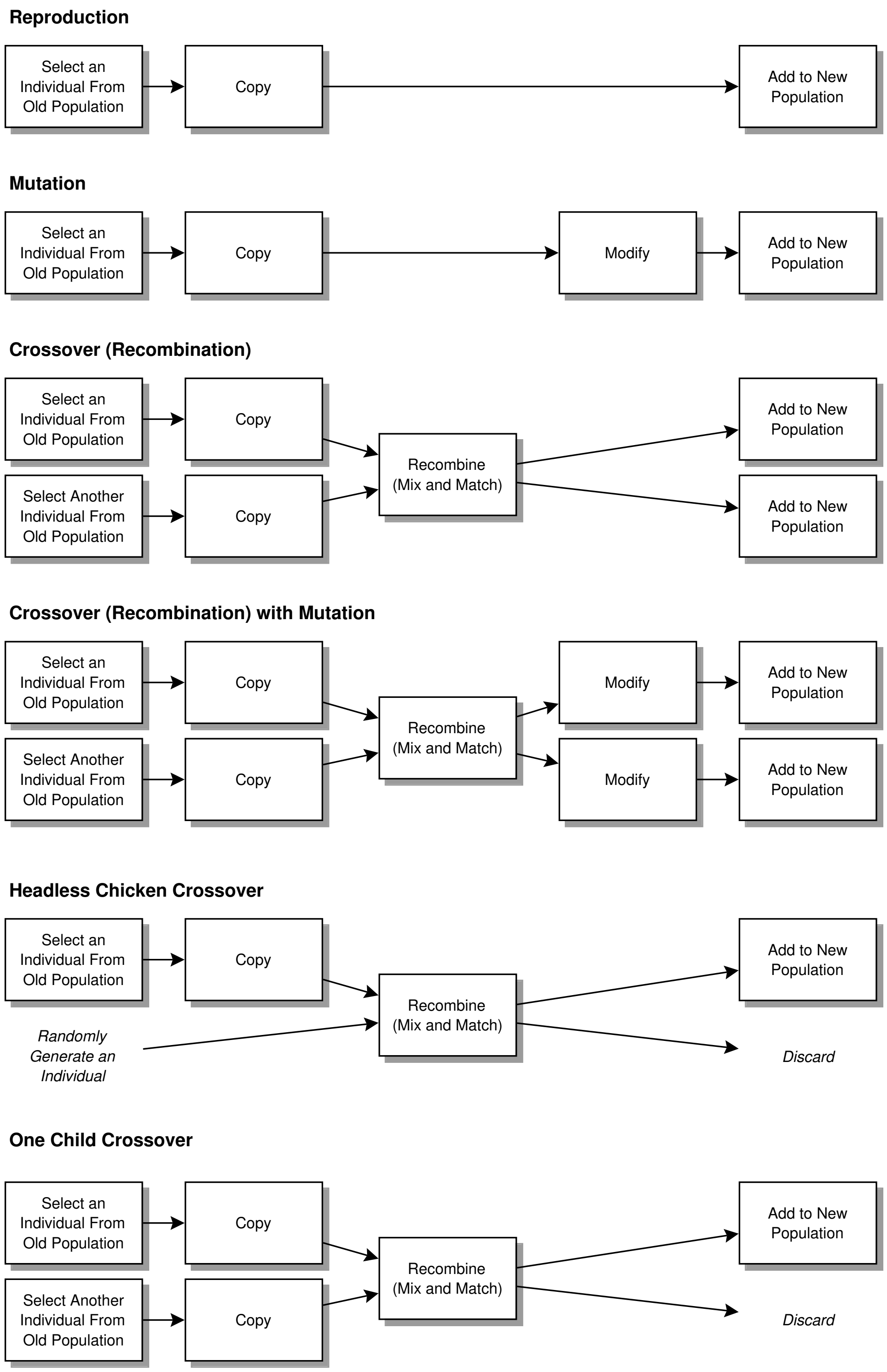



Figure 2.1: Six Breeding Methods in Evolutionary Computation

is usually done with through the application of one or more *breeding operators* to individuals in the population. Three common operators are *reproduction*, *crossover*, and *mutation*. These, plus three others discussed in this thesis, are listed in Figure 2.1. As shown in the figure, individuals are bred using combinations of selection, copying (asexual reproduction — duplicating the individual), recombining (sexual reproduction — mixing and matching two individuals), and modifying.

In order to breed individuals, they must first be selected from among the population according to their fitness. There are several common selection strategies in use:

**Fitness-Proportional Selection** This selection method, due to [Holland 1975], normalizes all the fitnesses in the population. These normalized fitnesses then become the probabilities that their respective individuals will be selected. Fitnesses may be transformed in some way prior to normalization; for example, [Koza 1992] normalizes the adjusted fitness rather than the standardized fitness.

**Ranked Selection** One of the problems with fitness-proportional selection is that it is based directly on the fitness. Assessed fitnesses are rarely an accurate measure of how "good" an individual really is. Another approach which addresses this issue is to rank individuals by their fitness, and use that ranking to determine selection probability. In *linear ranking* [Grefenstette and Baker 1989; Whitley 1989], individuals are first sorted according to their fitness values, with the first individual being the worst and the last individual being the best. Each individual is then selected with a probability based on some linear function of its sorted rank. This is usually done by assigning to the individual at rank $i$ a probability of selection

$$p_i = \frac{1}{\|P\|}(2 - c + (2c-2)\frac{i-1}{\|P\|-1})$$

where $\|P\|$ is the size of the population $P$, and $1 \leq c \leq 2$ is the *selection bias*: higher values of $c$ cause the system to focus more on selecting only the better individuals. The best individual in the population is thus selected with the probability $\frac{c}{\|P\|}$; the worst individual is selected with the probability $\frac{2-c}{\|P\|}$.

**Tournament Selection** The selection mechanism used in this thesis, *tournament selection*, is popular because it is simple, fast, and has well-understood statistical properties. In tournament selection, a pool of $n$ individuals are picked at random from the population. These are independent choices: an individual may be chosen more than once. Then tournament selection selects the individual with the highest fitness in this pool. Clearly, the larger the value $n$, the more directed this method is at picking highly-fit individuals. On the other hand, if $n = 1$, then the method selects individuals totally at random. Popular values for $n$ include 2 and 7. 2 is the standard number for genetic algorithm literature, and is not very selective. 7 is used widely in the genetic programming literature, and is relatively highly selective.

**Truncation Selection** In truncation selection, the next generation is formed from breeding only the best individuals in the population. One form of truncation selection, $(\mu, \lambda)$ selection, works as follows. Let the population size $\lambda = k\mu$, where $k$ and $\mu$ are positive integers. The $\mu$ best individuals in the population are "selected". Each individual in this group is then used to produce $k$ new individuals in the next generation. In a variant form, $(\mu + \lambda)$ selection, $\mu$ individuals are "selected" from the union of the population and the $\mu$ parents which had created that population previously. $(\mu, \lambda)$ and $(\mu + \lambda)$ selection are described further in [Schwefel 1977].

**Selection with Multiple Objectives** One area of increasing interest in evolutionary computation is searching for individuals which best match not one but several independent fitness criteria: for example, evolving a robot which runs the fastest and also uses the least energy. In this multiple-objective fitness assessment, an individual receives separate assessments in each of the criteria of interest, and the system must determine how to select individuals based on some function of these criteria. There are two common strategies for doing this.

The first strategy is to join the individual's assessments into one *aggregate fitness* by which the individual is selected. The most straightforward aggregation approach is simply to add the various assessments as a weighted sum [Hajela and Lin 1992]. Another way to do this is to set the standardized fitness to the maximum (worst) of the various fitness assessments [Wilson and Macleod 1993]. A third aggregation technique known as *lexicographic ordering* is used with ranked and tournament selections [Fourman 1985]. Lexicographic ordering assumes that there is an order of importance among the criteria. Individuals are first sorted (for ranking or tournaments) by the most important criterion. Ties are then broken by ranking by the second criterion, then the third criterion, etc.

The second strategy, also applicable to ranked and tournament selections, is to order individuals by their *pareto ranking*. If an individual is superior to another individual in all criteria, the first individual is said to *dominate* the second individual. Pareto ranking is a partial-order among the individuals based on who dominates whom. There are two common techniques which use pareto ranking. The first technique, due to [Goldberg 1989], is to assign a standardized fitness of 1 to all the individuals who are not dominated by anyone else in the population. Then these individuals are removed from consideration, and a standardized fitness of 2 is assigned to all the individuals who are not dominated by the remaining population. These individuals are then removed from consideration, and so on. This ranks individuals in layers based on their domination of others. The second technique simply sets an individual's standardized fitness as the number of individuals in the population which dominate the individual [Fonseca and Fleming 1993].

## 2.6 Forms of Evolutionary Computation

Evolutionary computation has two major camps which break down over philosophic differences about the breeding strategy to be used.

**Sexual Reproduction (Crossover)** The largest camp, *genetic algorithms* (GA) [Holland 1975] emphasizes crossover: mutation is relegated to the custodial duty of making certain that features cannot be completely eliminated from the population forever. GA usually evolves individuals with fixed-length vector genomes, and so searches over problem spaces of a fixed number of parameters. Only occasionally does GA use variable-length genomes (discussed in Section 4.1) or significant amounts of mutation. GA vectors have traditionally been bitstrings, but recent work has used vectors of integers or real-valued numbers. The article of faith behind GA crossover is the GA *building-block hypothesis*, which suggests that subparts of individuals ("building blocks") can be transfered from individual to individual through crossover, spreading good building blocks throughout the population as evolution progresses. This notion is founded on a theoretical underpinning known as the *schema theorem* [Holland 1975], which suggests that through selection, crossover, and mutation, good *schemata* (a formalism for building blocks) are expected gain dominance in the population at an exponential rate.

**Asexual Reproduction (Mutation)** In contrast, *evolutionary programming* (EP) [Fogel, Owens, and Walsh 1966] argues that mutation should be the sole breeding mechanism. EP has no traditional genome representation — it proposes evolving whatever genome structure is most suitable for the task, as long as a mutation operator can be devised to modify it. EP is the oldest evolutionary computation field, and was originally devised to evolve individuals in the form of finite-state automata. These early experiments mutated individuals by directly adding and deleting vertices and edges to individuals' FSA graphs.

A related field, *evolution strategies* (ES) [Rechenberg 1973], searches over fixed parameter spaces as GA does. ES has traditionally applied gaussian mutation to fixed-length vectors of real-valued numbers, using the $(\mu, \lambda)$ or $(\mu + \lambda)$ selection schemes. ES often applies the $\frac{1}{5}$ *rule* to determiningg the variance of the gaussian distribution used in mutation: if more than one fifth of new children are fitter than their parents, then the variance is increased to prevent the search from exploiting local improvements too much. Conversely, if fewer than one fifth of new children are fitter than their parents, then the variance is decreased to keep the search from exploring too widely. ES also can use *self-adaptive mutation*, adding to each individual one or two additional parameters which indicate how mutation should be applied to the individual. In this way, individuals can evolve not only their candidate solutions but the way in which the solutions are to be modified. ES traditionally uses only mutation, though evolution strategies researchers have recently adopted forms of crossover.

**Other Forms** There are also two important newcomers whose communities are distinguished by the kinds of problems they attack.

*Classifier systems* evolve sets of rules for machine learning and classification. LCS systems store rules in the form of *situation→action*, where *situation* and *action* are both binary strings, the first representing the world situation that causes the rule to fire, and the second representing the action to be taken when the rule fires. The aim is to find a set of rules which produce the best actions for all relevant world situations. There are two general kinds of classifier systems. In the *Pitt approach* [DeJong 1989], an individual consists of a set of rules, and the system searches for the individual whose rules collectively classify the problem best. In the *Michigan approach* [Goldberg 1989] each individual in the population is a single rule, and system evolves a ruleset collectively using the whole population together.

Lastly, *genetic programming* (GP) [Koza 1992] evolves symbolic computer programs or other algorithmic functions. Genetic programming is discussed at length in the next chapter.

## Chapter 3

# Genetic Programming

Much of evolutionary computation is devoted to evolving solutions to parameterized problem domains. In contrast, genetic programming (GP) is an evolutionary computation research community interested in a very different challenge: evolving actual computer programs to solve some computational task. Because of its special charge, GP mostly uses variable-length representations. By far the most popular such representation, and the one discussed in this thesis, describes computer programs as Lisp s-expressions, which are evaluated by directly executing them in a Lisp interpreter. This approach was originally proposed by [Cramer 1985], though most standard features of this representation are due to [Koza 1992].

## 3.1 Genetic Programming Individuals

Though genetic programming prints its individuals as s-expressions, it represents them internally as trees of named nodes. In genetic programming parlance, leaf nodes in the tree are known as *terminals* and nonleaf nodes are known as *nonterminals*. The mapping from tree to s-expression is straightforward: a terminal is a Lisp atom or a Lisp function with no arguments, and a nonleaf node is a function whose arguments are children to that node. That is, a nonterminal named "foo" appears in Lisp form as (foo *arg [arg*]*), where *arg* are subexpressions, one for each child subtree of the foo node. A terminal node named "bar" will always appear in Lisp s-expression form as bar (without parentheses). Some terminals are best thought of as functions with no child arguments — that is, (bar) — but will still be shown as bar to maintain consistency.

Figure 3.1 shows examples of GP trees and their equivalent s-expressions. In this thesis I refer to tree nodes both as "nodes" and also as "functions".

Depending on the problem being solved, an individual may be a single tree, or a forest of trees. The experimenter must provide the genetic programming system with a primordial soup of basic tree nodes from which to build its program trees. For each tree in an individual, the experimenter provides a *function set*, a collection of node templates from which GP can copy nodes for use in its trees. The experimenter is responsible for creating the custom machinery which permits these nodes to be evaluated as the appropriate Lisp-like functions, variables, and constants they represent.

In GP, the user-provided AssessFitness($p_i$) function usually assesses the fitness of $p_i$ by directly executing it in some problem domain as if it were an actual Lisp s-expression program, and then examining the result.

**Initial Tree Generation** At the beginning of the evolution process, initial individuals must be generated at random. Genetic programming creates these individuals' trees by applying a tree generation algorithm to each tree's function set. Tree generation algorithms work by selecting and copying nodes from the templates in the function set, then hanging the copied nodes together to form the tree. The three traditional tree-generation algorithms are GROW, FULL, and RAMPED HALF-AND-HALF.

GROW begins with a set of functions $F$ to place as nodes in the GROW randomly selects a root from the full set of functions (both terminals and nonterminals), then fills the root's arguments with random functions, then *their* arguments with random functions, and so on. A common variant is shown below:

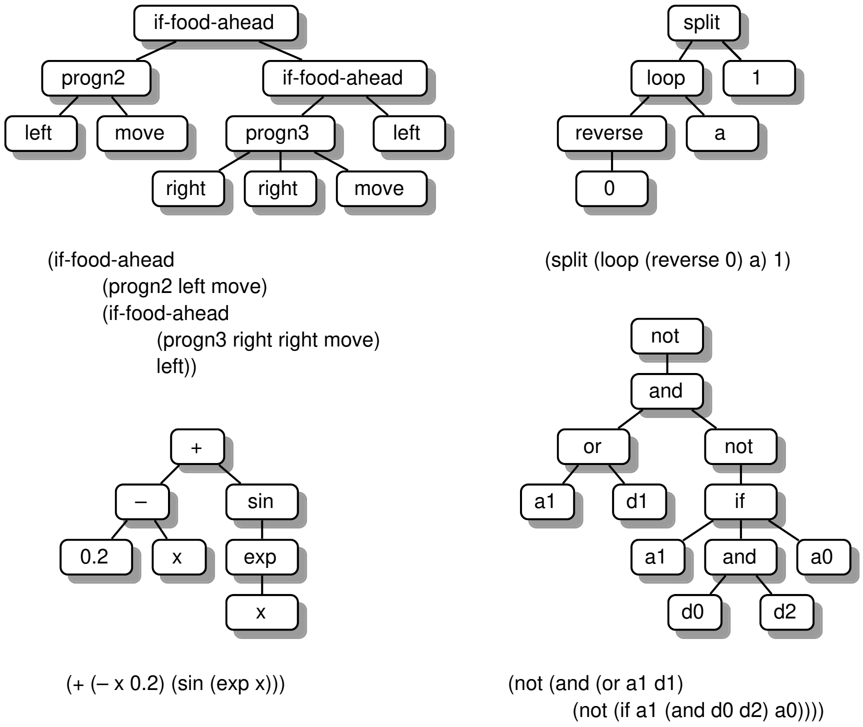



Figure 3.1: Four Example Genetic Programming Trees and Their Equivalent Lisp s-expression Forms

**Algorithm 3** GROW

*Given:*
  maximum depth bound $D$
  function set $F$ consisting of nonterminal set $N$ and terminal set $T$
*Do:*
  New tree $\mathbf{T}$ = GROW($0$)

*GROW*(depth $d$)
  *Returns:* a tree of depth $\leq D - d$

  If $d = D$, return a random terminal from $T$
  Else
○    Choose a random function $f$ from $F$
    If $f$ is a terminal, return $f$
    Else
      For each argument $a$ of $f$,
        Fill $a$ with GROW($d+1$)
      Return $f$ with filled arguments

GROW's companion algorithm (FULL) is used to generate full trees. It differs from GROW only in the line marked ○, where $f$ is chosen at random from $N$, not $F$. That is, $f$ is always a nonterminal. RAMPED HALF-AND-HALF randomly does either FULL or GROW. In most GP implementations the depth bound $D$ is picked randomly at tree-generation time from some range of valid depth bounds.

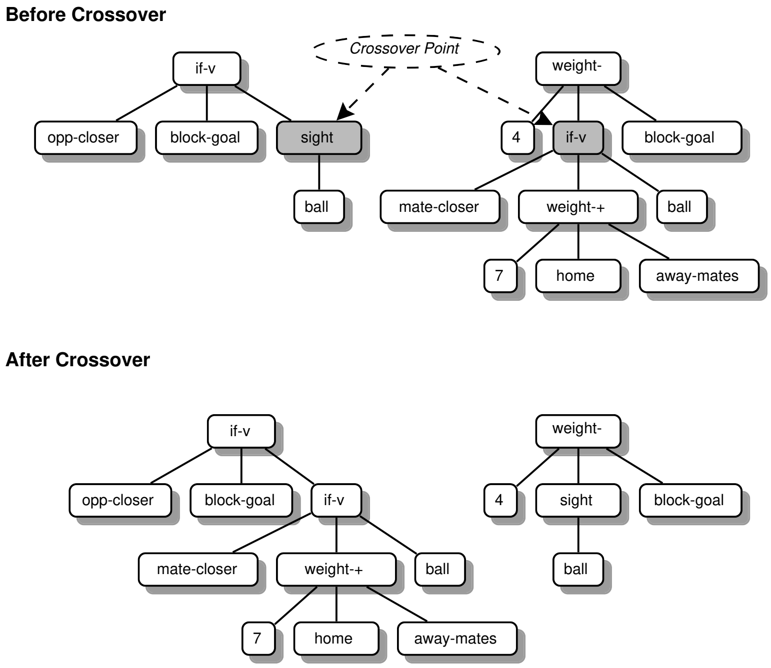



Figure 3.2: Subtree Crossover

Some variants of GROW permit the user to specify a maximum tree size $S$. They enforce this either by producing trees repeatedly until one of size less than $S$ is created, or by keeping track of the number of created nodes (so far) plus the number of unfilled arguments; when this total exceeds $S$, only terminals may fill arguments from then on (see [Chellapilla 1997] for an interesting example).

**Generating the Initial Population** In most cases, the Initialize($n$) function forms a population by generating individuals at random until it has collected $n$ unique individuals. In this thesis, individuals' trees are initially generated with the RAMPED HALF-AND-HALF algorithm, with a depth bound range from 2 to 6 inclusive, unless otherwise noted.

## 3.2 Genetic Programming Breeding

In standard genetic programming, Breed(...) picks from among the first three breeding strategies shown in Figure 2.1 (reproduction, mutation, and crossover) then applies one strategy to selected individuals and returns the results. Reproduction and mutation will return one individual, while crossover returns two. Traditionally, genetic programming chooses reproduction 10% of the time and crossover 90% of the time, but the experiments in this thesis use a several different probability settings. Breed(...) is repeatedly called until the new population is completely filled with newly-bred individuals.

To perform crossover and mutation, genetic programming must implement their respective "recombine" and "modify" steps as shown in Figure 2.1. The standard recombination method in genetic programming, *subtree crossover*, starts with two individuals selected and copied from the old population. Classically, a random point is selected within one tree of each copied individual with nonterminals selected 90% of the time and terminals selected 10% of the time. Then crossover swaps the subtrees rooted at these two nodes and returns the modified copies. If the crossover process results in a tree greater than a maximum depth bound (except where noted, the bound is 17), then the modified child is discarded and its parent (the tree into which a subtree was inserted to

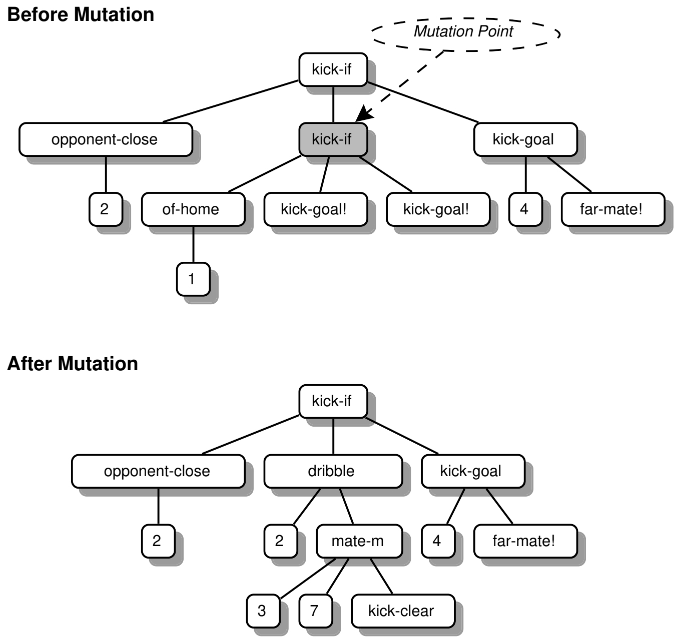



Figure 3.3: Subtree Mutation

form the child) is simply copied through reproduction. Crossover may only occur if two trees share the same function set. This process is illustrated in Figure 3.2.

The most common "modify" step uses *subtree mutation*, shown in Figure 3.3. Subtree mutation starts with a single individual selected and copied from the old population. One node is selected from among of the copied individual's trees, using the same node-selection technique as described for subtree crossover. The subtree rooted at this node is removed and replaced with a randomly-generated subtree, using the GROW algorithm and the appropriate function set for the tree. If mutation results in a tree greater than the maximum depth (again, 17), then the copy is discarded and its parent is reproduced instead.

## 3.3 Extensions to Genetic Programming

Three popular extensions to genetic programming are used or discussed in this thesis. *Strongly-typed genetic programming* (STGP) [Montana 1995] adds type constraints to breeding and tree generation. *Ephemeral random constants* provide numeric constants in Lisp s-expressions. Lastly *automatically-defined functions* and *automatically-defined macros* encourage modularity in the evolution of genetic programs.

**Strongly-Typed Genetic Programming (STGP)** Strongly-typed genetic programming adds type constraints to the return values and child arguments of nodes. In STGP, a subtree rooted at node $N$ may serve as a subexpression in argument position $x$ to node $M$ only if the return type for $N$ is compatible with the type for argument $x$ in $M$. For example, if the node inner-product has a return type constraint of FLOAT (this is just a symbol, and is not necessarily a data type) and the function (scalar-multiply *float matrix*) expects objects of return type FLOAT to fill its first argument and objects of return type MATRIX to fill its second argument, then (scalar-multiply (inner-product ...) ...) is valid but not (scalar-multiply ... (inner-product ...)). Similarly, trees have a specific type: a node $N$ may serve as the root of the tree only if its return type is compatible with the tree's type. These type constraints are provided by the user to restrict which kinds of nodes may serve as children of which other nodes, and in what position. Tree generation, crossover, and mutation must conform to

Function Sets

| Result-Producing Branch | ADF0 Branch | ADF1 Branch | ADF2 Branch |
|---|---|---|---|
| + | + | + | + |
| − | − | − | − |
| x | x | x | x |
| y | y | y | y |
| adf0 | adf2 | arg0 | |
| adf1 | arg0 | | |
| | arg1 | | |

Example Generated Individual

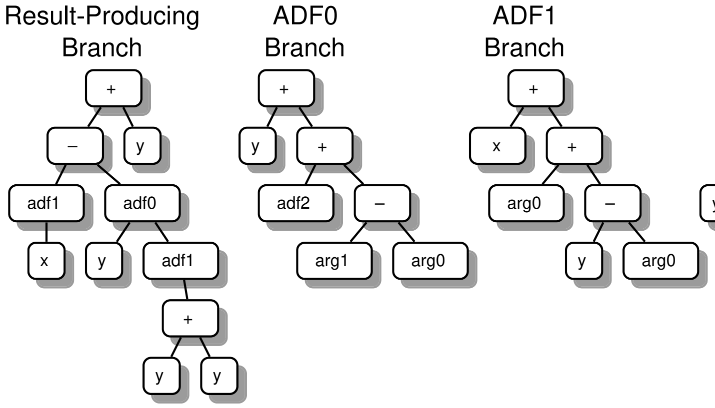



Figure 3.4: Automatically Defined Functions Example

these constraints. If crossover after some number of tries cannot find type-compatible subtrees to swap, then it gives up and simply reproduces the parents.

STGP comes in several different flavors. The basic form of STGP uses constant symbols for types, and type compatibility is determined by symbol comparison, and there are a finite number of symbols to choose from. [Montana 1995] also describes STGP with generic functions and data. This allows, for example, the creation of a generic if-then statement (if *boolean generic generic*) which returns any arbitrary type, and allows any arbitrary type to fill the two *generic* argument positions, as long as the return type and the two argument types are all the same. [Haynes, Schoenefeld, and Wainwright 1996] further extended this notion to cover hierarchical types.

Performing these advanced forms of STGP requires the ability to do type unification throughout a tree every time that a subtree is modified. This can be an expensive process. On the other hand, the cost of basic STGP is $O(1)$, but is not very flexible. One compromise is what I call *set-based* STGP. Here, types are sets of symbols, and types are compatible if their intersection is nonempty. There are only a finite number of symbols to choose from. Set-based STGP can be used for many of the tasks for which the more advanced forms of STGP would normally be required, but the cost is low, at worst $O(s)$ where $s$ is the number of symbols.

**Ephemeral Random Constants (ERCs)** Some problem domains require that some terminals in the tree take the form of numeric constants. GP does this by creating *ephemeral random constants* or ERCs. ERCs are added to trees by including an "ERC template" terminal function in the function set. When the tree generation algorithm copies a node from this template, the copied node is automatically assigned a random numeric value from a user-specified numeric range. Once a node has been set to this numeric value, its value becomes immutable.

**Automatically Defined Functions and Macros (ADFs and ADMs)** Automatically Defined Functions (ADFs) [Koza 1994] are an approach to modularizing genetic programming individuals. If a problem calls for ADFs, individuals take the form of forests of trees. Each tree is known as a *branch*, and these branches are equivalent to Lisp defun or defmacro statements. One tree is called the *result-producing branch*, and an individual is assessed by executing this tree. All the other trees are called *ADF branches*, and are "function-called" from the result-producing branch or recursively from other ADF branches. ADF branches are labeled ADF0, ADF1, ADF2, and so on.

Each branch may have its own unique function set if it is appropriate to the problem. To these function sets are added two special kinds of function nodes: adf nodes and arg (argument) nodes. An adf node in a function set represents a function call to a corresponding ADF branch. Each adf node is named for the particular ADF branch it represents (adf0 represents ADF0, for example). The arity of an adf node is determined by the number of "arguments" to its ADF branch, which is in turn defined by the number of arg nodes in that ADF branch's function set.

An arg node in a given ADF branch's function set is a terminal which represents an argument passed to its ADF branch when the ADF branch is called. Each arg node has a different symbolic name (arg0, arg1, etc. are common), and each is associated with a different child of the adf node which called the arg node's enclosing ADF branch. Since the result-producing branch is initially called without arguments, arg nodes may not appear in its function set.

When an adf node is encountered, its children are first evaluated, then each of their return values is assigned to the appropriate arg node in the corresponding ADF branch. Then that ADF branch is evaluated, with the arg nodes returning their assigned values. The adf node then returns the returned value of the evaluated ADF branch. The total number of ADF branches, and which trees may call them, is up to the experimenter. ADFs may call other ADFs, though recursion is rarely permitted.

As an example, imagine if the user has specified three ADFs, ADF0, ADF1, and ADF2. ADF0 and ADF1 may be called by the result-producing branch, and ADF0 may call ADF2. ADF0 takes two arguments, ADF1 takes one argument, and ADF2 takes no arguments. In addition, the function sets of all four trees include the terminals x and y, as well as the two-argument nonterminal functions + and − . Figure 3.4 shows the function sets for each branch in this example, followed by a feasibly-created individual under these constraints. This individual is equivalent to the following Lisp code:

```
(defun adf2 () (+ (+ y x) x))

(defun adf1 (arg0) (+ x (+ arg0 (− y arg0))))

(defun adf0 (arg0 arg1) (+ y (+ (adf2) (− arg1 arg0))))

;;; the result
(+ (− (adf1 x) (adf0 y (adf1 (+ y y)))) y)
```

An *automatically defined macro* (ADM) [Spector 1996] is identical to an automatically-defined function except that children to an adm node are lazily evaluated only (and each time) the corresponding arg node is called. The evaluated value of the child is then returned by the arg node. It is possible, then, that a particular child of an adm node may be evaluated repeatedly (and return different values), or not at all, during the ADF branch's execution. This differs from an adf node, which evaluates all of its children exactly once, and stores their values away prior to executing the ADF branch. ADMs are thus similar to Lisp macro control structures like loops and if/then statements. Just like compiled Lisp macros, ADMs may not be called recursively. If in the previous example all the adf nodes were instead adm nodes, then the individual would be equivalent to the following Lisp code:

| Function Syntax | Arity | Description |
|---|---|---|
| (+ *i j*) | 2 | Returns $i + j$ |
| (− *i j*) | 2 | Returns $i - j$ |
| (∗ *i j*) | 2 | Returns $ij$ |
| (% *i j*) | 2 | If $j$ is 0, returns 1, else returns $\frac{i}{j}$ |
| (sin *i*) | 1 | Returns sin($i$) |
| (cos *i*) | 1 | Returns cos($i$) |
| (exp *i*) | 1 | Returns $e^i$ |
| (rlog *i*) | 1 | If $j$ is 0, returns 0, else returns $\log(\|i\|)$ |
| x | 0 | Returns the value of the independent variable ($x$). |
| *ERCs* | 0 | (*Optional* ) Ephemeral random constants chosen from floating-point values from -1 to 1 inclusive. |

Table 3.1: Genetic Programming Function Set for the Symbolic Regression Domain

```
(defmacro adm2 () ‘(+ (+ y x) x))

(defmacro adm1 (arg0) ‘(+ x (+ ,arg0 (− y ,arg0))))

(defmacro adm0 (arg0 arg1) ‘(+ y (+ (adm2) (− ,arg1 ,arg0))))

;;; the result
(+ (− (adm1 x) (adm0 y (adm1 (+ y y)))) y)
```

Because the arguments to ADMs are evaluated lazily and repeatedly, whereas ADF arguments are evaluated only once prior to the ADF call, the functional semantics of ADMs and ADFs differ only when the arguments are not referentially transparent, that is, when calling the arguments causes side effects with an explicit order. This often happens either in domains which modify the state of the world during evaluation (as in the Artificial Ant domain, discussed later in Section 3.4.3), or which contain internal memory (as in the *indexed memory* technique discussed later in Section 4.2).

## 3.4 Example Problems for Genetic Programming

This section introduces six standard benchmark problem domains solved by GP, which are used later in various experiments in the thesis. Though several originate from earlier machine learning literature, [Koza 1992] has established the standard implementation of these problems for genetic programming, and this thesis follows those examples.

The problems introduced are: Symbolic Regression (evolving a mathematical equation from transcendental functions), 6- and 11-Bit Boolean Multiplexer and Parity (evolving boolean functions), and Artificial Ant and Lawnmower (evolving simple path planning operations).

### 3.4.1 Symbolic Regression

Symbolic Regression is the canonical example domain for genetic programming. The object of symbolic regression is to find a symbolic function which best fits a set of data points of the form $\langle x, y\rangle$ in the real cartesian plane. Symbolic Regression differs from classic regression methods in statistics in that the function set includes transcendental functions (sine, cosine, natural logarithm).

| Function Syntax | Arity | Description |
|---|---|---|
| (and *i j*) | 2 | Returns $i \cap j$ |
| (or *i j*) | 2 | Returns $i \cup j$ |
| (not *i*) | 1 | Returns $\neg i$ |
| (if *test then else*) | 3 | If *test* is true, then *then* is returned, else *else* is returned. |
| a0, a1 | 0 each | Return the values of variables A0 and A1 respectively. |
| a2 | 0 | (*11-Multiplexer Only*) Returns the value of variable A2. |
| d0, d1, d2, d3 | 0 each | Return the values of variables D0, D1, D2, and D3 respectively. |
| d4, d5, d6, d7 | 0 each | (*11-Multiplexer Only*) Return the values of variables D4, D5, D6, and D7 respectively. |

Table 3.2: Genetic Programming Function Set for the 6-Bit and 11-Bit Multiplexer Domains

| Function Syntax | Arity | Description |
|---|---|---|
| (and *i j*) | 2 | Returns $i \cap j$ |
| (or *i j*) | 2 | Returns $i \cup j$ |
| (nand *i j*) | 2 | Returns $\neg(i \cap j)$ |
| (nor *i j*) | 2 | Returns $\neg(i \cup j)$ |
| d0, d1, d2, ... | 0 each | Return the values of variables D0, D1, D2, ... respectively. The number of d*x* nodes in the function set is the number of bits in the particular Parity problem being run. |

Table 3.3: Genetic Programming Function Set for the Parity Domain

The benchmark problem for Symbolic Regression has 20 random data points to fit. The $x$ values of the points are picked at random; these form the "independent variables" in the data set. Their corresponding $y$ values are computed from the benchmark function $y = x^4 + x^3 + x^2 + x$. GP individuals will try to fit to this function. Table 3.1 shows Symbolic Regression's function set. In some versions of Symbolic Regression, ephemeral random constants are added to the function set.

A Symbolic Regression individual consists of a single tree. AssessFitness($p_i$) works as follows. For each data point $\langle x_i, y_i \rangle$, the independent variable (the value that the terminal x will return) is set to $x_i$. The individual's tree is then evaluated and stored in $r_i$. The standardized fitness is $\sum_{i=1}^{n} |(y_i - r_i)|$. A standardized fitness of 0 represents a perfect match. An example ideal individual is:

**Result:** (+ (* x (* (+ x (* x x)) x)) (* (+ x (cos (- x x))) x)).

### 3.4.2 Boolean Domains: 6- and 11-Bit Multiplexer, and Parity

The objective of the 6-Bit and 11-Bit Multiplexer problems is to find a boolean function which performs multiplexing over a 2-bit or 3-bit address. In 6-Bit Multiplexer, there are two boolean-valued address variables (A0 and A1) and four corresponding boolean-valued data variables (D0, D1, D2, D3). The 6-Bit Multiplexer function must return the value of the data variable at the address described by the binary values of A0 and A1. For example, if A1 is true and A0 is false, the address is 2 (binary 10), and in this case, the optimal individual must return the value stored in D2. Since there are six boolean variables altogether, there are 64 permutations of these variables and hence 64 test cases.

11-Bit Multiplexer is similar, except that it has three address variables (A0, A1, A2), and thus has eight data variables (D0, D1, D2, D3, D4, D5, D6, D7). In this case, there eleven variables altogether, and so there are 2048 test cases. Table 3.2 shows the 6-Bit and 11-Bit Multiplexer function sets.

| Function Syntax | Arity | Description |
|---|---|---|
| (progn3 *a b c*) | 3 | *a*, *b*, then *c* are executed. |
| (progn2 *a b*) | 2 | *a*, then *b* are executed. |
| (if-food-ahead *then else*) | 2 | If food is immediately in front of the ant, *then* is executed, else *else* is executed. |
| move | 0 | Moves the ant forward one square, eating food if it is there. |
| left | 0 | Rotates the ant ninety degrees to the left. |
| right | 0 | Rotates the ant ninety degrees to the right. |

Table 3.4: Genetic Programming Function Set for the Artificial Ant Domain

A Multiplexer individual consists of a single tree. AssessFitness($p_i$) works as follows. For each test case, the data and address variables are set to return that test case's permutation of boolean values, and the individual's tree is then evaluated. The standardized fitness is the number of test cases for which the individual returned the wrong value for the data variable expected, given the current setting of the address variables.

An example of an ideal 6-Multiplexer individual is:

**Result:** (if a1 (not (not (if a0 d3 d2))) (if (not (and (if a0 a1 d0) (and d1 d2))) (if (not (if a1 a0 d1)) (if a0 a1 d0) (or (or a1 d0) (or (if d2 d0 d2) (and d1 a0)))) (if (or d1 a1) (if (not (if a1 d3 (if d3 d1 d1))) (and (and d2 a1) (not a0)) (or (or a1 d0) (or (if d2 d0 d2) (and d1 a0)))) (and (not d0) (and a0 d1)))))

Although this looks like a complex solution, in fact 6-Multiplexer is a fairly easy problem for genetic programming to solve.

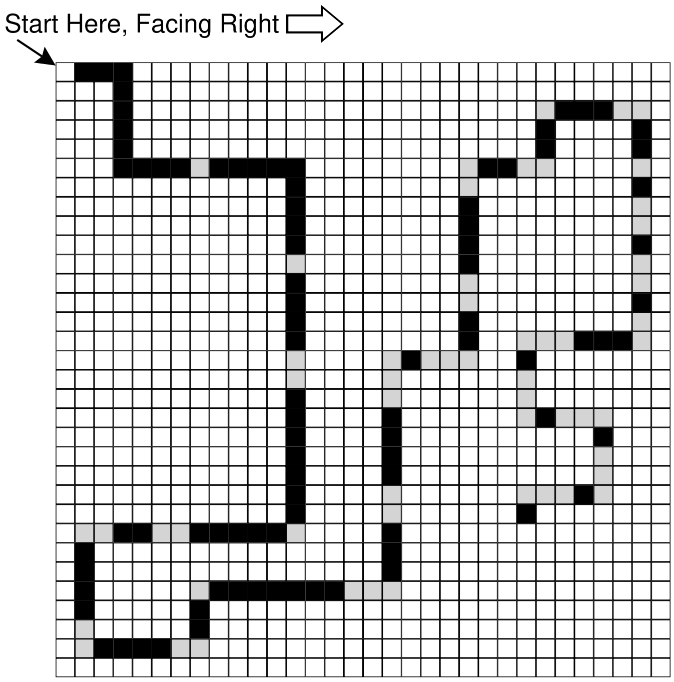
The Parity problem is a similar boolean problem over some $n$ number of data variables. In the Parity problem, the objective is to return true if, for the current boolean settings of these variables, there is an even (or odd) number of variables whose value is true. For the N-Parity problem, there are then $2^N$ test cases. AssessFitness($p_i$) works essentially the same as for the Multiplexer problems. Table 3.3 shows the Parity function set.

The dynamics of the boolean problem domains differ from Symbolic Regression in important ways. First, the boolean problems have a discrete solution space, and so have only a finite number of possible fitness values. In contrast, Symbolic Regression forms functions of continuous variables, and so can have fitness values anywhere in the real range from 0 to infinity. One important consequence of this, discussed in Section 9.2.1, is that it is common for crossover or mutation to cause boolean individuals to radically change in structure and function, but not change in fitness at all. This rarely happens in a continuous domain like Symbolic Regression.

### 3.4.3 Artificial Ant

Artificial Ant is a relatively difficult problem for genetic programming. The Artificial Ant problem attempts to find a simple robotic ant algorithm which will find and eat the most food pellets within 400 time steps. The ant may move forward, turn left, and turn right. If when moving forward it chances across a pellet, it eats it. The ant can also sense if there is a pellet in the square directly in front of it. The grid world in which the Artificial Ant lives is shown in Figure 3.5. The pellet trail shown is known as the "Santa Fe Trail". The world is toroidal: walking off an edge moves the ant to the opposite edge. The Artificial Ant function set is shown in Table 3.4.

An Artificial Ant individual consists of a single tree. AssessFitness($p_i$) works as follows. The ant starts out on the trail in its initial position and orientation. The tree is executed: as each sensory or movement node is executed, the Ant senses or moves as told. When the tree has completed execution, it is re-executed again and again. Each movement counts as one time step. Assessment finishes when the Ant has eaten all the pellets in the world or when the 400 time steps have expired. The Ant's standardized fitness is the number of pellets it did not eat at the end of the trial. A (highly parsimonious) example of an optimal Artificial Ant solution is:




**Legend.** Squares are black where there are food pellets, and white or grey elsewhere.

Figure 3.5: “Santa Fe Trail” Used in the Artificial Ant Domain

**Result:** (progn3 (if-food-ahead move (progn2 left (progn2 (progn3 right right right) (if-food-ahead move right)))) move right)

The Artificial Ant domain is different from the Symbolic Regression and the boolean domains in that the return value of each tree node is ignored. The only thing that matters is each node’s actions in the world, that is, each node’s side effect: moving the ant, turning it, etc. This means that in Artificial Ant, the order in which the nodes are executed determines the operation of the individual. In Symbolic Regression and the boolean domains, it really doesn’t matter in what order subtrees are evaluated, or if both the *then* and *else* branches to an if statement are evaluated but only one is used, because the return values of these branches is their only contribution to the individual. But in Artificial Ant, evaluating a branch means potentially changing the world state. This makes it very easy for tiny modifications to the individual to randomize its effect on the world. Like the boolean domains, Artificial Ant is a “discrete” domain.

### 3.4.4 Lawnmower

In the Lawnmower domain, the individual directs a lawnmower to mow a toroidal grid lawn, much as the Artificial Ant domain directs an ant to move about its toroidal grid world. In the Lawnmower domain, an individual may turn left, mow forwards, or “hop” some $\langle x, y\rangle$ units away. Lawnmower has no sensor information: it must be hard-coded to mow the lawn blind. The standard lawn size is 8 by 8.

[Koza 1994] used this domain originally to demonstrate the advantages of automatically defined functions (ADFs). Lawnmower is difficult without ADFs but fairly trivial when using ADFs. When not using ADFs, a Lawnmower individual consists of a single tree using the function set shown in Table 3.5. When using ADFs, a Lawnmower individual consists of three trees: a result-producing branch, an ADF0 branch, and an ADF1 branch. In addition to using the functions in the basic function set in Table 3.5, each branch also uses additional ADF-specific functions as outlined in Table 3.6.

| Function Syntax | Arity | Description |
|---|---|---|
| (progn2 *a b*) | 2 | *a*, then *b* are executed. Returns the return value of *b*. |
| (v8a *i j*) | 2 | Evaluates *i* and *j*, adds the vectors they return, modulo 8, and returns the result. |
| (frog *i*) | 1 | Evaluates *i*. Let $\langle x, y\rangle$ be *i*'s return value. Then frog moves $\langle x, y\rangle$ squares relative to its present rotation, where the positive $X$ axis points in the present "forward" direction of the lawnmower, and the positive $Y$ axis points in the present "heading left" direction of the lawnmower. Returns $\langle x, y\rangle$. |
| mow | 0 | Moves the lawnmower forward one square, mowing that square of lawn if it is not already mown. Returns $\langle 0, 0\rangle$. |
| left | 0 | Rotates the lawnmower ninety degrees to the left. Returns $\langle 0, 0\rangle$. |
| *ERCs* | 0 | Ephemeral random constants of the form $\langle x, y\rangle$, where $x$ is an integer chosen from the range $(0, ..., x_{max} - 1)$ and $y$ is an integer chosen from the range $(0, ..., y_{max} - 1)$, where $x_{max}$ and $y_{max}$ are the width and height of the lawn in squares, respectively. |

Table 3.5: Genetic Programming Function Set for the Basic Lawnmower Domain

| Function Syntax | Arity | Description |
|---|---|---|
| | | Result-Producing Branch |
| (adf0 *arg0*) | 1 | Automatically defined function which calls the ADF0 branch. |
| adf1 | 0 | Automatically defined function which calls the ADF1 branch. |
| | | ADF0 Branch |
| adf1 | 0 | Automatically defined function which calls the ADF1 branch. |
| arg0 | 0 | The value of argument *arg0* passed when the ADF0 branch is called. |
| | | ADF1 Branch |
| ADF1 does *not* have (frog *i*). | | |

Table 3.6: Genetic Programming Function Set Extensions for the Lawnmower Domain with Two Automatically-defined Functions

AssessFitness($p_i$) is simple. The lawnmower is placed somewhere on the lawn, and the individual's tree is executed once. Each mow and frog command moves the lawnmower and mows the lawn in its new location. Once the tree has been executed, the standardized fitness for the individual is the number of squares of lawn not yet mown. An example optimal individual (using ADFs) is:

**Result:** (progn2 (progn2 (adf0 (progn2 (adf0 left) (v8a ⟨7,0⟩ ⟨0,4⟩))) (progn2 left ⟨3,4⟩)) (v8a (progn2 (adf0 (v8a left left)) (progn2 (frog mow) (adf0 adf1))) (adf0 (progn2 (v8a ⟨6,7⟩ adf1) (progn2 ⟨1,1⟩ mow)))))

**ADF0:** (v8a (v8a (v8a (progn2 (v8a adf1 mow) (v8a adf1 mow)) (frog (v8a mow arg0))) (v8a (v8a (frog arg0) (progn2 ⟨1,4⟩ ⟨2,6⟩)) (progn2 (v8a ⟨1,5⟩ adf1) (frog mow)))) (v8a (v8a (v8a (progn2 adf1 adf1) (v8a adf1 mow)) (v8a (progn2 arg0 adf1) (frog left))) (frog (v8a (v8a arg0 left) (v8a ⟨7,0⟩ mow)))))

**ADF1:** (progn2 (v8a (progn2 (v8a (v8a mow mow) (v8a mow ⟨5,1⟩)) (v8a (v8a mow left) (progn2 left mow))) (v8a (progn2 (v8a mow mow) (progn2 ⟨1,3⟩ ⟨2,1⟩)) (v8a (progn2 ⟨3,6⟩ mow) (progn2 left ⟨3,4⟩)))) (v8a (progn2 (progn2 (v8a mow left) (progn2 ⟨6,6⟩ ⟨1,4⟩)) (progn2 (v8a mow left) (v8a mow ⟨7,7⟩))) (progn2 (v8a (progn2 left left) (v8a mow left)) (v8a (progn2 left ⟨2,1⟩) (v8a ⟨1,7⟩ mow)))))

Although this individual looks imposing, in fact with ADFs Lawnmower is fairly easy for genetic programming to solve. Much of this individual is junk. The reason ADFs work so much better in this domain is simple and unfair: a Lawnmower individual is executed only once, and has no iteration or recursion, and so within its tree must exist enough commands to move lawnmower to every spot of lawn. To do this for a single tree demands a big tree. But with ADF branches, the result-producing branch can repeatedly call ADFs (and ADF0 can repeatedly call ADF1), so the total size of the individual can be much smaller and still take advantage of many more total moves.

Lawnmower, like Artificial Ant and Multiplexer, is a "discrete" domain. Also, Lawnmower operates via side-effects like Artificial Ant does, and so an individual's node execution order is important. However, Lawnmower has no if statement and no sensory information, so all nodes are executed exactly once. Further, unlike Artificial Ant, Lawnmower nodes' return values are important.

# Chapter 4

# Issues in Evolutionary Computation and Genetic Programming

This chapter reviews literature in evolutionary computation with variable-length representations, additional GP variants, examples of GP's scalability, and the present debate over tree bloat. Parts of this literature review discuss graph structures. To avoid confusion, I will always use "vertex" to describe the nodes in graphs and "node" to describe the nodes in GP trees.

## 4.1 Variable-Length Representations

Genetic programming represents the lion's share of variable-length representation literature, but variable-length genomes have been in use in EC since its very inception. [Fogel, Owens, and Walsh 1966] developed the field of evolutionary programming to search for arbitrary-sized finite-state automata, applying operators to add and delete edges and vertices. A recent application of EP is GNARL, which evolves neural networks for a variety of tasks [Angeline, Saunders, and Pollack 1994] such as boolean language induction and pattern recognition. Classifier systems also use variable-length representations. [Smith 1980] evolved simple rule-based programs which learned problems such as maze navigation and poker betting. This work eventually led to the "Pitt Approach" to learning classifier systems [DeJong 1989], in which each individual in the population is a classification rule set of arbitrary length.

Besides the tree-style genetic programming discussed in Chapter 3, there is a substantial body of work using other variable-length genome constructions to evolve computer programs. One approach represents the program as a list of machine instructions (for example, [Stoffel and Spector 1996; Nordin 1997; Banzhaf *et al.* 1998]). [Teller 1998] instead evolves programs as cyclic call-graphs. Some work has also been done representing programs as directed acyclic graphs of functions [Keijzer 1996; Handley 1994].

Other interest in arbitrary-length genomes has sprung from attempts to improve on GA's fixed-length vector genotypes while still evolving for a fixed number of parameter settings. Best known in this area is the *messy genetic algorithm* [Goldberg *et al.* 1993], which evolved arbitrary lists of $\langle parameter, value \rangle$ pairs. This work was later extended to the *gene expression messy genetic algorithm* (GEMGA) [Kargupta 1996]. Some theoretical work has also treated genomes as unordered sets of arbitrary objects [Radcliffe and George 1993].

Another area of interest has been in applying a truly "DNA-like" genome to GA problems. Genomes in such approaches have typically been long strands of DNA-like codons, often even using a four-letter A,T,G,C alphabet. Although the genomes are formally fixed in size, they are usually very long, and genes are randomly dispersed throughout the genome, delimited by start and stop codon sequences. This means that the number of genes expressed in the genome can be of any size, and genes can be any length (up to a point). [Fullmer and Miikkulainen 1991] used such a genome to evolve neural networks, calling it *marker-based encoding*. [Jakobi 1995] introduced a similar encoding to attack problems in evolutionary robotics. [Wu and Lindsay 1995] examined the dynamics of non-coding segments (parts of the strands that did not define genes) in these kinds of genomes. [Burke *et al.* 1998] continued this work, adding more ideas from biology, including multiple reading frames (allowing genes to overlap) and homologous crossover (recombination at points where codons are most similar). Unfortunately, most work in this area has been exploratory; few papers have compared such approaches empirically to other EC work in nontrivial domains. [Luke, Hamahashi, and Kitano 1999] borrowed similar ideas from cellular biology, but eschewed the notion of using DNA codon strings. Instead, we used a genome consisting of an arbitrary-sized set of genes, each tagged with a real-valued locus between 0 and 1 which placed them on the chromosome. Drawing from biological gene expression networks, these genes formed rules whose influence on each other depended on their similarity of fit in a "gene-expression space". The genes' rules were then mapped to state transitions in finite-state automata and the technique evolved FSAs to do language induction on a popular benchmark, with good results compared to the existing literature.

One fertile area for variable-length representations has been in attempts to evolve graphs and networks. Early attempts at evolving neural networks fixed the network topology and evolved the weights as values in the genome [Collins and Jefferson 1991], and usually were expressed with simple fixed-length vectors. But later approaches used variable representations to directly encode arbitrary numbers of graph vertices and edges in the genome [Fullmer and Miikkulainen 1991; Lindgren *et al.* 1992; Angeline, Saunders, and Pollack 1994]. Still later techniques attempted to use morphological rules to describe a neural network. [Kitano 1990] was the first publication to attempt to evolve graph structures using sets of rules for "building" a graph, rather than explicitly stating the vertices and edges. [Boers *et al.* 1993] also "grew" networks, using L-grammars operating on graph vertices and edges. Genetic programming techniques have also been used to evolve a variety of graph structures, including push-down automata [Zomorodian 1994] and graph-based programs [Teller 1996]. [Koza and Rice 1991] used GP to directly encode the edges and vertices of a neural network. The most oft-discussed technique in the literature is *cellular encoding* [Gruau 1992], discussed below.

## 4.2 Variations on Genetic Programming

**Cellular Encoding** Cellular encoding [Gruau 1992] is a genetic programming variant where the GP tree is a program which builds a graph, often for use as a neural network. Cellular encoding does this through repeated operations on an embryonic graph vertex, "growing" additional vertices and edges out of the embryonic vertex in a process reminiscent of cellular mitosis. Cellular encoding tree programs are structurally identical to those in genetic programming, but their order of execution is breadth-first, whereas GP is depth-first.

A cellular encoding node is passed one graph vertex, which it modifies in some way. The modified vertex is then passed to the first child of the node. If a node has more than one child, it must create new vertices in its vertex modification step, one vertex for each of its other children. Child nodes do not modify their respective vertices (and hand results to *their* children) until it is their turn to do so in the breadth-first execution order. A node may also delete its vertex from the graph. Modification may also create or delete edges and change edge and vertex features. A terminal node signifies the end of modification to a given graph vertex; it becomes a permanent fixture in the graph. Execution begins by adding the "embryonic" vertex to the graph, and passing that vertex to the root node. Because of its unusual breadth-first execution order, cellular encoding cannot be easily made to work with automatically defined functions or macros.

Cellular encoding is particularly interesting for several reasons. First, it can describe all possible neural network topologies —- in fact, [Gruau 1992] presents a compiler which transforms Pascal functions (with a few limitations) into neural networks which compute the same functions. Second, cellular encoding contains special nodes which attempt to reuse subtrees in the tree in a recursive fashion, thereby generating very large networks from reasonably small genomes. Lastly, cellular encoding takes advantage of GP-like tree genotypes to permit network phenotypes of any size, and to provide a straightforward mechanism for crossover between networks of widely differing topology.

Similar techniques have been used to evolve electrical circuits from a "circuit embryo". [Koza *et al.* 1997a] developed a time-optimal fly-to controller circuit in this way. [Jones and Joines 1999] used a related technique to evolve antenna designs. Though they are inspired by cellular encoding, these techniques have more in common with its cousin *edge encoding*, described in Chapter 6.

**Multiagent Coordination** Genetic programming is usually applied to multiagent coordination in one of two ways. First, individuals from subpopulations might work together on a common problem: [Andre 1995] used this approach to evolve communication between agents with different skills. Second, an individual might represent a whole cooperative team at a time, an approach popularized by [Haynes *et al.* 1995], who used GP to evolve cooperation in a predator-prey environment. [Raik and Durnota 1994] used a similar approach to evolve sporting strategies. [Luke and Spector 1996] used a predator-prey environment to consider two different team-formation styles: *homogeneous* teams, where all members of the team use the same GP program tree; and *heterogeneous* teams, where an individual consists of a forest of program trees, and each member of the team uses a different tree as his program. [Iba 1996a] used three populations to evolve two-agent cooperative behavior in the TileWorld domain. In the first population, individuals followed a homogeneous approach. The other two populations evolved agents which cooperated to solve the problem. Highly fit homogeneous agents were permitted to migrate to the other two populations.

**Evolving State** [Teller 1994b] introduced the notion of *indexed memory* to add a global memory state to genetic programming. Indexed memory adds two functions to a function set: (read *index*) and (write *index value*). read returns the current value of a slot in a global array accessible by the individual. write sets a slot to some value. Before an individual begins fitness assessment, the global memory is reset to zeroes.

[Teller 1994b] showed that by adding indexed memory and automatically-defined functions to GP, its tree description language becomes turing complete. This does not mean that GP can evolve any partial recursive function, however. This is because in order to evaluate the fitness of such a tree, one would have to be able to determine if its program halts (which is impossible), otherwise fitness evaluation conceivably goes on forever. Instead, one must halt the program after some $N$ maximal execution steps.

[Spector and Luke 1996] built on indexed memory to experiment with the cultural transmission of information between individuals in a population. This was done by not reinitializing the global memory array at fitness assessment time; thus individuals could leave information in the array to presumably help other members of the population, and even individuals in future generations.

## 4.3 Genetic Programming Applied to Difficult Problems

Genetic programming has of late been scaled to increasingly difficult problem domains. Besides the examples given in Chapters 5 and 8, the literature has a great many other examples. I give only a sampling here.

**Quantum Computing** [Spector *et al.* 1999] used genetic programming to evolve quantum algorithms. Quantum computation takes advantage of theoretical quantum physics to build computing devices which, depending on the algorithm, have a measurably better computational complexity than the best possible classical algorithms. Data in quantum computers is not stored with bits but with *qubits*, basis vectors representing possible quantum states of the data. Quantum algorithms often take the form of feedforward networks of "quantum gates", special matrices which operate on these qubits. Quantum computing is very complicated, but [Spector *et al.* 1999] was able to apply GP to find a number of interesting quantum algorithms. This included rediscovering known quantum algorithms (Deutsch's Early Promise Problem and Grover's Database Search Problem), finding a quantum equivalent to the classical Scaling Majority-On Problem, and discovering quantum results which an expert had thought could not exist (in the And-Or Query Problem).

**Chemical Structures** [Nachbar 2000] evolved chemical structures with GP, encorporating the QSAR chemistry model (Quantitative Structure Activity Relationship) as part of the fitness function. This technique encoded both cyclic and acyclic chemical structures in GP in a way that preserved proper valences among chemical bonds. The result was able to optimize for a variety of problems in organic and inorganic chemistry.

**Spacecraft Attitude Control** [Howley 1996] evolved near-minimum-time spacecraft attitude maneuvers which outperformed hand-coded attempts at the problem, and approached the provably minimum error possible. The GP technique, with a function set similar to Symbolic Regression annotated with ADFs and if-then statements, was able to produce control laws for two classes of spacecraft attitude maneuvers: rest-to-rest and RLNZ (rate-limited non-zero terminal velocity).

**Cellular Automata Rules** The Majority Classification Problem is an unsolved problem in cellular automata. The objective is to find a one-dimensional cellular automata ruleset which, when initialized in some arbitrary state, comes to rest with all 1's or all 0's, depending on whether or not the initial state had more 1's or more 0's to start with. The best hand-coded solution in the past twenty years has achieved an 82.178% accuracy. The best computer-learned rule to date (found through GA) achieved a 76.9% accuracy. [Andre, Bennett III, and Koza 1996] applied GP to the problem and were able to produce an evolved rule which achieved an 82.326% accuracy. Interestingly, this rule was radically different in form from the best hand-written rules, suggesting a new way to approach the problem in the future.

**Electrical Circuits** This section cannot be complete without a mention again of John Koza's considerable work in evolving human-competitive analog electrical circuits using a technique similar to cellular encoding. [Koza *et al.* 1997b] succeeded in rediscovering a complex sorting network of seven items using sixteen steps. [Koza *et al.* 2000] evolved electrical circuits containing free variables. [Koza *et al.* 1997c] evolved complex circuitry for performing squaring and cubing, square and cube roots, and the logarithm. The authors are not aware of any preexisting cube root circuit in the literature. This work necessitates very large population sizes (often 640,000), running for 500 generations, on large-scale Beowulf-class supercomputers with island models. See [Koza *et al.* 1999] for an extensive survey of this work.

## 4.4 The Race Against Bloat

Variable-length representations are susceptible to *bloat*, the uncontrolled growth in individuals over time, independent of equivalent changes in fitness. [Smith 1980] noted bloat as a serious impediment to evolving arbitrary rule sets for classifier systems. [Lanzi 1999a] discussed the problem of "overspecification" (increasing numbers of genes which code for the same thing) in evolving classifier systems using messy genetic algorithms, and suggested mutation procedures to deal with the issue. But genetic programming literature has documented the problem the most extensively, as it is a serious issue for both tree-based GP genomes (for example [Koza 1992; Blickle and Thiele 1994; McPhee and Miller 1995; Angeline 1998; Langdon *et al.* 1999; Lanzi 1999b]) and for linear GP genomes [Banzhaf *et al.* 1998].

There are two major approaches to dealing with GP tree bloat. First, by improving breeding, selection, and tree-generation, GP can be made to search more efficiently to find better individuals before bloat sets in. Second, various techniques can help GP stave off bloat as long as possible, lengthening the search interval.

### 4.4.1 Improving GP Breeding Operators

Devising the right breeding operator is critical to producing high-quality individuals before bloat takes over. Most of the genetic programming literature has chosen to use 95% subtree crossover and 5% direct reproduction, with no mutation of any kind. These settings are partly due to inertia: Koza's early experiments with the 6-Bit Multiplexer problem supported his argument for heavy use of crossover ([Koza 1992], pp. 599–600), and most later GP work has followed closely in the Koza tradition. But the popularity of crossover may also be due to a latent belief that GP crossover must somehow trade "things of value" from individual to individual.

This notion of "things of value" is borrowed directly from genetic algorithms, from which the GP community emerged. GA emphasizes crossover, using only a tiny bit of mutation if any, citing the GA schema theorem for support. But as the GA and EP communities have converged, new evidence and theory has cast some doubt on the building-block hypothesis and suggested that crossover may not be as useful for GA-style vector chromosomes as previously thought (see for example [Shaffer and Eshelman 1991; Tate and Smith 1993; Hinterding, Gielewski, and Peachey 1995]). Still others have produced new arguments in favor of crossover, for example, [Spears 1993].

Genetic programming's tree-based genome is so distant from the genetic algorithm vector genome that it is very difficult to form a similar theoretic justification for favoring crossover over mutation. Even so, GP literature freely uses, though with little theoretical support, the overall building-block and schemata concepts used in GA (for example, [Iba and de Garis 1996; Rosca and Ballard 1996; Soule, Foster, and Dickinson 1996b]). And GP researchers have made some initial stabs at a GP building block hypothesis ([Haynes 1997; Poli and Langdon 1997; Rosca 1997; Poli 2000]). However, recent studies have countered the notion of building blocks in GP [O'Reilly and Oppacher 1995; Angeline 1997] and shown that mutation can outperform crossover [Angeline 1998; Chellapilla 1997]. [Angeline 1998] argues that subtree crossover is a significant source of tree bloat, while [Langdon and Poli 1997c] instead indicts subtree mutation.

### 4.4.2 Fighting Bloat

The most common method for controlling code bloat are the three ad-hoc techniques used in [Koza 1992]:

- The GROW and FULL algorithms are depth-limited to between 2 and 6 for the generation of initial individuals.
- The GROW algorithm is depth-limited to 4 for the generation of subtrees for subtree mutation, and mutation points are picked from nonterminals 90% of the time.
- Subtree mutation and subtree crossover are both restricted to producing individuals of depth 17 or less.

The last technique listed have been shown to have unforeseen effects when most trees in the population reach the 17-depth limit [Gathercole and Ross 1996]. Chapter 8 reveals that without this depth limiting, common function sets will cause individuals' expected tree sizes to approach infinity. Even with the depth limiting in place, subtree mutation can still have bloating characteristics, albeit not to such a dramatic effect. And with the depth limiting in place, near-full trees at the maximum alloted depth will dominate the initial trees and subtrees generated, biasing the search space to a very limited set of tree structures.

The second most common method for controlling bloat is *parsimony pressure*: adding a tree size penalty as an additional criterion in the individual's fitness assessment. This multiobjective-selection idea is not originally from genetic programming; [Smith 1980] proposed it to deal with bloat when evolving sets of rules. While some approaches use a linear or constant function for parsimony pressure (for example [Soule, Foster, and Dickinson 1996a]), others add parsimony pressure adaptively in response to tree growth metrics such as the amount of introns [Iba, de Garis, and Sato 1994; Blickle 1996].

Another method is *code editing*: physically deleting introns in programs. [Soule, Foster, and Dickinson 1996a] and [Blickle 1996] report strong results with this approach. [Iba and Terao 2000] argues that programs edited for introns specific to the fitness cases presented can still produce solutions general to all possible test cases. However, [Haynes 1998] warns that editing can lead to premature convergence in the evolutionary system. Still another method is *explicitly defined introns* [Nordin, Francone, and Banzhaf 1996; Smith and Harries 1998; Blickle 1996]. The idea here is to allow the inclusion of special nodes which increase the likelihood of crossover at specific positions in the tree. A related approach is examined in [Angeline 1996].

Lastly some researchers (notably [O'Reilly 1995]) have suggested a form of hillclimbing: rejecting crossover results which do not produce a child better than its parent. When crossover is rejected, the parent is replicated into the new population instead of its failed child. As discussed later, this technique can decrease bloating significantly.

### 4.4.3 What Causes Bloat?

There are four major theories of bloat, only a summary of which is given here. Extensive literature surrounding these theories is instead found in Chapter 9, because the arguments made in that chapter are directly tied to specific variations of these theories in the literature.

- *Hitchhiking* [Tackett 1994] argues that many genetic programming trees contain subtrees which perform no purpose. These subtrees are known as *introns*. As "important" pieces of code (building blocks) are spread from individual to individual, introns attached to this code come along for the ride, making individuals grow larger and larger.

- *Defense against crossover* (from [Blickle and Thiele 1994; Nordin and Banzhaf 1995] and many others) argues that introns directly act to increase the number of points in a tree for which crossover is unlikely to modify the individual. Thus the more introns an individual has, the more likely it is to survive crossover.

- *Removal bias* [Soule and Foster 1998] focuses on a special kind of intron, *inviable code*, which defines an area in an individual's tree where no crossover can possibly change an individual's function or fitness. Just as in the defense against crossover theory, removal bias says that the more inviable code there is in a tree, the more likely the tree is to survive crossover.

- *Diffusion* [Langdon and Poli 1997b] argues that trees grow because genetic programming starts with small trees, and during evolution these trees diffuse from a search region with few good solutions (small trees) to one with many more good solutions (large trees).

## Chapter 5

# Evolved Robot Soccer

This chapter introduces a nontrivial problem domain for genetic programming, illustrating what can be done with the technique and hilighting the problem of tree bloat. The problem domain evolves cooperative robot team strategies for playing soccer in the RoboCup Soccer Server.

RoboCup is a competition which pits teams of robots against each other in a robotic soccer tournament [Kitano *et al.* 1995]. To be successful at RoboCup, a team of robotic soccer players must be able to cooperate in real time in a noisy, highly-dynamic environment against an opposing team. In addition to the two "real-robot" leagues at RoboCup, there is a softbot league which competes inside a provided soccer simulator, the RoboCup Soccer Server [Itsuki 1995]. The simulator enforces very limited and noisy sensor information, complex physics, real-time dynamics, and limited intercommunication among softbots. The result is a rather challenging real-time domain.

Practically all entrants in the RoboCup simulator league used hand-coded team strategy algorithms; some fine-tuned a few low-level functions (like ball interception) with backpropagation or decision trees. In contrast, at the University of Maryland a group of undergraduates[1] and I entered a softbot team whose high-level strategies were entirely learned through genetic programming.

Unlike other teams, who had refined well-understood robotics techniques in order to win the competition, we saw the RoboCup simulator as a difficult environment to push the bounds of what was possible to do with existing evolutionary computation techniques. For a variety of reasons detailed later, the soccer simulator is *very* difficult to evolve programs for. Hence our goal was relatively modest: to produce a team which played at all. As it turned out, we were pleasantly surprised with the results. Our evolved teams learned to disperse throughout the field, pass and dribble, defend the goal, and coordinate with and defer to other teammates. At the IJCAI/RoboCup97 competition our team managed to win its first two matches against human-coded opponents, and took home the RoboCup97 Scientific Challenge award.

## 5.1 The RoboCup Soccer Server Domain

In a full match, the Soccer Server admits eleven separate player programs per team, each controlling a different virtual soccer player in its simulation model. By regulation rules, these player programs must be separate processes which are not permitted to communicate with each other except through the limited facilities provided by the Soccer Server. Each player on the team makes a separate UDP socket connection to the simulator. Once connected, a player program receives UDP datagrams once every $n$ milliseconds (commonly 300, depending on user-picked sensor options) providing it with sensor information and messages "yelled" by other players on the field. The player issues commands to the server by sending it UDP datagram messages no faster than once every 100 milliseconds. Commands are not queued: if the player issues commands faster than this, they are simply ignored by the server. The server updates its internal world model every 10 milliseconds, which means that under ideal load conditions a player's command request is processed within 10 milliseconds of its issuance.

[1] These students were: Charles Hohn, Jonathan Farris, and Gary Jackson

Sensor information relays game status and the relative positions of viewable objects on the field. The player may choose to receive either high- or low-quality sensor information (low-quality information is received more often but is nearly useless, as it does not provide directional information — to my knowledge no teams used it at all). Players receive information only about other objects they can "see" given the direction they are facing. Players may choose narrow (45 degrees), medium (90 degrees), or wide (180 degrees) ranges of view, but the wider the range, the more slowly sensor updates are received. Almost all entrants in the competition, including us, opted for medium-range sensing. Sensor information includes only:

- The ball position and relative movement.
- Positions and relative movement of other players on the field. If players are far enough away, their jersey numbers cannot be ascertained. If players are very far away, the *team* they're on cannot be determined.
- Goal positions.
- The position of flags placed at corners of the field, and on the edges of the field at its midpoint.
- The distance to and perceived angle of the soccer field boundary line crossing the player's field of vision.
- The state of the ball in play, including free, goal, side, and corner kicks, out-of-bounds, pre-game and mid-game setup, kickoffs, etc.

Sensing is further complicated in three ways. First, the coordinate positions of objects are given *relative to the player's position and the direction he is facing*, but the server does not tell the player any information about his own whereabouts. Second, the server gives information only about "very close" objects or objects inside the player's field of view (the widest is only 180 degrees). Third, sensor information is very noisy, and noise increases as the distance to an object increases.

Player movement is nonholonomic, that is, players do not have the ability to move immediately in any direction, but instead must first turn and then dash, complicating control algorithms greatly. And like sensor information, movement is also subject to a great deal of noise. Each movement cycle, a player may issue one of several commands:

- Rotate *n* degrees from his current facing direction.
- Dash in the direction he is facing with *n* power. A player must repeatedly dash to keep up forward movement. Dashing also decreases stamina; players that dash with high power will soon start running much slower than they realize. A player is not told his current stamina, nor how fast he is currently running. Players may also dash backwards, but at most a third of maximum power.
- Kick the ball in a certain direction (relative to the direction the player is facing) and with a certain power. Players can only kick a ball when it is "sufficiently close" to them.
- Yell a message. There are strong limitations on this. Messages may be yelled only very infrequently, and fellow players can hear only so many messages each sensor cycle. Further, players aren't told where a yell came from; hence yells can be (and often are) faked by opponents.
- Move the player to a specific (X,Y) coordinate position and facing a specific angle. This is only permitted while the ball is out of play.

Play happens in real time. If a player cannot process sensor information or make moves fast enough, he will fall behind. The soccer server also maintains complex dynamics among moving objects. Balls and players have acceleration and momentum, and cannot immediately stop, change direction, or move at a certain velocity on command. Players and the ball have different mass, hence can move at different rates (the ball can move much faster). Players take up space and collide with the ball and other players inelastically. When a team is given possession of the ball (for a goal kick, perhaps), the server "bumps" opposing players from the general ball area. Though not used in the competition, the simulator can also provide wind and other complicating conditions.

When a ball is kicked out-of-bounds, ball control is transferred to opponents and the ball is moved to the appropriate kick-in position per regulation rules. Goals are scored when the ball passes through the goal line. The server allows a human referee to make foul calls for ungentlemanly play. The RoboCup97 simulator enforced standard soccer rules with two very large exceptions. First, as robot players have no hands, there is no goalie, and no goalie area. Second, there is no offside rule.

## 5.2 The Challenge for Evolutionary Computation

As should be obvious from the above description, the Soccer Server domain is very complex, with a large number of options and controls, and a correspondingly large number of boundary conditions and special cases that must be accounted for. This alone makes it a tough problem to tackle with GP.

As if the soccer server's complex dynamics didn't make evolving a robot team hard enough, the server also adds one enormously problematic issue: time. As provided, the soccer server runs in real-time, and all twenty-two players connect to it via separate UDP sockets. Because of the enforced ten-millisecond delay between world model updates, games can last from many seconds to several minutes. Game play can be sped up by hacking the server and players into a unified program (removing UDP) and eliminating the ten-millisecond delay. Unfortunately, we found that for a variety of reasons this does not increase speed as dramatically as might be imagined, and if not carefully done, runs the risk of changing the game dynamics (and hence "changing" the game the players would optimize over).

The reason all this is such a problem is that evolving a computer program to work successfully in this domain would likely require a very large number of evaluations, and each new evaluation is another soccer simulator trial. In previous experiments with considerably simpler cooperation domains, [Luke and Spector 1996] found that genetic programming could require on order of 100,000 evaluations to find a reasonable solution. The soccer domain could be much worse. Consider that just 100,000 5-minute-long evaluations in serial in the soccer server could require up to a full year of evolution time.

Our challenge was to cut this down from years to a few weeks or months, but still produce a relatively good-playing soccer team from only a few evolutionary runs. We accomplished this in several ways:

- Brute force. We sped up play by performing up to 200 single-player evaluations and up to 32 full-team game evaluations in parallel on a supercomputer cluster. We also cut down game time from 10 minutes to between 20 seconds and one minute.

- We attempted to cut down the necessary population size and number of generations to produce a reasonable team.

- We developed an additional layer of software which simplified and orthogonalized the domain, eliminating many of the boundary conditions the GP programs would have to account for. We also spent much time designing a function set and evaluation criteria to promote better evolution in the domain.

- We performed simultaneous runs with different genome structures to give us a wider set of options as competition time neared.

## 5.3 Evolving Soccer Behaviors

As the Soccer Simulator dynamics were very complex, we began by coding a large multithreaded socket library which simplified some of the oddities of the domain. The library received all incoming sensor information and

| Function Syntax | Returns | Description |
|---|---|---|
| home | v | A vector to my home (my starting position). |
| ball | v | A vector to the ball. |
| findball | v | A zero-length vector to the ball. |
| block-goal | v | A vector to the closest point on the line segment between the ball and the goal I defend. |
| away-mates | v | A vector away from known teammates, computed as the inverse of $\sum_{m \in \{\textit{vectors to teammates}\}} \frac{max-\|\|m\|\|}{\|\|m\|\|} m$ |
| away-opps | v | A vector away from known opponents, computed as the inverse of $\sum_{o \in \{\textit{vectors to opponents}\}} \frac{max-\|\|o\|\|}{\|\|o\|\|} o$ |
| squad1 | b | t if I am first in my squad, else nil. |
| opp-closer | b | t if an opponent is closer to the ball than I am, else nil. |
| mate-closer | b | t if a teammate is closer to the ball than I am, else nil. |
| (home-of *i*) | v | A vector to the home of teammate *i*. |
| (block-near-opp *v*) | v | A vector to the closest point on the line segment between the ball and the nearest known opponent to me. If there is no known opponent, return *v*. |
| (mate *i v*) | v | A vector to teammate *i*. If I can't see him, return *v*. |
| (if-v *b v1 v2*) | v | If *b* is t, return *v1*, else return *v2*. |
| (sight *v*) | v | Rotate *v* just enough to keep the ball in sight. |
| (ofme *i*) | b | Return t if the ball is within $\frac{i}{max}$ units of me, else nil. |
| (ofhome *i*) | b | Return t if the ball is within $\frac{i}{max}$ units of my home, else nil. |
| (ofgoal *i*) | b | Return t if the ball is within $\frac{i}{max}$ units of the goal, else nil. |
| (weight-+ *i v1 v2*) | v | Return $\frac{v1(i)+v2(9-i)}{9}$. |
| (far-mate *i k*) | k | A vector to the most offensive-positioned teammate who can receive the ball with at least $\frac{i+1}{10}$ probability. If none, return *k*. |
| (mate-m *i1 i2 k*) | k | A vector to teammate *i1* if his position is known and he can receive the ball with at least $\frac{i2+1}{10}$ probability. If not, return *k*. |
| (kick-goal *i k*) | k | A vector to the goal if the kick will be successful with at least $\frac{i+1}{10}$ probability. If not, return *k*. |
| (dribble *i k*) | k | A "dribble" kick of size $\frac{i}{20}(max)$ in the direction of *k*. |
| kick-goal! | k | Kick to the goal. |
| far-mate! | k | Kick to the most offensive-positioned teammate. If there is none, kick to the goal. |
| kick-clear | k | Kick out of the goal area. Namely, kick away from opponents as computed with (away-opps), but adjust the direction so that it is at least 135 degrees from the goal I defend. |
| (kick-if *b k1 k2*) | k | If *b* is t, return *k1*, else return *k2*. |
| (opponent-close *i*) | b | Return t if an opponent is within $\frac{max}{(1.5)i}$ of me. |
| 0,1,2,3,4,5,6,7,8,9 | i | Constant integer values. |

**Legend.** *max* is the approximate maximum distance of kicking, set to 35. *k* is a kick-vector, *v* is a move-vector, *i* is an integer, and *b* is a boolean. Functions are strongly-typed by their return types and argument types, using atomic STGP.

Table 5.1: Genetic Programming Function Set for the Soccer Domain

boiled it down into the *absolute* position of all visible teammates and opponents (and the player himself), the ball, and the goals. We included a simple state-estimation mechanism that interpolated teammate and opponent positions in-between sensor cycles and maintained estimates of the player's stamina. The boiled-down domain provided information about whose ball it was during free-kicks, goal-kicks, etc., but this information was largely unused as the server would keep players out of the kick area. As we decided to ignore the evolutionary complexities of intercommunication, our domain also eliminated the ability to yell or listen.

We also made some significant changes to the traditional GP genome. Instead of a player algorithm consisting of a single tree, our players consisted of two algorithm trees. The first tree was responsible for moving the player, and when evaluated would output a vector which gave the direction and speed with which to turn and dash. The second tree was responsible for making kicks, and when evaluated would output a vector which gave the direction and power with which to kick the ball. At evaluation-time, the program executing the player's moves would follow the instructions of one or the other tree based on the following simplifying state-rules:

- If the player can see the ball and is close enough to kick the ball, call the kick tree. Kick the ball as told, moving the player slightly out-of-the-way if necessary. Turn in the direction the ball was kicked.
- If the player can see the ball but isn't close enough to kick it, call the move tree. Turn and dash as told; if the player can continue to watch the ball by doing so, dash instead by moving in reverse.
- If the player cannot see the ball, turn in the direction last turned until the player can see it.

This state mechanism eliminated a great many troublesome boundary conditions. First, by combining "movement" into turn-dash pairs, we allowed the GP tree function set to assume its player had holonomic movement. Second, by doing everything reasonable to keep the ball in view, we were able to eliminate many of the boundary conditions which occur when ball suddenly disappears due to arbitrary player movement (a big problem in our early tests). Third, no yelling was used.

Before evolving a team, we had to create the set of low-level "basic" behavior functions to be used by its players. This required some compromise. Ideally, we would have liked to produce soccer players out of a variety of very low-level, generic vector functions. This would have allowed us to say that in no way did we bias the function set to produce certain kinds of strategies. But our early tests suggested that the domain was so complex that there was little hope of evolving a team with this kind of function set. Instead, we designed a large set of functions (some generic, some specialized) we thought would have particular utility in the soccer domain. In doing so, we tried to stay as general as possible but still come up with a function set that we thought stood a chance of evolving successfully.

To achieve this, we used Strongly-Typed GP [Montana 1995] to provide for a variety of different types of data (booleans, vectors, etc.) accepted and returned by GP functions, restricting tree formation to conform to these type rules. This allowed us to include a large, rich set of GP functions (allowing for many more player options), but still constrain the possible permutations of function combinations.

Table 5.1 gives a sampling of the basic functions we provided our GP system with which to build GP trees. We decided early on to enrich a basic function set of vector operators and if-then control constructs with some relatively domain-specific behavior functions. Some of these behaviors could be derived directly from the Soccer Server's sensor information. This included vector functions like kick-goal!, or home. Other behaviors were important to include but were hand-coded because we found evolving them unsuccessful, at least within our limited time constraints. These included good ball interception (a surprisingly complex task), which was formed into ball, or moving optimally to a point between two objects (forming (block-near-opp $v$), for example).

We used genetic programming to evolve other low-level behaviors. Notably, we used symbolic regression to evolve functions determining the probability of a successful goal-kick or a pass to a teammate, given opponents in various positions [Hohn 1997]. Our symbolic regression data points were generated by playing actual trials in the soccer server (with a kicker, receiver, and opponent for teammate-pass trials, or just a kicker and opponent for

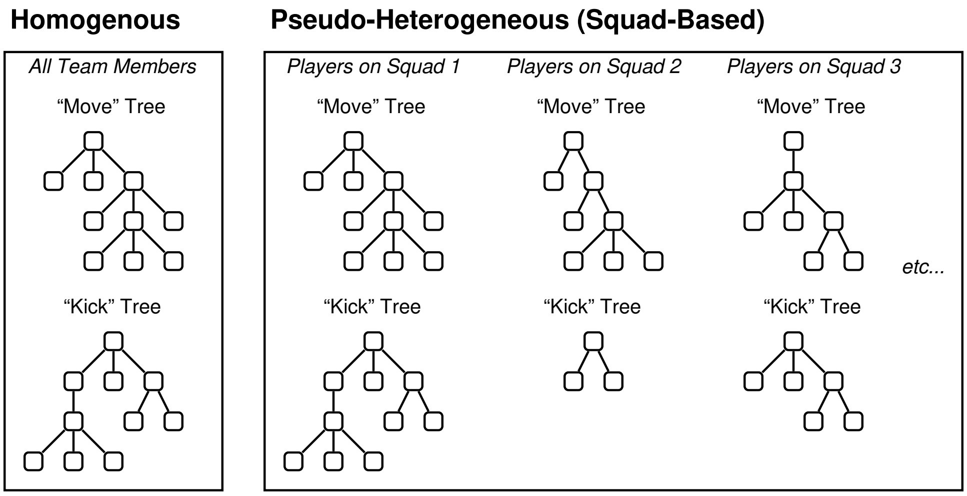



Figure 5.1: Homogeneous and Pseudo-Heterogeneous (Squad-Based) Genome Encodings

goal-kick trials). The evolved results formed the probabilistic mechanism behind the decision-making functions (kick-goal *i k*), (mate-m *i1 i2 k*), and (far-mate *i k*).

Given a basic function set, there are a variety of ways to use genetic programming to "evolve" a soccer team. An obvious approach is to form teams from populations of individual players. The difficulty with this approach is that it introduces the *credit assignment problem:* when a team wins (or loses), how should the blame or credit be spread among the various teammates? We took a different approach tried before in [Luke and Spector 1996]: the genetic programming genome is an entire team; all the players in a team stay together through evaluations, breeding, and death.

This raises the question of a homogeneous or heterogeneous team approach. Using a homogeneous team approach, each soccer player would follow effectively the same algorithm, and so a GP genome would be a single kick-move tree pair used by all teammates during play. With a heterogeneous approach, each soccer player would develop and follow its own unique algorithm, so a GP genome would be not just a kick-move tree pair, but a forest of such pairs, one pair per player. In a domain where heterogeneity is useful, the heterogeneous approach provides considerably more flexibility and the promise of specialized behaviors and coordination. However, homogeneous approaches can take far less time to evolve, since they require evolving only a single algorithm rather than (in this case) eleven algorithms.

To implement a fully heterogeneous approach in the soccer domain would necessitate evolving a genome consisting of twenty-two separate GP trees, far more than could reasonably evolve in the time available. Instead, we ran separate runs for homogeneous teams and for hybrid *pseudo-heterogeneous* teams (see Figure 5.1). The hybrid teams were divided into six squads of one or two players each; each squad evolved a separate algorithm used by all players in the squad. This way, pseudo-heterogeneous teams had genomes of six turn-dash tree pairs. Each player could still develop his own unique behavior, because the primordial soup included functions which let each player distinguish himself from his squadmates.

Because our genomes consisted of forests of trees, we adapted the GP crossover and mutation operators to accommodate this. In our runs, crossover and mutation would apply only to a single tree in the genome. For both homogeneous and pseudo-heterogeneous approaches, we disallowed crossover between a kick tree and a move tree. For pseudo-heterogeneous approaches, we allowed trees to cross over only if they were from the same squad: this "restricted breeding" has in previous experience proven useful in promoting specialization [Luke and Spector 1996]. We also introduced a special crossover operator, *root crossover*, which swapped whole trees at the root instead of swapping subtrees. This let teams effectively "trade players", which we hoped would spread good strategies through the population more rapidly.

To speed up evolution time, we used small population sizes between 200 and 400. We felt these small populations (given the problem complexity) required a somewhat unusual mix of breeding operators. We permitted no reproduction, and decided to use a large dose of subtree mutation (up to 70%, depending on the run), with a GROW depth bound $D$ of 4, to produce higher overall fitness with the small population, and to stave off premature convergence. The rest consisted of subtree crossover and a small (10%) amount of root crossover.

Another issue was the evaluation function needed to assess a genome's fitness. One way to assess a team would be to play the team against one or more hand-created opponent teams of known difficulty. There are two problems with this approach. First, from our experience, evolutionary computation strategies often work more efficiently when the difficulty of the problem ramps up as evolution progresses, that is, as the population gets better, the problem gets harder. A good ramping with a suite of pre-created opponents is difficult to gauge. Second, unless there are several opponents at any particular difficulty level, one runs the risk of evolving a team to beat that *particular set* of hand-made opponents, instead of generalizing to "good" soccer.

We opted instead to use a particular form of competitive fitness. In our implementation, the GP system first paired off teams in the population, then played one match with each pair. In this way a teams' fitness was assessed based on its success against an opposing team from the same population. This fitness assessment approach naturally ramps problem difficulty because teams in the population play against peers of approximate ability. Cooperative fitness can also promote generalization, because the number of "opponents" a population faces is the size of the population itself, rather than a small set of predetermined fitness cases.

In the interest of time, our implementation assessed a team's fitness based on a single game against a single opposing team (that is, for each generation of size $N$, there were $\frac{N}{2}$ games played). This probably did not do a good enough job of distinguishing high-quality teams from middling ones. In retrospect, it may have been better to determine fitness through a single-elimination tournament, which would have only required twice as many games ($N - 1$) as our approach did. [Angeline and Pollack 1993] gives a detailed description of this and other approaches.

We initially based fitness on a variety of game factors including the number of goals, time in possession of the ball, average position of the ball, number of successful passes, etc. However, we found that in our early runs, the entire population would converge to very poor solutions. Ultimately, we found that by simply basing fitness on the difference in goals scored by the two opponents, the population avoided such convergence. At first glance, such a simplistic fitness assessment would seem an overly crude measure, as many early games might end with 0–0 scores. Luckily, this turned out to be untrue. We discovered that initial game scores were in fact very high: vectors to the ball and to the goal were fundamental parts of the function set, so teams did simple offense well, but defense poorly. Only later, as teams evolved better defensive strategies, would scores come down to more reasonable levels.

We performed our GP runs in parallel using a custom strongly-typed, multithreaded version of lil-gp 1.1 [Zongker and Punch 1996], running on a 40-node DEC Alpha supercomputer. At evaluation time, the system split the generation into groups and assigned each group to a separate evaluation thread. These evaluation threads would work with our socket communication library to pair off teams and play games in separate Soccer Server processes.

The extremely long evaluation time (a single generation would take a day's computer time or longer) necessitated some drastic measures to ensure that our population converged to its best suboptima possible prior to the competition. At some point I felt we would need the population to stop global searches and start narrowly tweaking its best strategies to date in preparation for the competition. As such, we ran the final runs for forty generations, at which time we re-introduced into the population high-fitness individuals from past generations. The intent of this unusual step was to force the population to rapidly converge to a narrow set of suboptima. We then continued runs up to the time of the RoboCup-97 competition (for twelve more generations).

Just prior to the competition, we held a "tournament of champions" among the twenty highest-performing teams at that point, and submitted the winner. While I feel that, given enough evolution time, the learned strategies of the pseudo-heterogeneous teams might ultimately outperform the homogeneous teams, the best teams at competition time (including the one we submitted) were homogeneous.


Figure 5.2: Some Players Begin to Hang Back and Protect the Goal, while Others Chase after the Ball

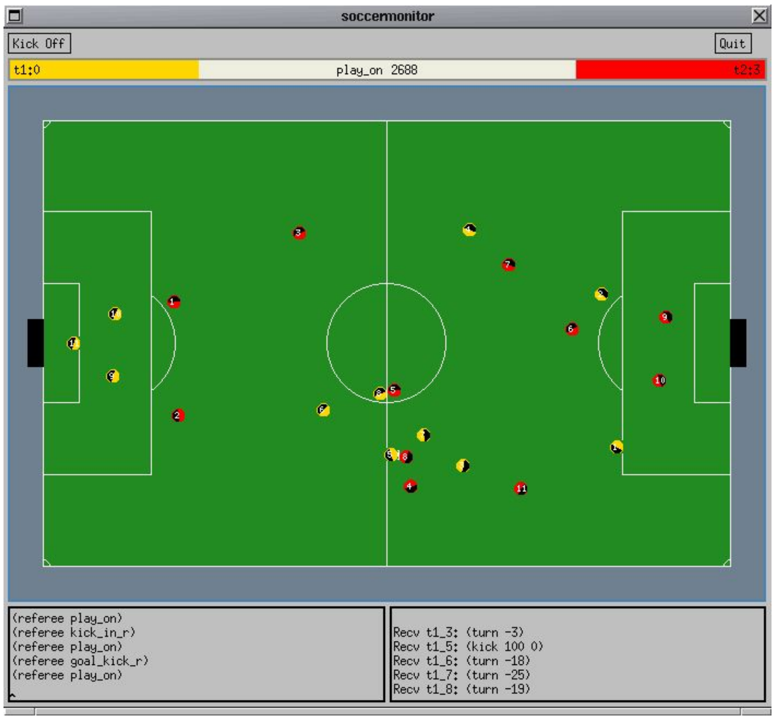



Figure 5.3: Teams Eventually Learn to Disperse Themselves throughout the Field

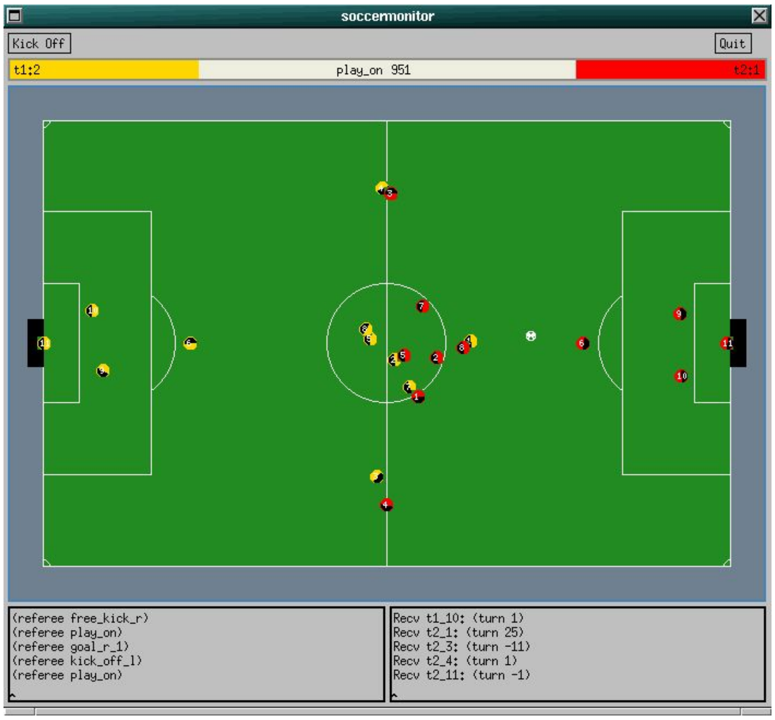



Figure 5.4: A Competition Between Two Initial Random (and Randomly Moving) Teams


Figure 5.5: "Kiddie-Soccer", a Problematic Early Suboptimal Strategy, where Everyone on the Team Would Go After the Ball and Try to Kick It into the Goal

## 5.4 A History of Evolution

One of the fun parts of working with GP is being able to watch the population learn. In a typical GP experiment one would conduct a large number of runs, which provides a statistically meaningful analysis of population growth and change. For obvious reasons, this was not possible for us to do. Given the one-shot nature of our RoboCup runs (the final run took several months' time), our observations of population development are therefore admittedly anecdotal. Still, we observed some very interesting trends.

Our initial random teams consisted primarily of players which wandered aimlessly, spun in place, stared at the ball, or chased after teammates. Because ball and kick-goal! were basic terminal functions, there were occasional players which would go to the ball and kick it to the goal. These players helped their teams rack up stratospheric scores against helpless opponents. Figure 5.4 shows two random teams playing.

Early populations produced all sorts of bizarre strategies. One particular favorite was a (homogeneous) competition of one team programmed to move away from the ball, against another team programmed to move away from the first team. Thankfully, such strategies didn't last for many generations.

One suboptimal strategy, however, was particularly troublesome: "everyone chase after the ball and kick it into the goal", otherwise known as "kiddie-soccer", shown in Figure 5.5. This strategy gained dominance because early teams had effectively no defensive ability. Kiddie-soccer proved to be a major obstacle to evolving better strategies. The overwhelming tendency to converge to kiddie-soccer and similar strategies was the chief reason behind our simplification of the evaluation function (to be based only on goals scored). After we simplified the evaluation function, the population eventually found its way out of the kiddie-soccer suboptima and on to better strategies.

After a number of generations, the population as a whole began to develop rudimentary defensive ability. One approach we noted was to have a few players hang back near the goal when not close to the ball (Figure 5.2). Most teams still had many players which clumped around the ball, kiddie-soccer-style, this simple defense effectively eliminated the long-distance goal shots which had created such high scores in the past.

Eventually teams began to disperse players throughout the field and to pass to teammates when appropriate instead of kicking straight to the goal, shown in Figure 5.3. Homogeneous teams did this usually by using players' home positions and information about nearby teammates and ball position. But some pseudo-heterogeneous teams appeared to be forming separate offensive and defensive squad algorithms. It was unfortunate that the pseudo-heterogeneous teams were not sufficiently fit by the time RoboCup arrived; we suspect that given more time, this approach could have ultimately yielded some very good strategies.

## 5.5 Bloat

It is probably serendipitous that we decided to use large amounts of subtree mutation, given the small populations in our runs. As Chapter 7 details, subtree mutation often yields better fitness results and smaller tree sizes when dealing with such small populations.

Even so, just prior to the tournament-of-champions stage, individuals had bloated to the point where their size was interfering with finding crossover and mutation points due to the imposed Koza-style maximum depth limit (17). By the fortieth generation, the mean homogeneous individual size was 187 nodes, and the mean heterogeneous individual's size was 551 nodes. On a per-tree basis, this suggests that heterogeneous individuals did a better job at fighting bloat, since heterogeneous individuals have twelve trees, and homogeneous individuals have only two trees. However, only one crossover or mutation event occurred per individual per generation; thus the growth rate for both approaches should have been the same, yet heterogeneous individuals far outstripped homogeneous ones. The reason for this is unknown.

The code that follows is a typical example of a homogeneous team late in evolution, which weighs in at 179 nodes. Note that the kick tree is much smaller than the dash tree. This trend was nearly universal throughout the population. The reason for this is also unknown.

**Kick:** (kick-goal 1 (kick-goal 0 (kick-if (opponent-close 1) (kick-goal 1 (mate-m 1 1 kick-goal!)) (kick-goal 0 (kick-goal 2 far-mate!)))))

**Dash:** (if-v mate-closer (if-v (ofme 1) (if-v mate-closer (if-v (ofme 1) (if-v (ofgoal 0) (if-v (ofhome 1) (if-v opp-closer (if-v mate-closer (if-v squad1 (if-v squad1 (if-v (ofhome 3) (if-v (ofme 5) (home-of 1) (if-v (ofhome 7) (mate 7 block-goal) (if-v opp-closer block-goal ball))) (inv (if-v (ofme 2) (mate 9 home) (inv findball)))) home) (if-v (ofgoal 2) (mate 1 (if-v squad1 away-mates ball)) (weight-+ 1 (if-v mate-closer (if-v (ofme 1) block-goal (weight-+ 1 ball home)) ball) (sight (if-v (ofhome 1) (if-v (ofme 8) (home-of 8) (block-near-opp away-mates)) (if-v (ofme 2) (if-v opp-closer (if-v mate-closer (sight (if-v opp-closer (if-v opp-closer (if-v opp-closer home home) home) ball)) ball) ball) (home-of 1))))))) ball) ball) (if-v opp-closer home findball)) (if-v (ofhome 1) (if-v (ofhome 1) (weight-+ 1 (if-v (ofme 1) (block-near-opp away-opps) (if-v mate-closer ball away-opps)) home) (weight-+ 1 (if-v (ofhome 2) (if-v squad1 away-opps block-goal) ball) (if-v opp-closer (if-v mate-closer home ball) (sight ball)))) (home-of 1))) (weight-+ 1 ball home)) ball) (weight-+ 1 ball home)) ball)

Could bloating have interfered with fitness assessment? This is a difficult question to answer. If a tree was sufficiently large, the evaluation of the tree might be slow enough to raise the individual's decision-making time to beyond the 100-millisecond action rate of the soccer simulator, possibly causing sensor information to be dropped. This would not affect the population as a whole, however, because we had adopted a multithreaded fitness evaluation architecture, in which each game was played in its own preemptive thread. This meant that if a large team was playing, only that team and its opponent would miss sensor packets. While this might conceivably bias the results of that game, it was unlikely to affect games in other threads.

## 5.6 Summary

RoboCup is an example of genetic programming successfully applied to a problem of some difficulty. The complexity of the domain and the long evaluation times involved can present quite a challenge for evolutionary computation. Still, GP was able to produce soccer robot teams competitive with hand-crafted offerings. This problem domain also highlights the problem of bloating in GP. For this domain, the evolution run capped tree depth at 17, and well before forty generations individuals had expanded to bump up against this limit.

# Chapter 6

# Edge Encoding: Evolving Sparse Graph Structures

This chapter introduces *edge encoding*, a genetic programming graph-rewriting technique similar to cellular encoding, which I designed to evolve sparse networks for use in applications like finite state automata and electrical circuits.

Although cellular encoding is a powerful technique, it has weaknesses. First, cellular encoding's genome traversal (breadth-first) and highly execution-order-dependent nodes tend to cause crossover and mutation to randomize an individual. Other disadvantages stem from cellular encoding's use of graph vertices as the target of its operations: as cellular encoding modifies vertices, the edges multiply rapidly, but cellular encoding provides only a very limited mechanism, *link registers,* to label and modify individual edges. Additionally, cellular encoding tends to produce graphs with large numbers of edges. This is useful for cellular encoding's primary focus, namely dense graphs such as those found in feedforward networks. However, it is not desirable for evolving sparse graphs.

Edge encoding addresses many of these issues. The technique uses node functions which operate on edges rather than vertices, and does not require cellular encoding's total-ordered breadth-based tree search. Even so, edge encoding can produce any graph necessary, depending on the function set used. This chapter describes edge encoding applied to evolving finite state automata, and shows that it can evolve all FSAs of interest and, in a more advanced form, all possible graph structures. The chapter then describes an experiment with edge encoding in inducing the Tomita Language Set, a popular language induction benchmark. This experiment further illustrates bloating characteristics inherent in edge encoding.

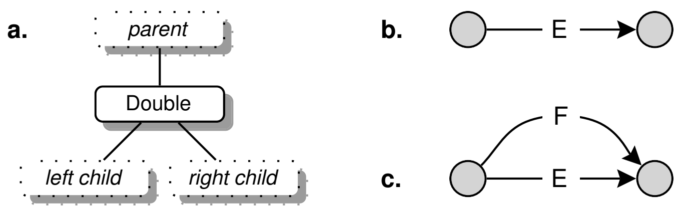
This chapter freely mixes graph structures and GP trees in its discussion. To avoid confusion, I will always use "vertex" to describe the nodes in graphs and "node" or "function" to describe the nodes in GP trees.

## 6.1 Edge Encoding

An edge encoding is a genetic programming tree which produces a directed graph when evaluated. This graph is then used in the problem domain as appropriate — as an electrical circuit, FSA, etc. Similar to cellular encoding, edge encoding vertices are each passed a single edge in the graph; and make modifications on their respective edges, possibly adding more edges and vertices to the graph. The modified edge is passed to the first child of the node. If the node has more than one child, it must create new edges in its edge-modification step, one vertex for each of its other children. A node may also delete its edge from the graph. However unlike cellular encoding, edge encoding uses an ordinary depth-first traversal like standard GP.

For example, consider the double function shown in Figure 6.1. This function has two children in the encoding tree. It receives from its parent a single edge $E(a,b)$ in the graph (where $a$ is the tail of the edge $E$, and $b$ is $E$'s head). From $E(a,b)$, double "grows" an additional edge $F(a,b)$. These two edges are each passed to child functions for additional modifications; $E$ is passed to the double's left child, and $F$ is passed to its right child.

Edge encoding's graph-generation process begins with a graph consisting of a single edge. This edge is passed to the root node in the edge encoding tree, which modifies the edge and passes resultant edges to its




**Legend.** (a) shows the function relative to its parent and children functions in the encoding tree. (b) shows the initial edge *E* passed to double from its parent. (c) shows the edges *E* and *F* after double's execution. *E* and *F* are then passed to double's left and right children, respectively.

Figure 6.1: The double Function

| Function Syntax | Arity | Description |
|---|---|---|
| double | 2 | Create an edge $F(a,b)$. |
| bud | 2 | Create a vertex $c$. Create an edge $F(b,c)$. |
| split | 2 | Create a vertex $c$. Modify $E$ to be $E(a,c)$. Create an edge $F(c,b)$. |
| loop | 2 | Create a self-loop edge $F(b,b)$. |
| reverse | 1 | Reverse $E$ to be $E(b,a)$. |

Table 6.1: Simple Topological Functions for Edge Encoding

children, and so on. Terminals in the edge encoding tree have no children, and so stop the modification process for a particular edge. After all nodes in the tree have made their modifications, the resultant graph is returned. Unlike cellular encoding, edge encoding can also work with a special form of automatically-defined macros.

The functions in a particular edge encoding are commonly of two forms. First, there are functions which change the topology of the graph, by adding or deleting edges or vertices. Second, there are functions which add semantics to the edges or vertices: labeling edges, assigning transfer functions to vertices, etc.

## 6.2 Encoding an NFA

To demonstrate an edge encoding, we begin with a simple set of basic functions which cannot describe all directed graphs, but are sufficient to build all nondeterministic finite-state automata (NFA). These functions each take an edge and no optional data from their parents, and pass on to children at most two resultant edges. A nice property of these functions is that although they have side effects (in the way of modifying the graph), these side effects are localized in such a way that the functions are referentially transparent. Thus nodes can be executed in any order, so long as parent functions are executed prior to child functions.

In this simple encoding, each individual consists of a single tree of functions. Assume that each function is passed some edge $E(a,b)$, which after processing is passed to the left child. The functions which describe the topology of the graph are shown in Table 6.1. These functions are sufficient to develop the topology of an NFA which recognizes any regular expression. To develop the full NFA, some custom semantic functions are necessary to define the starting and accepting states of the NFA and label the edges with tokens. For example, suppose one were trying to develop an NFA that matched the regular expression ((0|1)*101), from a language consisting only of 1's and 0's. Using edge encoding, five more functions are necessary; these functions are shown in Table 6.2.

Figure 6.2 shows an edge encoding genome using these functions whose phenotype is an NFA that reads the regular expression ((0|1)*101). Figure 6.3 shows the development of the NFA from this genome.

The following theorem shows that the functions reverse, split, and double alone are sufficient to build NFAs

| Function Syntax | Arity | Description |
|---|---|---|
| start | 1 | Assign the head of $E(a,b)$ (vertex $b$) to be a starting state. |
| accept | 1 | Assign the head of $E(a,b)$ (vertex $b$) to be an accepting state. It's valid for a state to be a starting state and an accepting state at the same time. |
| 1 | 0 | Label an edge with a "1", that is, define it to be an edge which can be traversed only on reading a 1. |
| 0 | 0 | Label an edge with a "0", that is, define it to be an edge which can be traversed only on reading a 0. |
| $\epsilon$ | 0 | Label an edge with an "$\epsilon$", that is, define it to be an edge which may be traversed without reading any token. |

Table 6.2: NFA Semantic Functions for Edge Encoding

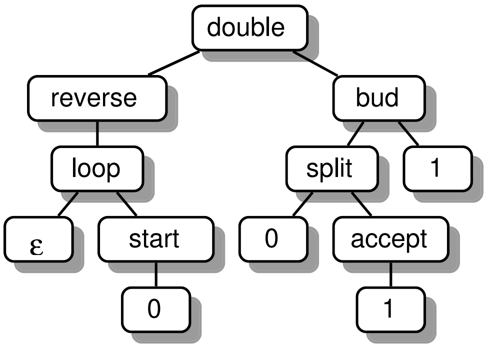


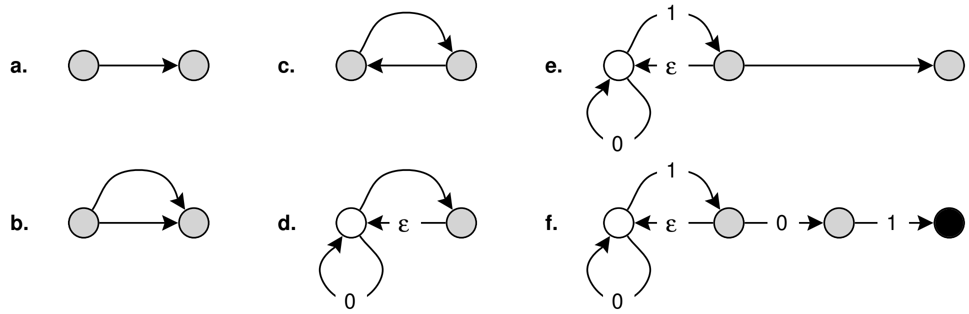

Figure 6.2: An Edge Encoding Genome which Describes an NFA that Reads the Regular Expression ((0|1)*101)

which parse all possible regular expressions. To do this, the proof roughly follows Thompson's construction as described in [Aho, Sethi, and Ullman 1988; Thompson 1968]. Thompson's construction first parses a regular expression into its subexpressions, then builds an NFA bottom-up by grouping smaller NFAs that represent those subexpressions.

**Theorem 1** Let $\Sigma$ be an alphabet. It is possible to construct a Nondeterministic Finite-State Automata (NFA) to parse any regular expression on $\Sigma$, using the following functions, as described in Tables 6.1 and 6.2: double, bud, split, loop, reverse, start, accept, $\epsilon$, plus one additional terminal function $t_\sigma$ for each symbol $\sigma \in \Sigma$. Each terminal function $t_\sigma$ labels the edge *E* passed to it to be "$\sigma$".

**Proof** By definition, for any regular expression **r**, one of the following is true.

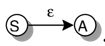
- **r** is $\epsilon$.
- **r** is a symbol $\sigma \in \Sigma$.
- **r** can be broken down into **s t** for some regular expressions **s** and **t**.
- **r** can be broken down into **s**|**t** for some regular expressions **s** and **t**.
- **r** can be broken down into **s*** for some regular expression **s**.
- **r** can be broken down into **(s)** for some regular expression **s**.



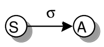

**Legend.** (a) The initial edge. (b) After applying double. (c) After applying reverse. (d) After applying loop, S, $\epsilon$, and 0. The white circle is a starting state. (e) After applying bud and 1. (f) After applying split, 0, A, and 1. The black circle is an accepting state.

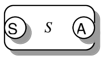
Figure 6.3: The Growth of the NFA from the Encoding in Figure 6.2

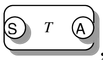
For each of these cases, we provide the edge encoding which produces the appropriate NFA, and we also provide the NFA itself. Each constructed NFA will have a one start-state and one accepting-state, indicated with an "S" and "A" respectively.

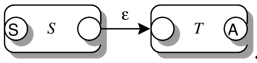
**Case 1.** **r** is $\epsilon$. In this case, the edge encoding for **r** is simply $\epsilon$. This produces the NFA 

.

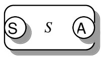
**Case 2.** **r** is a symbol $\sigma \in \Sigma$. In this case, the edge encoding for **r** is simply $t_\sigma$, where $t_\sigma$ is the terminal that corresponds with **r** as described above. This produces the NFA 

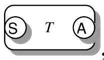

**Case 3.** **r** can be broken down into **s t**. Let $S$ and $T$ be the edge encodings for **s** and **t**, respectively, with NFAs 

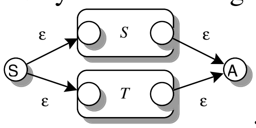
and 

, respectively. Then the edge encoding for **r** is (split $S$ (split $\epsilon$ $T$)), which produces the NFA 

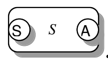
.

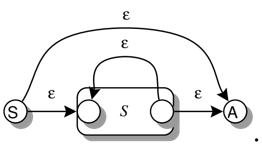
**Case 4.** **r** can be broken down into **s**|**t**. Let $S$ and $T$ be the edge encodings for **s** and **t**, respectively, with NFAs 

and 

, respectively. Then the edge encoding for **r** is (double (split (split $\epsilon$ $S$) $\epsilon$) (split (split $\epsilon$ $T$) $\epsilon$)), which produces the NFA 

.

**Case 5.** **r** can be broken down into **s***. Let $S$ be the edge encoding for **s**, encoding the NFA 

. Then the edge encoding for **r** is (double (split (split $\epsilon$ (double $S$ (reverse $\epsilon$))) $\epsilon$) $\epsilon$), which produces the NFA 

.

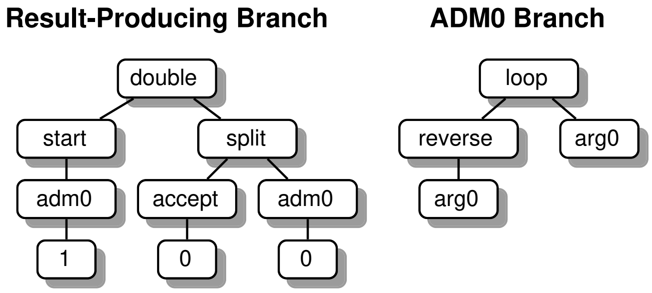
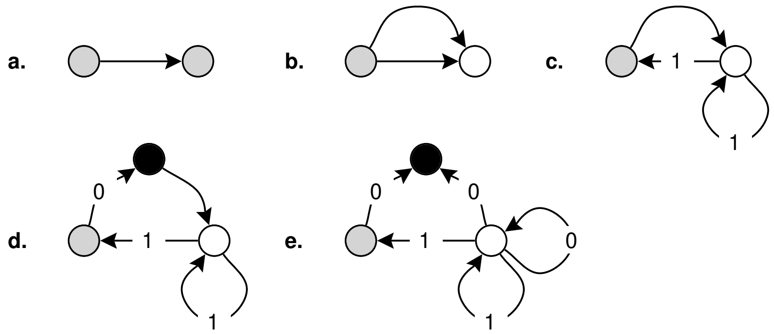
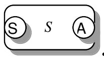
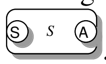



Figure 6.4: An Edge Encoding Individual with One Automatically-defined Macro with a Single Argument




**Legend.** (a) The initial edge. (b) After double and start. (c) The ADM0 branch is called, as are its arg0 nodes. (d) After split and accept. (e) The ADM0 branch is called again, as are its arg0 nodes.

Figure 6.5: The Growth of the Graph Structure Defined in Figure 6.4

**Case 6.** **r** can be broken down into **(s)**. Let *S* be the edge encoding for **s**, respectively, encoding the NFA . Then the edge encoding for **r** is the same as *S*, which producing (trivially) the NFA .

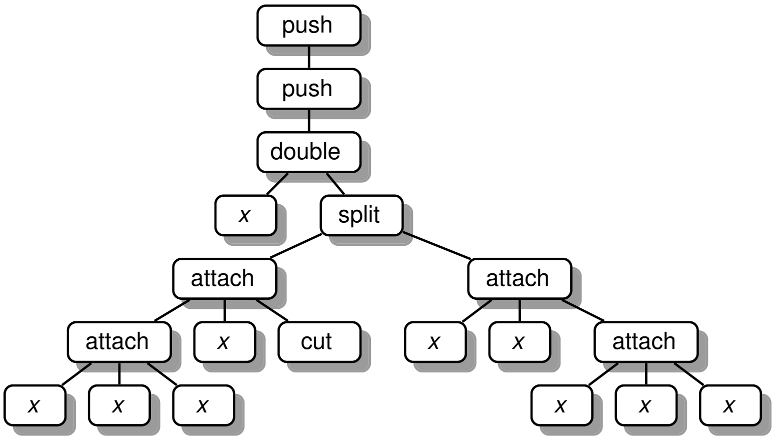
**Labeling Start and Accepting States.** So far we have indicated the start-state and accepting-state for an NFA, but have not actually labeled them as such in the edge encoding. Realize that as constructed in this proof, the start-state vertex will have one or more labeled edges leaving it, and the accepting-state vertex will have one or more labeled edges entering it. When we have finished constructing our final NFA, we can indicate its start state by finding the terminal function (we'll call it *A*) responsible for labeling some outgoing edge of the start-state vertex. We convert this terminal function into (reverse (start (reverse *A*))), which labels the start-state vertex. To indicate the NFA's accepting state, we find the terminal function (*B*) responsible for labeling some incoming edge of the accepting-state vertex. We convert this terminal function into (accept *B*), which labels the accepting-state vertex. ■

## 6.3 Additional Functions

The functions presented above develop enough planar graphs to describe, among other things, NFAs sufficient to accept any regular language. They do not describe more sophisticated planar graphs, however: for example, the complete graph of four vertices ($K_4$) cannot be generated using the above functions. One way to generate additional planar graphs is to use a function borrowed from cellular encoding: split-vertex. This function acts

| Function Syntax | Arity | Description |
|---|---|---|
| double | 2 | Create an additional edge $F(a,b)$. |
| loop | 2 | Create an additional edge $F(b,b)$. |
| reverse | 1 | Modify $E$ to be $E(b,a)$. |
| cut | 0 | Eliminate edge $E$. |
| push | 1 | Create a new vertex $c$. Make a copy of the stack. Push $c$ onto the copy, and give the copy to push's child. |
| attach | 3 | Make a copy of the stack. If the stack is empty, create and push a new vertex onto the copy. Let the top vertex on the stack copy be $c$. Create two new edges $F(a,c)$ and $G(b,c)$. Pop $c$ off the stack copy, then pass the copy to attach's children. This in effect "attaches" edges to the top vertex on the stack, forming a triangle $EFG$. |

Table 6.3: Functions for Creating General Graphs




**Legend.** Each instance of $x$ may be replaced by any terminal function which does not delete its edge or add new edges.

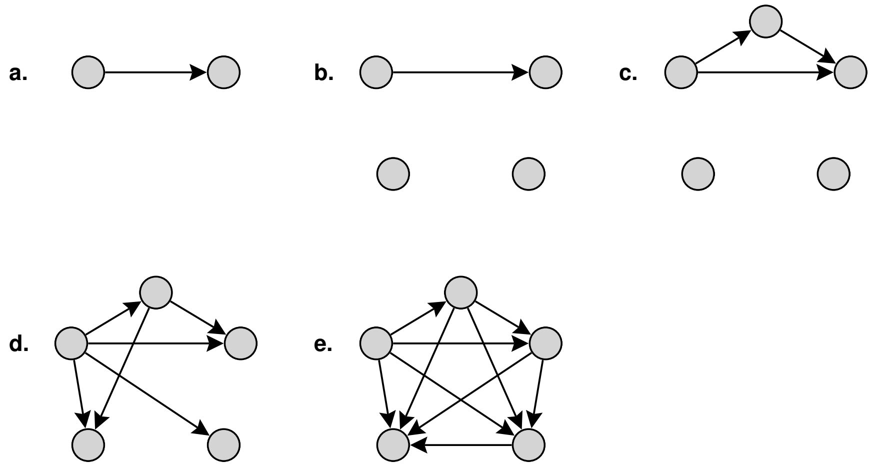
Figure 6.6: One of Many Genomes which Produce One Version of $K_5$ with No Multi-edges or Self-loops

on the vertex $b$ at the head of $E(a,b)$. First, split-vertex creates a new vertex $n$. Then for each edge $F(b,c)$, this function modifies $F$ to be $F(n,c)$. Lastly, the function adds a new edge $G(b,n)$. In essence, split-vertex splits vertex $b$ into two vertices $b$ and $n$, moving to $n$ all formerly outgoing edges of $b$, and adding single edge between $b$ and $n$. The corollary to split-vertex is merge, which merges the two vertices attached to its edge $E$, then deletes $E$.

Such functions come at a price, however. Because split-vertex modifies vertices associated with edges other than the function's official edge $E$, we must now ensure a total execution order throughout the tree, or else the interpretation of the tree is ambiguous. One way to do this is to always traverse the tree in depth-first, left-to-right order.

Edge encoding also works with a form of automatically-defined macros (ADMs), where the subtree represented by an arg node is given the edge handed to the corresponding node of the calling ADM parent. This can potentially grow large graphs from relatively small-sized trees. Figure 6.4 shows an individual with a single one-argument ADM, and Figure 6.5 shows the growth pattern of the phenotype graph.

There are many other functions which do not require an explicit execution order. For example: cut simply eliminates its edge from the graph. (export $a$ $b$) creates a new edge separated from the graph entirely. And there




**Legend.** 1. The initial edge. 2. After the two pushes. 3. After double and split. 4. After the leftmost attaches and cut. 5. After the remaining attaches.

Figure 6.7: The Growth of the Graph Structure Described in Figure 6.6

are variants of existing functions. For example, (loop-tail *a b*) might add a self-loop to the tail of the edge rather than its head, an abbreviation for (reverse (loop (reverse *a*) *b*)).

## 6.4 Creating All Graphs

Although edge encodings using the functions above can describe many interesting graphs, they cannot describe *all* graphs, especially ones that have nonplanar interconnections. To create all possible graphs, edge encoding must pass some limited vertex information as well as edge information. In this scheme, functions pass to their children not only an edge but a (possibly empty) stack of graph vertices. This permits earlier functions to create new vertices to which later functions may attach edges. A function may modify this stack by making a copy of it, modifying the copy, and passing the copy to its children. Functions which do not modify the stack (including all functions discussed so far) simply pass it through to their children. The initial stack passed to the root function is empty.

Assume that each function is passed an edge $E(a,b)$. Table 6.3 shows functions sufficient to describe the topology of all connected graphs of two or more vertices.

To create an arbitrary connected graph of $n$ vertices, first push $n$-2 vertices onto the stack in a series of push operations. Then create the complete graph $K_n$ of $n$ vertices, with some multi-edges, by performing attaches from the original edge to vertices, and from these newly attached edges to other vertices. Add multi-edges and self-loops as necessary using loop and double. reverse edges into their proper direction. Then cut out unwanted edges to form the resultant graph. Presumably there are more efficient ways to construct any particular graph, but this construction shows that the encodings can describe all possible graphs. Figures 6.6 and 6.7 give an example of an edge encoding which uses push and attach to produce a non-planar graph ($K_5$).

There are a large number of other possible stack-manipulation functions. For example, pop might pop a graph vertex off its stack without making any attachments. Or push-tail might push the tail vertex of an edge onto the stack. Variations on attach might attach only a single edge or attach edges but not pop the vertex off the stack afterwards.

| Language | Description |
|---|---|
| 1 | 1* |
| 2 | (10)* |
| 3 | (0\|11)*(1*\|(100(00\|1)*))<br>Any string without an odd number of consecutive 0's after an odd number of consecutive 1's |
| 4 | 1*((0\|00)11*)*(0\|00\|1*)<br>Any string without more than 3 consecutive 0's |
| 5 | (((1\|0)(1\|0))*(1\|0))\|((11\|00)*((01\|10)(00\|11)*(01\|10)(00\|11)*)*(11\|00)*)<br>Any string of even length which, making pairs, has an even number of (01)'s or (10)'s |
| 6 | ((0(01)*(1\|00))\|(1(10)*(0\|11)))*<br>Any string such that the difference between the number of 1's and 0's is a multiple of 3 |
| 7 | 0*1*0*1* |

Table 6.4: The Tomita Language Set

## 6.5 Experiment

[Luke, Hamahashi, and Kitano 1999] introduced a novel variable-size genome approach to EC inspired by gene expression networks. This approach evolved individuals as sets of "genes", each gene tagged with a floating-point locus value between 0.0 and 1.0 which indicated its location on the chromosome. In the experiments performed, each gene corresponded to a state in a DFA, and numerical values contained in each gene indicated the edges from state to state in the DFA and how they were to be labeled. Two experiments were performed to compare the effectiveness of this approach against existing language induction literature, using the Tomita language set [Tomita 1982] shown in Table 6.4.

To compare edge encoding with this approach, I repeated the second experiment in [Luke, Hamahashi, and Kitano 1999], using a population size of 500 and running for 100 generations. In this experiment, an individual's fitness is assessed by testing against a suite of 100 positive and 100 negative training examples for a given Tomita language. The standardized fitness is:

$$1.0 - \frac{\textit{correct negative examples} + \textit{correct positive examples}}{\textit{all negative examples} + \textit{all positive examples}}$$

This is an unfortunate fitness metric, since sometimes the negative examples far outnumber the positive examples (or vice versa), but it is the standard in the literature. The function set for this edge encoding is given in Tables 6.1 and 6.2. The suite of training examples was generated randomly at the beginning of the run, and used for all individuals during the course of the run.

After an evolutionary run completed, I measured the generalization accuracy of its highest-fitness individual. Generalization accuracy applies the same metric as that used for the standardized fitness, but instead using of 100 positive and 100 negative examples, it uses all possible strings up to 15 symbols in length. The resultant score estimates how closely the NFA was able to properly generalize to that particular Tomita language.

Table 6.5 compares the results with [Luke, Hamahashi, and Kitano 1999], showing the mean, variance, and best results for twenty runs. [Brave 1996] also performed a similar experiment using a cellular encoding approach, but with five times as many training examples and a large population (10,000); this is the same number of total individuals processed, but with a much larger training set. No results were reported except to say that for each language a single run discovered a fully generalizable solution, except for language 6, for which no solution was found even after ten runs.

| | Generalization Accuracy | | |
|---|---|---|---|
| **Language** | **Mean** | **Var.** | **Best** |
| | Edge Encoding | | |
| 1 | 0.9994 | 0.0000 | 1.0000 |
| 2 | 0.9986 | 0.0000 | 1.0000 |
| 3 | 0.9924 | 0.0011 | 1.0000 |
| 4 | 0.9940 | 0.0007 | 1.0000 |
| 5 | 0.7753 | 0.0243 | 1.0000 |
| 6 | 0.4979 | 0.0020 | 0.6272 |
| 7 | 0.9965 | 0.0001 | 1.0000 |
| | [Luke, Hamahashi, and Kitano 1999] | | |
| 1 | 0.9498 | 0.0500 | 1.0000 |
| 2 | 0.9477 | 0.0498 | 1.0000 |
| 3 | 0.9052 | 0.0504 | 1.0000 |
| 4 | 0.9469 | 0.0499 | 1.0000 |
| 5 | 0.7669 | 0.0564 | 1.0000 |
| 6 | 0.8309 | 0.0654 | 1.0000 |
| 7 | 0.9076 | 0.0510 | 1.0000 |

Table 6.5: Edge Encoding Experiment

As Table 6.5 shows, edge encoding outperformed [Luke, Hamahashi, and Kitano 1999] by a statistically significant margin, except for language 6, which the edge encoding technique was incapable of solving to any degree (a generalization score of 50% indicates random guesses). Interestingly, [Brave 1996] had similar problems with language 6 using a similar genetic programming technique.

### 6.5.1 Bloat

For the two languages which presented a difficulty to edge encoding (languages 5 and 6), trees bloated significantly during the run. By the final generation in language 6, the mean tree size had grown to 225.4. Excepting the three perfect solutions (which stopped the run prematurely), by the final generation of language 5, the mean tree size had grown to 205.4. These numbers indicate bloating bumping up against the Koza-style depth limit (17), which resulted in large numbers of directly reproduced individuals, effectively putting a stop to evolution. In contrast, language 1, which was always solved in the first generation, had a mean tree size of 10.3. The final generation took on average twenty times as long to execute as the first generation; were it not for the artificial depth limit, this would have been much larger still. Specific reasons for why edge encoding bloats in particular are discussed in Chapter 9.

## 6.6 Summary

Edge encoding allows genetic programming to describe any graph structure, and is particularly applicable to sparse, nonregular graphs such as are found in electrical circuits or finite state automata. This chapter, like the one before it, demonstrates a novel application of GP to a difficult task, in this case, language induction with the Tomita Language set using edge encoding. GP compares reasonably well with other examples in the literature when applied to this problem except on language 6, where it seems to have unusual difficulty worth future examination. However, once again bloating is considerable. As in RoboCup, edge encoding bloated to consume all permitted space. In the course of 100 generations, the population grew to the point where it was bumping up right against the imposed 17-depth limit.

# Chapter 7

# Comparing Mutation and Crossover

With a fixed-length representation form of EC, choosing a better breeding operator means choosing one which yields the highest overall fitness possible during the duration of a run for however long the run lasts. Genetic programming's breeding operator goals are slightly different: choosing a better operator means picking one which yields the highest overall fitness before bloat puts an effective stop to evolution. A good operator can do this in one of two ways. First, it can search more efficiently. Second, it can help delay bloat to extend the amount of time available for search. A good genetic programming breeding operator gives you the most bang for the buck.

As discussed in Section 4.4.1, there is presently significant debate in the genetic programming community over which breeding operators to use. Traditionally genetic programming has used roughly 95% subtree crossover and 5% direct reproduction, primarily due to inertia from the influence of [Koza 1992]. Despite evidence arguing against the existing of GP building blocks [O'Reilly and Oppacher 1995; Angeline 1997] crossover's dominance has still spawned a literature devoted to developing a GP building block hypothesis [Haynes 1997; Poli and Langdon 1997; Rosca 1997; Poli 2000]. However, as the EP and GP communities have converged, recent work has suggested that mutation can in fact outperform subtree crossover [Angeline 1998; Chellapilla 1997].

This chapter sets out to directly compare subtree crossover with the most common form of mutation, subtree mutation, to determine where each has superior fitness and tree size properties. One of the difficulties in comparing features in Genetic Programming is the large number of external parameters which can bias the results. Yet few studies perform experiments over a comprehensive set of parameter settings. This makes it easy for the sight of the trees to blind one to the forest as a whole. The experiment in this chapter tests subtree crossover and subtree mutation over a wide range of parameter settings to try to gauge trends in the "big picture".

## 7.1 Experiment

The experiment performed fifty independent runs with both subtree crossover and subtree mutation, over all 800 permutations of four parameters. The parameters chosen were the ones which seemed the most likely to affect the outcome of a genetic programming run:

- **Problem domain.** Four different domains were chosen of varying difficulty. The domains ranged from the trivial (6-Bit Multiplexer), to the moderate (Lawnmower, Symbolic Regression) to the relatively more difficult (Artificial Ant).

- **Population size.** The study performed runs with population sizes of 4, 8, 16, 32, 64, 128, 256, 512, 1024, and 2048.

- **Number of generations.** The study performed independent runs lasting 1, 2, 4, 8, 16, 32, 64, 128, 256, and 512 generations long.

- **Selectivity.** Runs were performed using tournament selection with tournament sizes of 2 and 7.

The experiment holds constant the myriad of other possible domain parameters, setting them to traditional default settings (those outlined in Chapter 3). The methodological justification for this is as follows. By making large-scale changes to the traditional domain settings, the experiment risks losing relevance to the large body of work which has used these settings in the past. And while one could choose to improve any number of parameter settings, there would always be someone arguing that if one only tried improvement *x*, the results might have been different.

In this experiment, the Symbolic Regression domain used no ephemeral random constants, and the Lawnmower domain used two ADFs as outlined in Chapter 3.4. Runs were performed using lil-gp 1.02 [Zongker and Punch 1996]. Subtree mutation used GROW with a depth bound $D$ from 2 to 6.

## 7.2 Fitness Results

The fitness results are shown in Figures 7.3 through 7.6. The landscape graphs show the adjusted fitness of the best individual of a run, averaged over fifty runs. Adjusted fitness ranges from 0.0 to 1.0, and higher fitness is better. Adjusted fitness is monotonic but not usually linear (depending on domain); a doubling in adjusted fitness does not necessarily translate to some doubling in "real fitness". The comparison graphs are black where crossover is better than mutation, white where mutation is better than crossover, and gray where the difference between the two is statistically insignificant (using a two-sample, two-tailed *t*-test at 95%).

Crossover had higher fitness in the majority of tests. As can be seen, the places where crossover or mutation is superior is highly domain and parameter-dependent. Nonetheless, there does appear to be an overall trend delimiting the areas where crossover or mutation is superior, at least for the 6-Bit Multiplexer, Lawnmower, and Symbolic Regression domains. This general trend is that mutation is more successful in smaller populations, and crossover is more successful in larger populations. Even when statistically significant, the difference between mutation and crossover is in many places surprisingly small (though one exception is Symbolic Regression).

The graphs are remarkably symmetrical with respect to choosing number of generations vs. population size. Certain domains favor one over the other only to a small degree (for example, Lawnmower favors number of generations, while Symbolic Regression favors population size). Traditional GP wisdom has been that favoring large populations (where crossover often works better) produces better results than favoring large numbers of generations (where mutation often works better); but these results do not support this.

## 7.3 Mean Tree Size Results

Figures 7.7 through 7.10 show the mean tree sizes of runs in the experiment, averaged over fifty runs. In the landscape graphs, higher areas denote larger trees. In the comparison graphs, black is where crossover is bigger, that is, worse than mutation, and white is where mutation is bigger (worse than crossover). Note that this is the opposite of the fitness graphs, where "bigger" fitness is better, not worse. Gray indicates areas where neither crossover nor mutation is significantly different (again using a two-sample, two-tailed *t*-test at 95%).

Mean tree size roughly corresponds to the amount of time necessary to evaluate and breed during the run. In the Lawnmower domain tree size is directly linked to evaluation time, since every node is executed exactly once. But in the Symbolic Regression, 6-Bit Multiplexer, and Artificial Ant domains it is possible to avoid evaluating unnecessary subtrees through short-circuiting. For example, if in (and *a b*) the subtree *a* always returns false, it is not necessary to evaluate the *b* subtree at all. This feature is known as *inviable code*, and will be discussed in detail in Chapter 9. Thus mean tree size is an accurate measure of runtime for the Lawnmower domain, but only an upper bound approximation for runtime in other domains.

Overall, crossover resulted in larger trees than mutation as the number of generations increased, especially in large populations. Mutation tended to produce larger trees in other areas. In some domains (Lawnmower, Artificial Ant), mutation almost never produced larger trees.

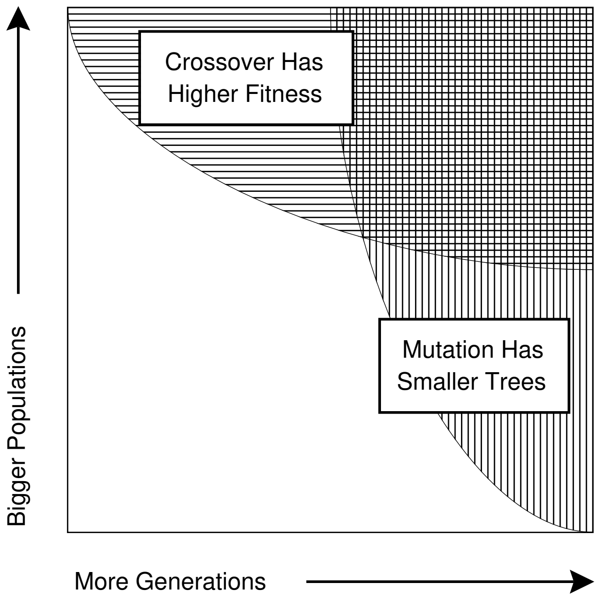



Figure 7.1: Areas where Subtree Crossover has Better Fitness, or where Subtree Mutation has Better Tree Size

Unlike the fitness results, the tree-size result graphs were not symmetric. As might be expected, increasing the number of generations clearly resulted in much larger trees. But it is interesting that tree size also increased with population size, albeit to a lesser degree.

## 7.4 Analysis and Speculation

The fitness and mean tree size results are highly domain-dependent. However, some tenuous trends seem apparent. While both population size and number of generations are important factors, it seems that in general population size more strongly determines whether or not crossover will have better fitness, while number of generations more strongly determines whether or not crossover will have larger trees. Fitness improvement was symmetric about the diagonal, but tree size growth was mostly in the direction of longer run sizes. The domain-specific nature of the results preclude hard-and-fast explanation, but they do lead to some interesting speculation, assuming these trends hold for most domains in general.

### 7.4.1 Getting Your Money's Worth

In determining the best choice of evolutionary run parameters, one must consider not only the expected fitness at the end of the run, but also how long the run will take and how much of the resources it will consume. If a new breeding strategy produces only slightly fitter individuals on average than an alternative approach does, but in the process creates trees twice as large due to bloat, it is probably worthwhile to use the alternative simply because it gives you the opportunity of trying two evolutionary runs in the same amount of time. In this sense, the quality of a breeding operator or other parameter setting is a function of both fitness and of tree size.

Figure 7.1 shows possible, but tenuous, trends in individual quality. Horizontal hatches show where crossover in general produces better-fit individuals; elsewhere mutation is better or the difference is statistically insignificant. Likewise, vertical hatches show where mutation produces smaller trees; elsewhere crossover is better or the difference is statistically insignificant. If these trends hold for domains in general, this suggests an

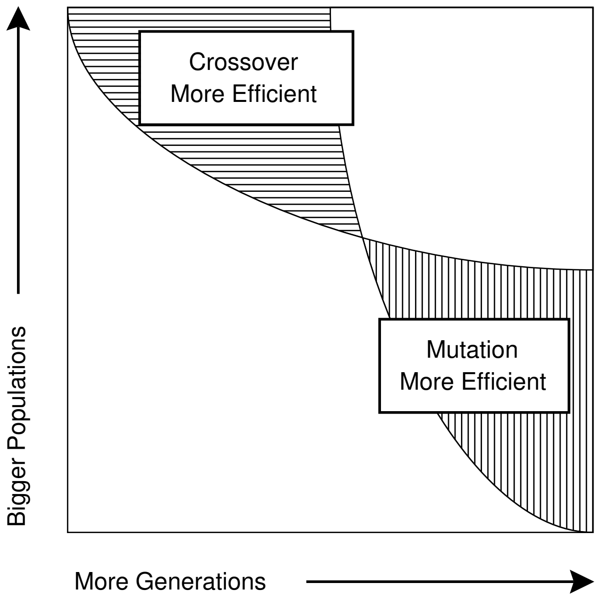



Figure 7.2: Areas where Subtree Crossover or Subtree Mutation is More Efficient

interesting conclusion, shown in Figure 7.2: crossover is a more efficient approach for large populations with small run lengths, whereas mutation is more efficient for small populations with long run lengths.

### 7.4.2 Why Crossover Isn't Always Better

The common wisdom of the GP community has favored a pure-crossover approach, under the presumption that, as in GA, genetic programming individuals transfer "things of value" (building blocks) through subtree crossover. However, this experiment shows that mutation often provides better fitness, smaller trees, or both. Why doesn't subtree crossover perform as advertised? Here is one speculation.

Genetic programming individuals are computer programs, and because of this they suffer greatly from the "linkage problem" discussed in the GA literature (for example, [Mitchell 1996] p. 159). That is, tree nodes at work in GP programs are not independent of each other but instead are usually closely linked (as in any algorithm) through functional, control, and data *dependencies*:

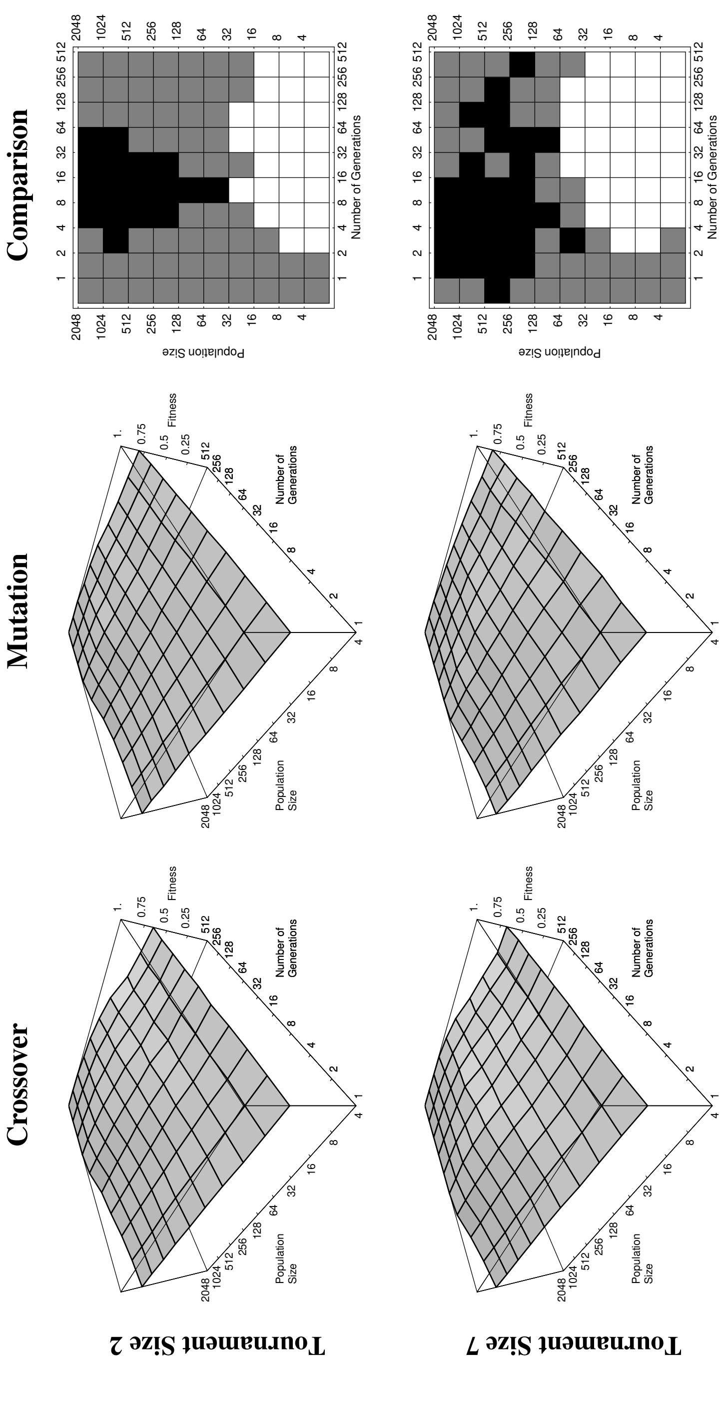
- A *functional dependency* exists between a child and its parent when the data passed from one to the other affects the operation or result of either. Changing a child could dramatically alter the parent's operation (and vice versa).
- A *control dependency* exists when changing a subtree changes the flow of control in the program, affecting whether or not a sibling subtree is executed at all.
- A *data* or *domain dependency* exists when functions and terminals write to a shared memory mechanism like indexed memory [Teller 1994a], or take turns manipulating a global environment in an explicit order. Changing a function or terminal can have a dramatic effect on the operations of other functions and terminals not only within its own subtree but *throughout an individual*.

Crossover between individuals does not just break a relationship between a subtree and its parent node; it also can break many global dependencies between nodes in the tree, and it can modify or introduce completely different global dependencies in the second individual when this subtree is added. Depending on the domain, crossover can change the operation of the individual in dramatic ways not limited to the local area surrounding the crossover point itself.

One speculation is that the noise caused by breaking and creating the dependencies inherent in Genetic Programming may be drowning out much of benefit crossover ordinarily would provide in terms of transferring "value" from one individual to another. Depending on domain, a crossed-over subtree may be so dependent on global dependencies for its operation that its introduction into a new individual dramatically changes its previous local effect. In the worst case, where a domain and function set tends to create complex webs of global dependencies, crossover and mutation may be both be little more than randomization operators.

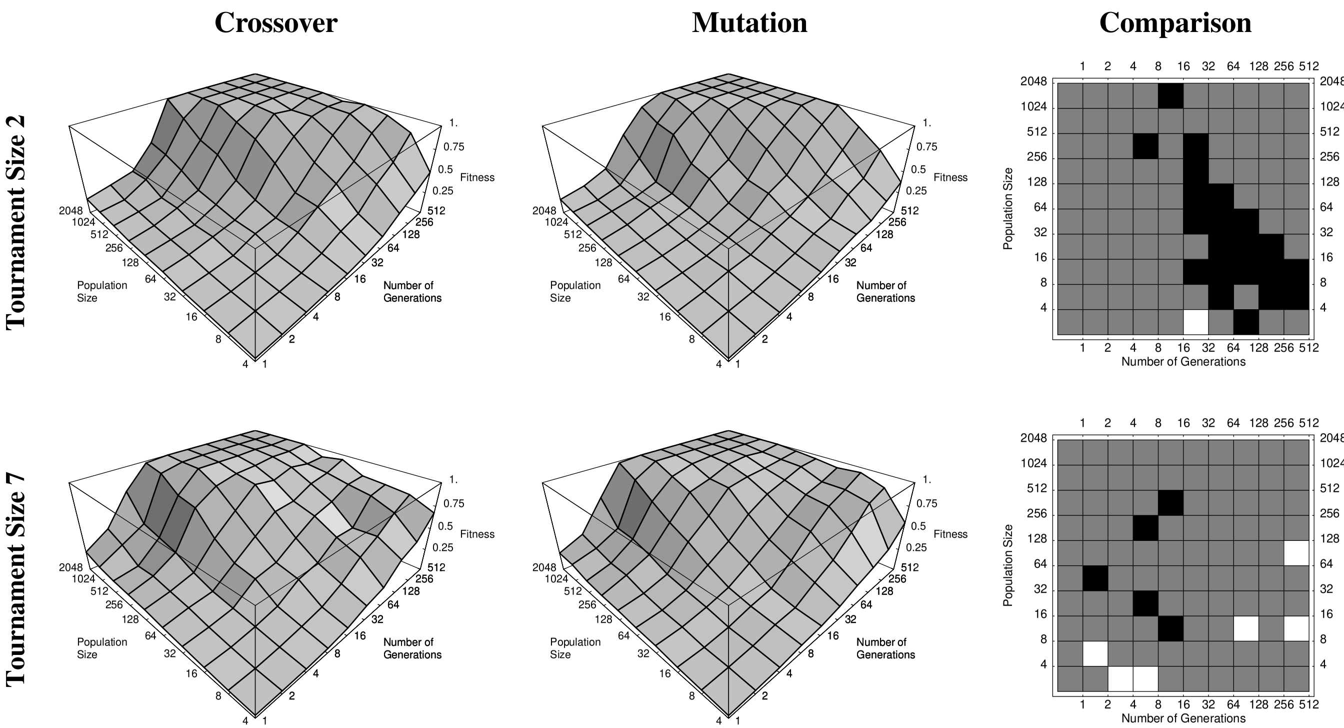
## 7.5 Summary

Picking the right breeding operator and parameter settings is important to finding the best individuals with the least bloat (hence the least waste of time and space). But discovering the best solution with the minimum resource consumption is a nontrivial problem. Through an extensive comparison of subtree mutation and subtree crossover, this chapter has shown that the solution is to some degree dependent on the problem domain and parameter combinations being tried. Nonetheless the trend seems to suggest that subtree crossover will achieve fitter, smaller individuals by using large populations and short run-lengths, while the opposite is true for subtree mutation. While the "big picture" is fraught with domain specificity, these results suggest good ways to allocate resources to optimize GP in light of the bloating issue.



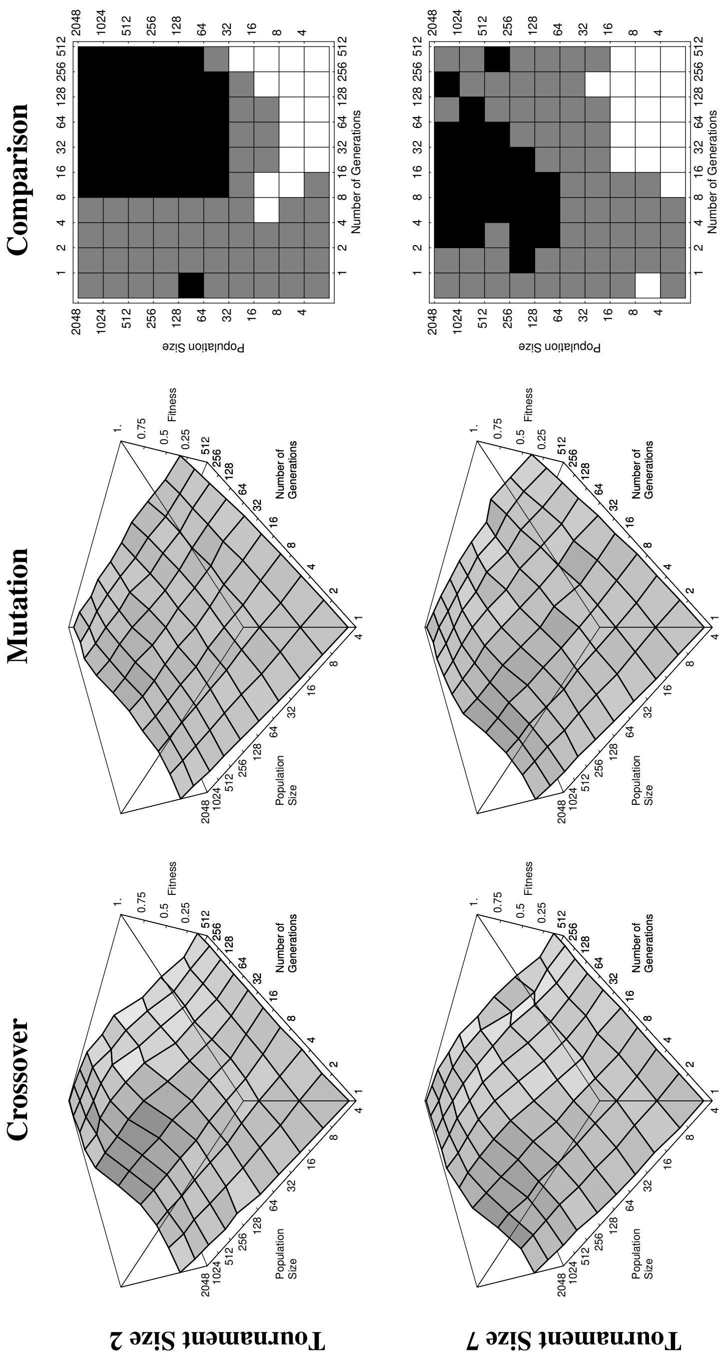

**Legend.** Graph points show the best adjusted fitness of any individual in a run. **Higher** fitness values are better. Comparison graphs are white where mutation is higher than crossover, black where crossover is higher than mutation, and gray where the difference is statistically insignificant.

Figure 7.3: Fitness Results for the 6-Multiplexer Domain



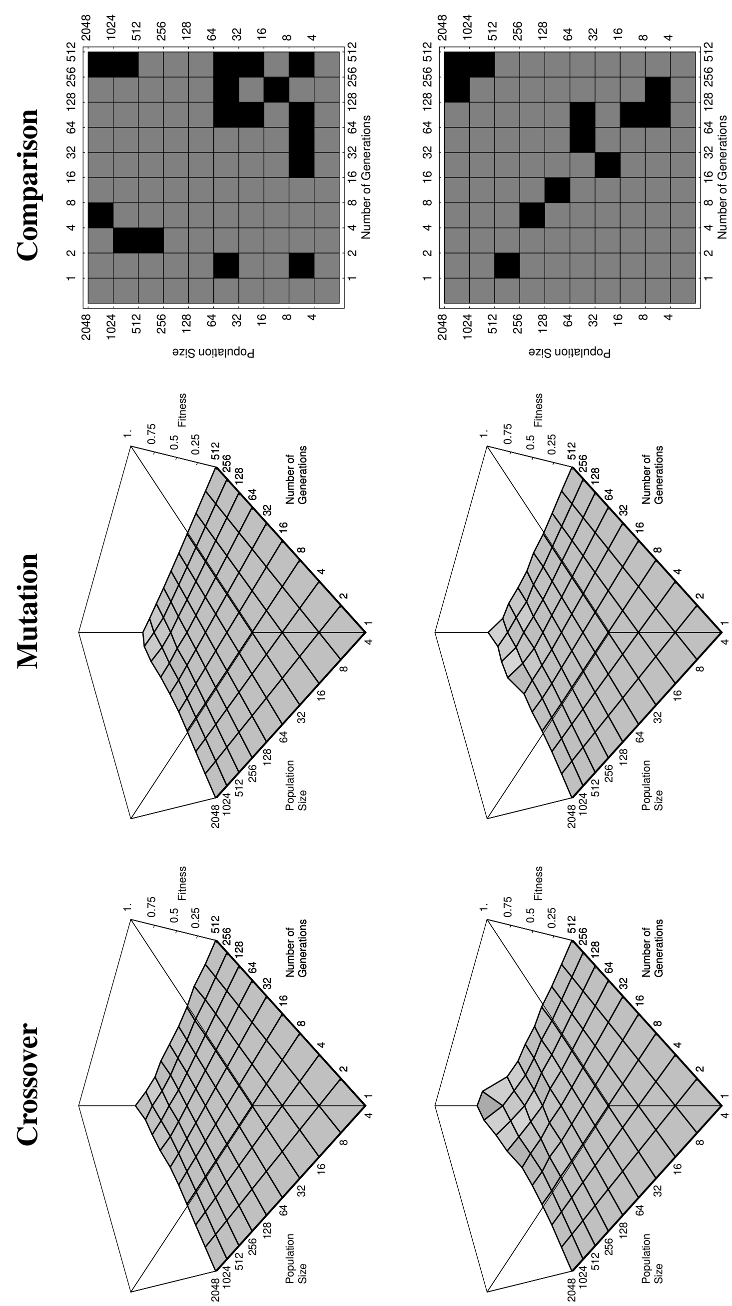

**Legend.** Graph points show the best adjusted fitness of any individual in a run. **Higher** fitness values are better. Comparison graphs are white where mutation is higher than crossover, black where crossover is higher than mutation, and gray where the difference is statistically insignificant.

Figure 7.4: Fitness Results for the Lawnmower Domain



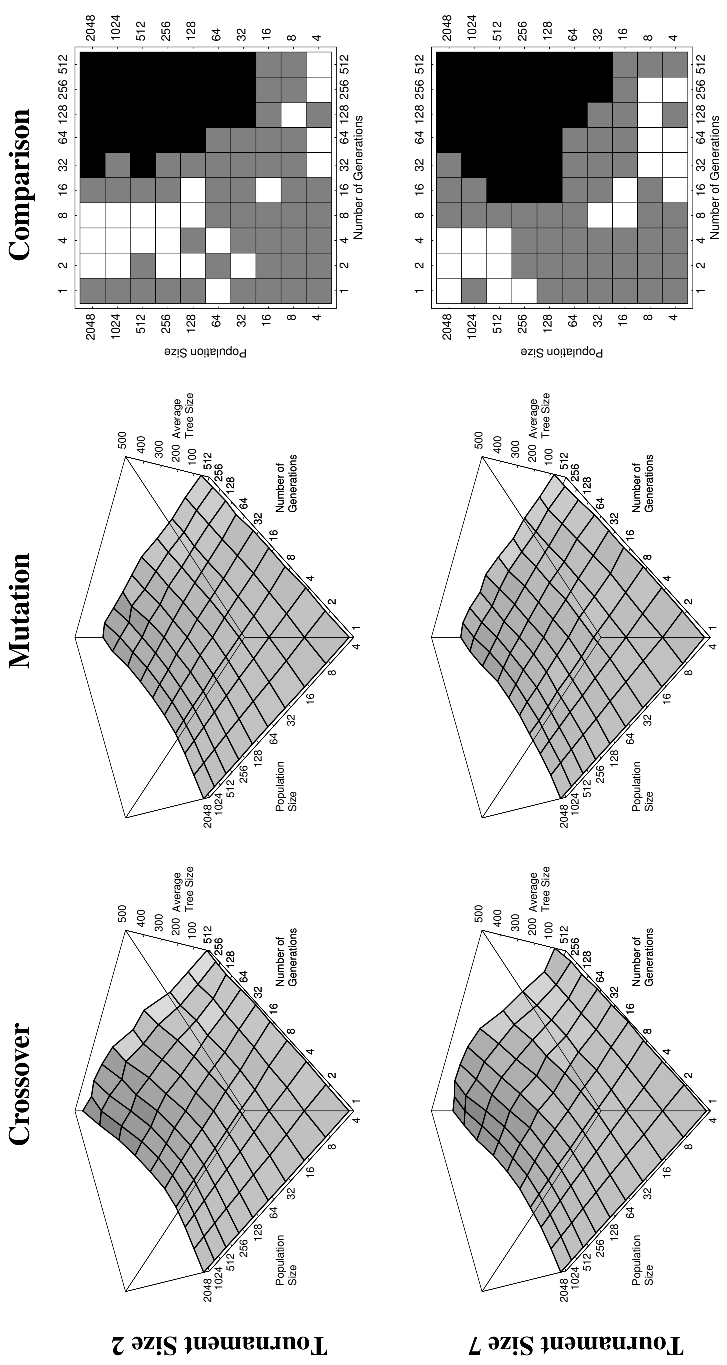

**Legend.** Graph points show the best adjusted fitness of any individual in a run. **Higher** fitness values are better. Comparison graphs are white where mutation is higher than crossover, black where crossover is higher than mutation, and gray where the difference is statistically insignificant.

Figure 7.5: Fitness Results for the Symbolic Regression Domain



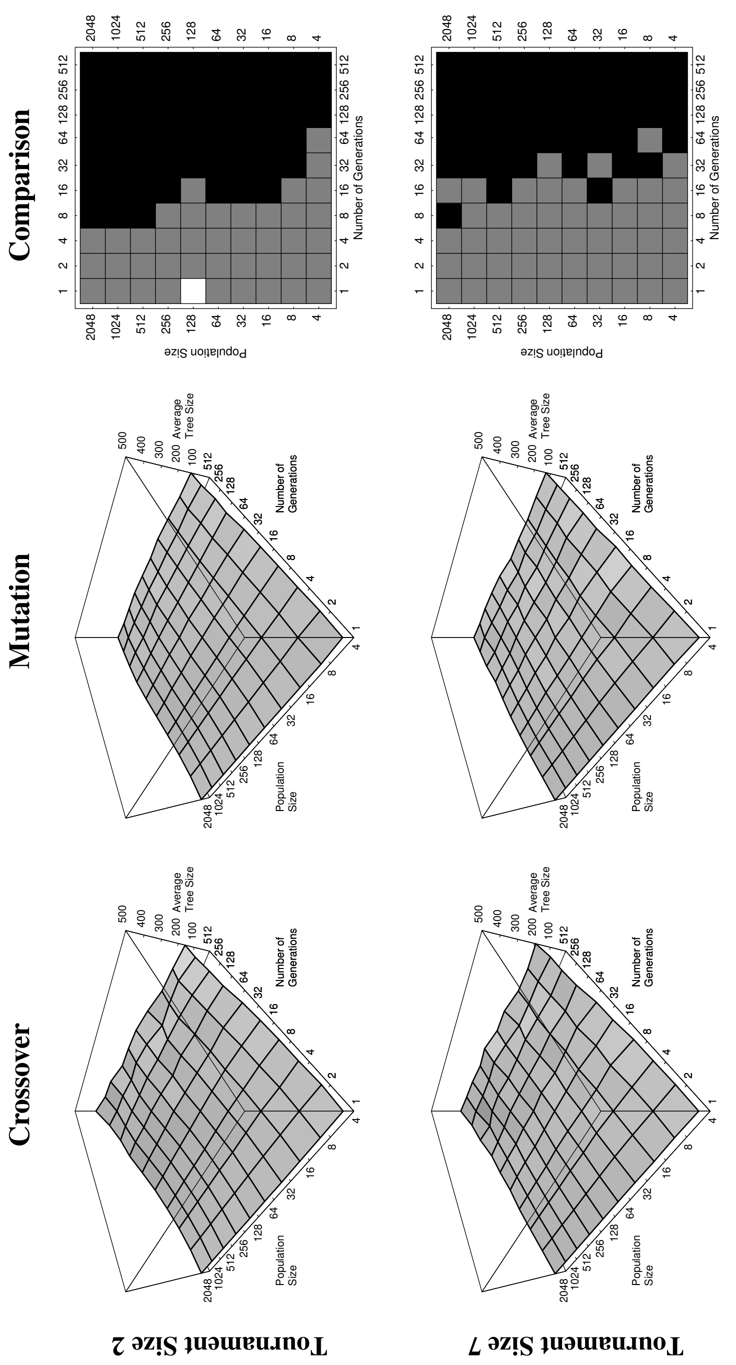

**Legend.** Graph points show the best adjusted fitness of any individual in a run. **Higher** fitness values are better. Comparison graphs are white where mutation is higher than crossover, black where crossover is higher than mutation, and gray where the difference is statistically insignificant.

Figure 7.6: Fitness Results for the Artificial Ant Domain



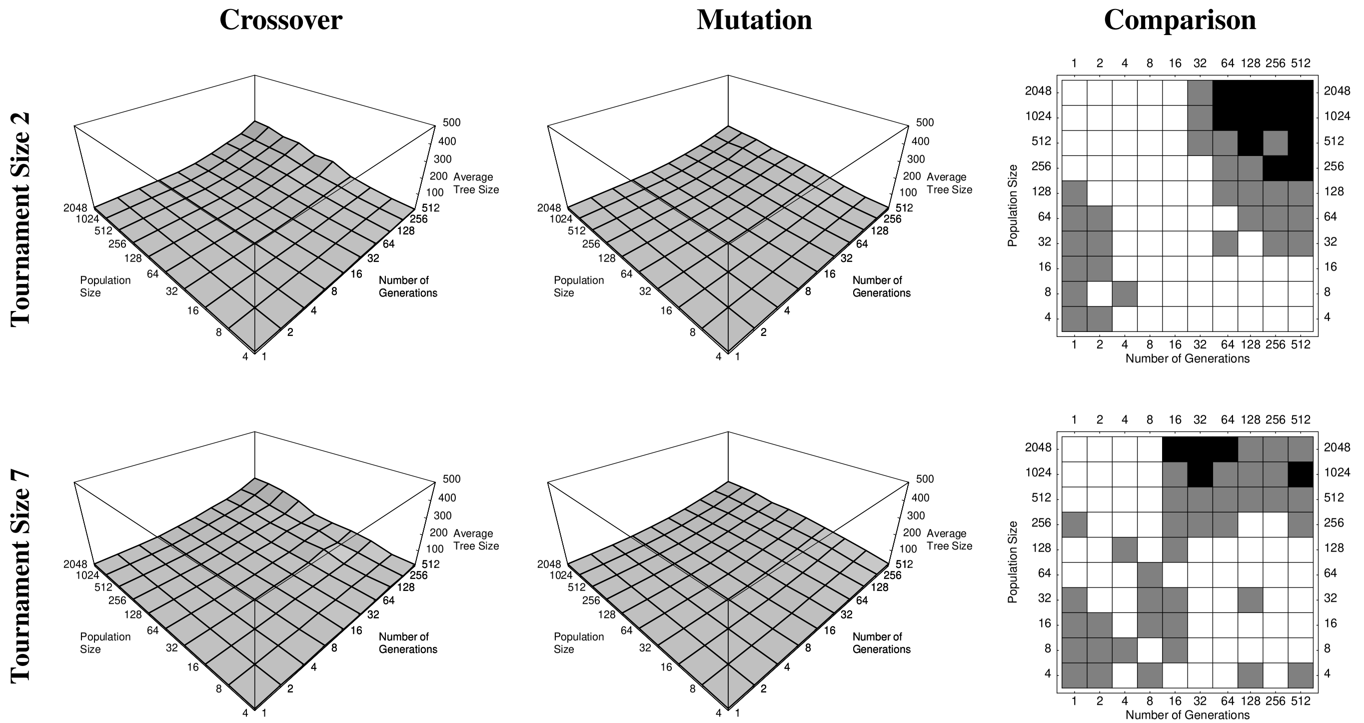

**Legend.** Graph points show the average tree size of individuals in a run. **Lower** fitness values are better. Comparison graphs are white where mutation has bigger trees than crossover, black where crossover has bigger trees than mutation, and gray where the difference is statistically insignificant. **Note:** these comparison graphs' meaning of "better" is the reverse of the fitness comparisons above.

Figure 7.7: Tree Size Results for the 6-Multiplexer Domain



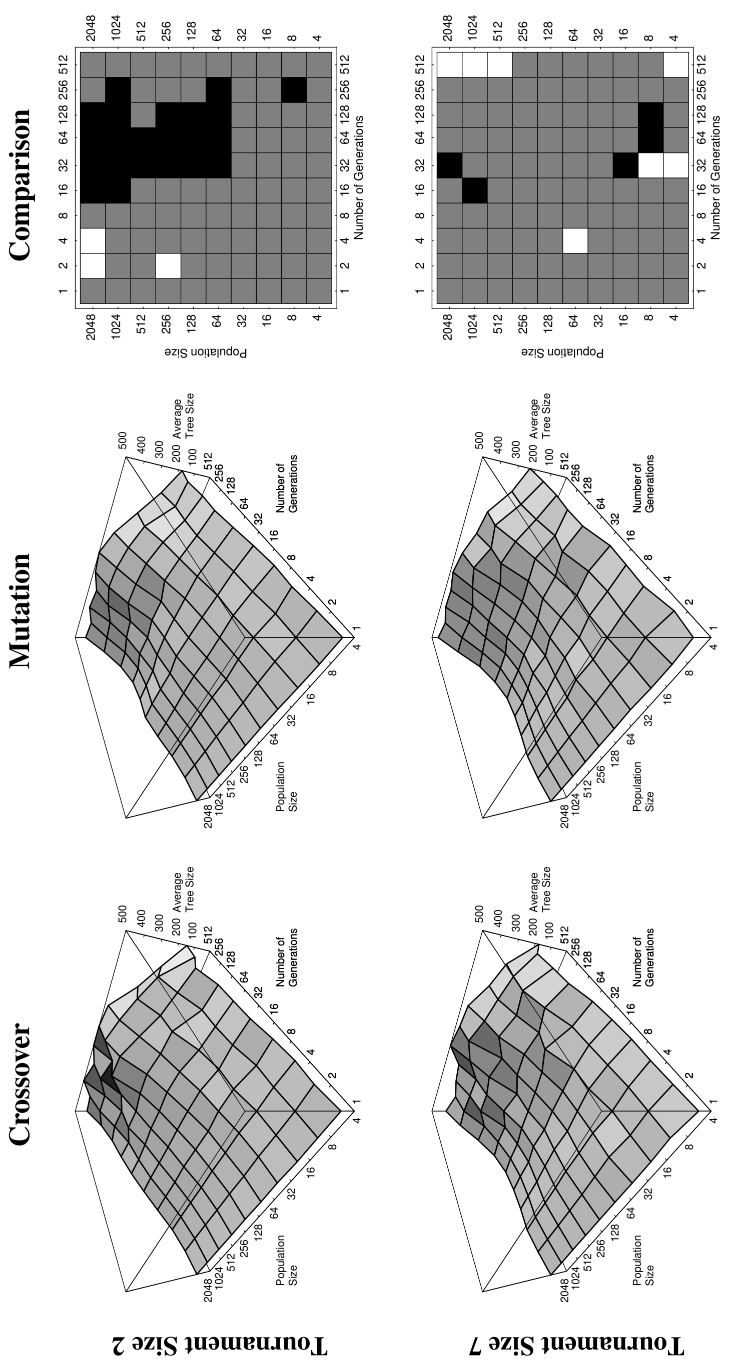

**Legend.** Graph points show the average tree size of individuals in a run. **Lower** fitness values are better. Comparison graphs are white where mutation has bigger trees than crossover, black where crossover has bigger trees than mutation, and gray where the difference is statistically insignificant. **Note:** these comparison graphs' meaning of "better" is the reverse of the fitness comparisons above.

Figure 7.8: Tree Size Results for the Lawnmower Domain




**Legend.** Graph points show the average tree size of individuals in a run. **Lower** fitness values are better. Comparison graphs are white where mutation has bigger trees than crossover, black where crossover has bigger trees than mutation, and gray where the difference is statistically insignificant. **Note:** these comparison graphs' meaning of "better" is the reverse of the fitness comparisons above.

Figure 7.9: Tree Size Results for the Symbolic Regression Domain




**Legend.** Graph points show the average tree size of individuals in a run. **Lower** fitness values are better. Comparison graphs are white where mutation has bigger trees than crossover, black where crossover has bigger trees than mutation, and gray where the difference is statistically insignificant. **Note:** these comparison graphs' meaning of "better" is the reverse of the fitness comparisons above.

Figure 7.10: Tree Size Results for the Artificial Ant Domain

## Chapter 8

# Tree Generation

GP needs a good random tree-creation algorithm to generate the trees that form the initial population, and to make new subtrees used in subtree mutation. This helps GP search the space of program trees in a more uniform fashion. But poor choices of tree creation algorithms can also have unforeseen bloating consequences during subtree mutation. Depending on the function sets and tree-creation algorithms used, subtree mutation will cause average tree size to asymptotically approach some value. This value varies — in some problems, it may be very small and actually act as a deterrent to tree growth. But in many others it can act to increase tree growth rapidly. This is not the chief source of bloat: Chapter 9 discusses the primary bloat causes. But its effect is one which has so far been entirely unrealized, possibly because mutation has only recently been studied significantly in the community. And it is also an interesting effect because it is integral to the algorithm, that is, it will happen whether or not any selection is occurring.

This effect can be countered, for example, by creating subtrees whose exact or expected tree size is the same size as the removed subtree [Chellapilla 1997]. But in order to try these techniques it is generally important first to be able to rigorously specify the expected tree size generated by the tree-creation algorithm used. In this regard the popular GROW algorithm [Koza 1992] is an especially poor choice. GROW by itself offers no user control over expected tree size at all; one must rely on solely on an absolute maximum depth bound. In fact, left to its own devices, GROW will generate infinite trees on average for very common function sets.

GROW and FULL are also lacking in other important areas. Neither offers fine-grained control over either the tree structure or over the expected probability that a particular function will appear in a tree; one is left to rely on the natural interplay of the algorithm and the function set provided.

This chapter discusses GROW and its variants. It presents common examples where GROW unexpectedly produces very small or infinite-sized trees on average. It then proves that GROW can cause tree size to bloat independent of selection, and presents experimental support of such bloating effects in the Symbolic Regression domain, even when the tree generation is depth-limited.

The chapter then discusses other algorithms which address the lack of tree-size control in GROW. Though these algorithms provide uniform tree distribution, they are computationally inefficient. Two new algorithms are then proposed which attempt to address deficiencies in GROW and other alternatives. These two new algorithms let the user request either an average tree size or specify a distribution of tree sizes from which to generate trees. Unlike recent alternatives, these two algorithms do not promise uniformly random tree structures. Instead they guarantee user-specified probabilities of occurrence for specific terminal and nonterminal functions within the generated trees. These two algorithms are also fast, running in near-linear time, or in linear time under reasonable constraints. The chapter analyzes the new algorithms, then gives additional versions of the algorithms tailored for "strongly-typed" genetic programming [Montana 1995].

## 8.1 Definitions

$\mathbf{T}$ denotes a newly generated genetic program tree. $S$ is the maximal number of nodes in a tree permitted by a given tree-generation algorithm, and $D$ is the maximal depth of a tree permitted by the algorithm. $E_{tree}$ is the

| Domain | $b$ | $p$ | $E_{tree}$ |
|---|---|---|---|
| Cart Centering | $\frac{11}{6}$ | $\frac{3}{4}$ | $\infty$ |
| Ant | $\frac{7}{3}$ | $\frac{1}{2}$ | $\infty$ |
| Regression | $\frac{3}{2}$ | $\frac{8}{9}$ | $\infty$ |
| 11-Multiplexer | 2 | $\frac{4}{15}$ | $\frac{15}{7}$ |
| 6-Multiplexer | 2 | $\frac{2}{5}$ | 5 |
| 3-Multiplexer | $\frac{5}{3}$ | $\frac{1}{2}$ | 6 |

**Legend.** $b$ is the expected number of children of a nonterminal picked from the function set. $p$ is the probability of choosing a nonterminal at tree-creation time. $E_{tree}$ is the expected size of a tree. From Theorem 2, this is $\frac{1}{1-pb}$ if $pb < 1$, $\infty$ otherwise.

Table 8.1: Algorithmic Results for the Introductory Domains, Using the GROW Algorithm

expected tree size (number of nodes) of $\mathbf{T}$. $\mathbf{T}$ is created by choosing nodes from a function set $F$, divided two disjoint sets, terminals $T$ and nonterminals $N$.

$p$ is the probability that, in the process of choosing nodes to form $\mathbf{T}$, a given tree-creation algorithm will pick a nonterminal from the function set (as opposed to a terminal). $b$ is the expected number of children to a nonterminal node picked at random from the function set. $g$ is the expected number of children to a newly generated node in $\mathbf{T}$. It is important to note that since the expected number of children of a terminal is 0, and the expected number of children to a nonterminal is $p$, then $g = pb + (1-p)(0) = pb$.

## 8.2 The GROW Algorithm

The GROW tree-creation algorithm was described in Section 3. GROW is by far the most common mechanism for creating trees and subtrees in genetic programming. Though little studied, GROW and FULL appear in the lion's share of existent GP literature and are the chief or only tree-creation algorithms for nearly all popular GP libraries (including lil-gp, GPSys, GPQuick/GPData, GPC++, DGPC, SGPC, and Koza's Simple-LISP code [Koza 1992]). GROW and a full-tree variant (FULL), are used to generate trees for the initial population at the beginning of the evolutionary process. GROW is also used almost universally to generate new subtrees for subtree mutation and is very popular for other special kinds of subtree mutation. For some examples of subtree mutation and other mutation approaches, see [Koza 1992; Angeline 1998; Chellapilla 1997; Langdon and Poli 1997c].

While GROW is easy to implement and runs in linear time, it has three weaknesses. First, the algorithm picks all functions with equal likelihood; there is no way to fine tune the preference of certain functions over others. Second, the algorithm does not give the user much control over the tree structures generated. Third, and most significant, while $D$ (or $S$) is used as an upper bound on maximal tree depth, there is no appropriate way to create trees with either a fixed or average tree size or depth.

The lack of a rigorous way to specify tree size is very problematic: in most GP literature, the sets of nonterminals and terminals are picked based on domain need, with little consideration given to their effect on tree generation. The expected tree size $E_{tree}$ under the traditional algorithm is determined solely by $g$, the expected number of children of a newly generated node. In particular, if $g < 1$, then $E_{tree} = \frac{1}{1-g}$, else $E_{tree}$ is infinite, as shown in Theorem 2.

**Lemma 1** Assume an algorithm that "grows" a tree from a randomly chosen root node by attaching randomly generated child nodes into unfilled argument positions of nodes currently in the tree. This is done for example in GROW. Let $g \geq 0$ be the expected number of children of a newly generated node. Then at depth $d$ in the tree, the expected number of nodes is $E_d = g^d$.

**Proof** The expected number of nodes $E_d$ at depth $d$ is

$$E_d = \begin{cases} 1 & \text{if } d = 0 \\ gE_{d-1} & \text{if } d > 0 \end{cases}$$

From this it follows that $E_d = g^d$. ■

**Theorem 2** As in Lemma 1, assume an algorithm which "grows" a tree from a randomly chosen root node by attaching randomly generated child nodes into unfilled argument positions of nodes currently in the tree. Let $E_{tree}$ be the expected size of a tree built with the algorithm. Let $g \geq 0$ be the expected number of children of a node newly generated by the algorithm. If $g < 1$, then $E_{tree}$ is finite ($E_{tree} = \frac{1}{1-g}$), else it is infinite.

**Proof** The expected size of a tree is the sum of the expected number of nodes at all levels, that is, $E_{tree} = \sum_{d=0}^{\infty} E_d$. From Lemma 1, $E_{tree} = \sum_{d=0}^{\infty} g^d$. From the geometric series, for $g \geq 0$,

$$\sum_{d=0}^{\infty} g^d = \begin{cases} \frac{1}{1-g} & \text{if } g < 1 \\ \infty & \text{if } g \geq 1 \end{cases}$$

■

For this reason, poor function set choices can have a dramatic unforeseen effect on tree creation. Consider the following example: imagine a typical domain that has 5 terminals and 5 nonterminals, where the average number of children of a nonterminal is 2. In this case, $g = 1$, and so the expected tree size is infinite! Although the complexity of GROW is linear in the size of the tree, this doesn't say much in the face of infinite tree sizes. As such, the worst-case time bound for GROW is in fact dependent entirely on the choice of functions in the function set.

Tweaking the function set to come up with a combination of terminals and nonterminals that give a reasonable $E_{tree}$ is often difficult; very slight modifications in a function set can result in an $E_{tree}$ that is either very small (say, less than 2) or infinite. As a result, many common published function sets inadvertently have either very small or very large, possibly infinite, expected tree sizes. For example, Table 8.1 shows that three of the four introductory examples in [Koza 1992] have an infinite expected tree size (cart centering, regression, ant). The fourth (11-multiplexer) has an average tree size of about 2.

The genetic programming code described in [Koza 1992] uses several ad-hoc methods to compensate for the tree size resulting from GROW. First, it imposes a small maximal depth $D$ (from 2 to 6) on trees generated for the initial population. Oddly, the maximal depth is not chosen at random from this range, but in a round-robin fashion. $D$ is set to 5 when GROW is called on to create subtrees for subtree mutation. Because GROW so often creates infinite-sized trees, this maximal depth limit "shaves" function trees to keep the initial population size reasonable, resulting in an excessive number of full or near-full trees. Second, it rejects all trees of depth 1, and eliminates duplicate trees, which increases average tree size. Third, when creating initial trees, a mixture of GROW and FULL is used, but when creating subtrees for modifying trees through subtree mutation, only GROW is used. Newly mutated trees are rejected if they exceed an absolute depth limit (typically 17).

## 8.3 Subtree Mutation and Code Growth

If there is no selection bias, repeated applications of subtree crossover cannot at the limit increase tree size, because the subtree crossover operation neither adds nor subtracts nodes from the population. But this is not the case for subtree mutation, which will move the average population size towards some asymptotic value. The actual value varies, depending on the way mutation points are chosen within the individual and the expected size of subtrees generated by the tree-creation algorithm. Unfortunately, when GROW is used, these variables are highly function-set dependent. Complex function sets can cause subtree mutation to bloat populations significantly even with no selection bias.

Consider first a simple example of GROW without a depth bound $D$ in force, but with a small expected tree size $E_{tree} = 2$. In this example, the function set will consist of one nonterminal of arity 1, and one terminal, both equally likely to appear. The resultant "lists" have an $E_{tree}$ of 2, and so the initial population is on average of size 2. But after applying subtree mutation this initially-generated population, the expected size of individuals in the population grows to 2.5 (proven in Theorem 3 below).

**Lemma 2** If $-1 < x < 1$, then $\sum_{i=1}^{\infty} ix^i = \frac{x}{(1-x)^2}$.

**Proof**

$$\begin{aligned} \sum_{i=1}^{\infty} ix^i &= x + 2x^2 + 3x^3 + \ldots \\ x\sum_{i=1}^{\infty} ix^i &= \phantom{x + {}} x^2 + 2x^3 + \ldots \end{aligned}$$

Therefore,

$$(1-x)\sum_{i=1}^{\infty} ix^i = x + x^2 + x^3 + \ldots = \sum_{i=1}^{\infty} x^i = \left(\sum_{i=0}^{\infty} x^i\right) - 1$$

From the Geometric Series, if $-1 < x < 1$, then

$$(1-x)\sum_{i=1}^{\infty} ix^i = \frac{1}{1-x} - 1 = \frac{x}{1-x}$$

$$\sum_{i=1}^{\infty} ix^i = \frac{x}{(1-x)^2}$$

■

**Theorem 3** Let the set of functions $F$ consist of one terminal and one nonterminal of arity 1 with $p = \frac{1}{2}$ (e.g., functions which build "lists"). The natural expected tree size of trees built from this set is $E_{tree} = 2$; however, after a subjecting the individuals in the initial generation to subtree mutation, the expected tree size rises (to $\frac{5}{2}$).

**Proof** Since there are only two functions (a terminal with 0 children and a nonterminal with 1 child) with equal likelihood of being generated, then the expected number of children of a newly generated node is $g = \frac{1}{2}$. From Theorem 2, this means that $E_{tree} = 2$. Since $p = \frac{1}{2}$, a "list" of node size 1 should clearly occur in $\frac{1}{2}$ of all cases, node size 2 should occur in $\frac{1}{4}$ of all cases, node size 3 has a probability of occurrence $\frac{1}{8}$, and so on. The probability that a "list" of node size $n$ will occur in a population is $\frac{1}{2^n}$.

The expected new size of an individual of size $n$ after undergoing subtree mutation is the original size minus the expected loss ($\frac{n+1}{2}$ for a list) plus the expected size of the new subtree. For a list this comes to $n - \frac{n+1}{2} + 2$. Hence the expected size of individuals in a population after subjecting each to subtree mutation is

$$\sum_{n=1}^{\infty} \frac{1}{2^n}(n - \frac{n+1}{2} + 2) = \left(\frac{1}{2}\sum_{n=1}^{\infty} \frac{n}{2^n}\right) + \left(\frac{3}{2}\sum_{n=1}^{\infty} \frac{1}{2^n}\right)$$

From Lemma 2, the first term reduces to 1. From the Geometric Series, the second term reduces to $\frac{3}{2}$. Therefore the new expected size of an individual is $\frac{5}{2}$. ■

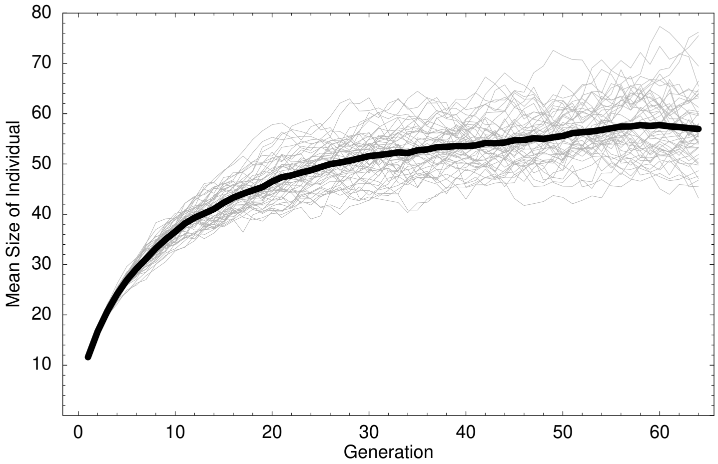



Figure 8.1: Asymptotic Bloating Characteristic of Subtree Mutation in the Symbolic Regression Domain, with Random Selection

Interestingly, if $E_{tree}$ for newly mutated subtrees is restricted to 1.5, this will maintain $E_{tree} = 2$ for the population as a whole. This is a provable example of growth under subtree mutation, but the expected tree size and function set used is unlikely in real problems. Unfortunately, while closed-form proofs may be derivable for certain restricted function sets (lists, for example), I do not believe that there exists a similar proof for trees in general, given arbitrary function. A Koza-style depth limit further complicates proof of expected subtree size by "shaving" trees in an ad-hoc fashion. For these more realistic function sets one must rely on empirical estimation. For example, Figure 8.1 shows the mean tree sizes of fifty Symbolic Regression runs with no ERCs and 100% subtree mutation, and in which trees were selected entirely at random. A subtree-generation depth bound $D$ of 5 is in force, but as can be seen, average tree size still rises from about 10 (the expected size of initially-generated trees) to about 60.

## 8.4 Previous GP Tree-Creation Algorithms

Previous improvements to GROW and FULL have focused on generating uniformly random tree structures of predetermined sizes.

Iba's RAND_tree algorithm [Iba 1996b] generates uniform tree structures by using Dyck words to build trees bottom-up. RAND_tree builds trees from a fixed-size pool of tree nodes, joining nodes together to form subtrees, and ultimately joining subtrees together to form the final tree. RAND_tree makes certain that each node in the tree has an arity selected from a user-supplied arity set (for example, all nonterminals might have either 2, 3, or 5 children). This arity constraint puts RAND_tree in conflict with more restrictive forms of GP (such as "strongly-typed" GP [Montana 1995], where each function has a specific return type and distinct argument types). To use strongly-typed GP with RAND_tree, the user must create a function set with all permutations of both the arity set and return types, else the algorithm will generate invalid tree structures.

Other approaches have tried production grammars [Whigham 1996; Geyer-Schulz 1995]. Significantly, [Bohm and Geyer-Schulz 1996] extend this approach by selecting trees with exact uniform probability from a tree-derivation grammar. Given the absolute maximum bound on tree size $S$, their approach first compiles (off-line) a table $\Pi(W, s)$ of probabilities of producing trees of size $s \leq S$ derived from some symbol $W$. Once this table has been compiled, their tree-generation algorithm first picks a statistically random tree size and start symbol. It then expands this symbol with some random production, using the table to recursively compute appropriate sizes for each subtree that will be derived from the symbols in the expansion. This elegant approach can generate traditional GP or strongly-typed GP trees of any size (up to $S$) from a completely uniform random distribution of tree structures.

The strength of all these approaches is that they permit user control over the size of the trees generated, and generate uniformly-distributed random tree structures. But there are two drawbacks to these approaches: they are combinatorically very slow, and they cannot guarantee user-defined probabilities of appearance of functions within their trees (because this conflicts with generating uniformly-distributed structures).

Iba notes that RAND_tree has very high (in some cases infinite) computational complexity because the tree-structure determination includes producing large Catalan numbers. Böhm and Geyer-Schulz's algorithm has linear complexity once the table $\Pi$ has been compiled, but compiling this table includes effectively enumerating all possible appropriate trees of size $\leq S$. Even with the help of dynamic programming, the complexity of this generation can be very high, though possibly polynomial. Böhm and Geyer-Schulz do not give a worst-case bound for generating this table. The authors note that combinatorics and other issues could make the practical application of the algorithm difficult.

## 8.5 PTC1 and PTC2

This paper offers two alternative tree-creation algorithms, Probabilistic Tree-Creation (PTC) 1 and 2, which take a different approach from past algorithms. Like past algorithms, PTC1 and PTC2 give the user control over generated tree size. However, these new algorithms do not attempt to generate completely uniformly distributed tree structures. Instead, they guarantee what previous approaches cannot: user-defined probabilities of appearance of functions within the tree. But most importantly, PTC1 and PTC2 have very low computational complexity (linear, under reasonable constraints).

PTC1 is a modification of GROW that allows the user to provide probabilities of appearance of functions in the tree, plus a desired *expected tree size*, and guarantees that, on average, trees will be of that size. PTC1 has formal results that have applicability to GROW. However, PTC1 does not give the user any control over the variance in tree sizes generated, which limits its usefulness.

With PTC2, the user provides a probability distribution of requested tree sizes. PTC2 guarantees that, once it has picked a random tree size from this distribution, it will generate and return a tree of that size or slightly larger. This approximation is less precise than PTC1, and PTC2 does not yield the same elegant theoretical results. However, it gives the user real control over tree size variance, a critical advantage.

## 8.6 The PTC1 Algorithm

The PTC1 algorithm is as follows: the set of functions $F$ is divided into two disjoint subsets: nonterminals $N$ and terminals $T$. During tree-generation time, the algorithm will alternately choose new child nodes from either the nonterminals or from the terminals. For each nonterminal $n$ in $N$ the user provides a probability $q_n$ that $n$ will be chosen from $N$ when the algorithm needs a nonterminal. Similarly, for each terminal $t$ in $T$ the user provides a probability $q_t$ that $t$ will be chosen from $T$ when the algorithm needs a terminal. The user also provides a maximum depth bound $D$ as before, though this bound is used only to enforce an absolute, and preferably very large, memory restriction. Lastly, the user indicates an expected tree size $E_{tree}$.

Before generating any trees, the algorithm computes $p$, the probability of choosing a nonterminal over a terminal in order to produce a tree with an expected tree size $E_{tree}$, as

$$p = \frac{1 - \frac{1}{E_{tree}}}{\sum\limits_{n \in N} q_n b_n}$$

where $b_n$ is the arity of nonterminal $n$. This computation need be done only once offline. Then the algorithm proceeds to create the tree:

**Algorithm 4** PTC1

*Given:*
 maximum depth bound $D$
 function set $F$ consisting of nonterminal set $N$ and terminal set $T$
 computed probability of choosing a nonterminal $p$
 probabilities $q_t$ and $q_n$ for each $t \in T$ and $n \in N$
*Do:*
 new tree $\mathbf{T}$ = PTC1$(0)$

*PTC1*(depth $d$)
 *Returns:* a tree of depth $\leq D - d$

 If $d = D$, return a terminal from $T$ (by $q_t$ probabilities)
 Else if, with probability $p$ a nonterminal must be picked,
  Choose a nonterminal $n$ from $N$ (by $q_n$ probabilities)
  For each argument $a$ of $n$,
   Fill $a$ with PTC1$(d + 1)$
  Return $n$ with filled arguments
 Else return a terminal from $T$ (by $q_t$ probabilities)

This algorithm guarantees an expected tree size of $E_{tree}$ for trees and subtrees by determining the appropriate nonterminal-selection probability $p$. In the trivial case where there are *only* terminals in the function set, the algorithm of course cannot provide any $E_{tree}$ other than 1. Additionally, by adjusting nonterminals with large fan-outs to have a lower (or higher) probability of occurrence than nonterminals with small fan-outs, the user can bias the typical "bushiness" of a tree, yet keep $E_{tree}$ the same.

PTC1 attempts to fix the expected tree size $E_{tree}$ yet still provide the user with as much freedom as possible in defining probabilities of appearance for each function. Recognize that $E_{tree}$ can be controlled by fixing $g$, the expected number of children of a newly generated node, shown in Theorem 4 below. Note that since $g = \sum_{f \in F} q_f b_f$, to fix $g$ over some user-defined set $F$ of functions with known arities ($b_f$), the algorithm must somehow adjust the relative appearance ($q_f$) of functions within the set. PTC1 accomplishes this simply by picking the right $p$, the probability that, at node-creation time, a node will be picked from the nonterminal set (as opposed to the terminal set), as shown in Theorem 5 below. Within the respective nonterminal or terminal sets, the user is still free to set his own $q_t$ and $q_n$.

**Theorem 4** For PTC1, assume that $N$, the set of nonterminal functions, is nonempty. Let $p$ be the probability that a newly generated node will be a nonterminal. Let $b$ be the expected number of children of a nonterminal node picked from the function set. Let $g$ be the expected number of children of a newly generated node. Then a $p$ can be predetermined to guarantee any specific $E_{tree} \geq 0$.

**Proof** Since $N$ is nonempty, therefore $b > 0$. Since $g = pb$, given a constant, nonzero $b$, a $p$ can clearly be picked to produce any desired $g$. From Theorem 2, a $g$ (and hence a $p$) can thus be picked to determine any finite or infinite $E_{tree} \geq 0$. ■

**Theorem 5** For PTC1, assume that $N$, the set of nonterminal functions, is nonempty. For each $n \in N$, let $b_n$ be the arity of $n$, and let $q_n$ be the probability that a newly generated nonterminal will be $n$. Let $p$ be the probability that a newly generated node will be a nonterminal. Then the nonterminal-choice probability $p$ necessary to guarantee that a tree $\mathbf{T}$ will be of expected finite size $E_{tree} > 0$ is $p = \frac{1 - \frac{1}{E_{tree}}}{\sum_{n \in N} q_n b_n}$.

**Proof** As a consequence of Theorems 2 and 4, $E_{tree} = \frac{1}{1-pb}$, where $b$ is the expected number of children of a nonterminal node picked from the function set. Because $N$ is nonempty, $b > 0$, hence $p$ may be determined as $p = \frac{1-\frac{1}{E_{tree}}}{b}$. Since $b = \sum_{n \in N} q_n b_n$, thus $p = \frac{1-\frac{1}{E_{tree}}}{\sum_{n \in N} q_n b_n}$. ■

### 8.6.1 Complexity of PTC1

The time bound for PTC1 is determined by the complexity $c_T$ of choosing a random terminal from some probability distribution of terminals, and the complexity $c_N$ of choosing a random nonterminal from a probability distribution of nonterminals. From Theorem 6 below, the number of nonterminals in a tree is $pE_{tree}$ and the expected number of terminals is $(1-p)E_{tree}$, hence the complexity of generating a full tree of terminals and nonterminals is $O(c_N p E_{tree} + c_T(1-p)E_{tree})$.

**Theorem 6** For PTC1, let $E_{tree}$ be the expected tree size, and $p$ be the precomputed nonterminal-selection probability to generate a tree of expected size $E_{tree}$. Then the expected number of nonterminals in a tree is $E_{n,tree} = pE_{tree}$ and the expected number of terminals is $E_{t,tree} = (1-p)E_{tree}$.

**Proof** Let $g$ be the expected number of children of a newly generated node under PTC1. Then the expected number of nonterminals $E_{n,d}$ at depth $d$ is

$$E_{n,d} = \begin{cases} p & \text{if } d = 0 \\ E_{n,d-1}bp & \text{if } d > 0 \end{cases}$$

That is, $E_{n,d}$ is equal to the expected number of nonterminals $E_{n,d-1}$ at depth $d-1$, times the expected number of children $b$ to each of these nonterminals, times the probability $p$ that these children are nonterminals themselves.

From this it follows that $E_{n,d} = p(bp)^d = pg^d$. Since by Lemma 1, the expected number of nodes $E_d = g^d$, then the number of terminals is $E_{t,d} = E_d - E_{n,d} = g^d - pg^d = (1-p)g^d$. Hence for the whole tree, the expected number of nonterminals in a tree is then $E_{n,tree} = \sum_{d=0}^{\infty} E_{n,d} = p\sum_{d=0}^{\infty} g^d$ and the expected number of terminals is $E_{t,tree} = \sum_{d=0}^{\infty} E_{t,d} = (1-p)\sum_{d=0}^{\infty} g^d$. From Theorem 2 we get $E_{n,tree} = pE_{tree}$ and $E_{t,tree} = (1-p)E_{tree}$. ■

Note that $c_N$ or $c_T$ is at most the complexity of a binary search through some probability distribution. To achieve this for the nonterminal probabilities, for example, arrange all the $q_n$ into disjoint intervals from 0 to 1 corresponding to each $n \in N$:

$$\begin{array}{ll} n_1: & [0\ ,\ q_{n_1}) \\ n_2: & [q_{n_1}\ ,\ q_{n_1} + q_{n_2}) \\ n_3: & [q_{n_1} + q_{n_2}\ ,\ q_{n_1} + q_{n_2} + q_{n_3}) \\ \dots & \dots \\ n_{|N|}: & [1 - q_{n_{|N|}}\ ,\ 1] \end{array}$$

A random event points somewhere into this set of intervals; an $O(\lg(\|N\|))$ binary search through this set finds which $n$ corresponds to the random event. This can be done similarly for terminal probabilities $(q_t)$, hence averaged over several iterations, an upper complexity bound on PTC1 is

$$\begin{aligned} & O(\lg(\|N\|)pE_{tree} + \lg(\|T\|)(1-p)E_{tree}) \\ & \leq O(\lg(\|F\|)E_{tree}) \end{aligned}$$

If $c_T$ and $c_N$ were constant, then the complexity would reduce to $O(E_{tree})$. This might happen if all the terminals and nonterminals had equal $q$ probabilities, in which case selecting a random terminal or nonterminal can be done with a simple $O(1)$ random event as in GROW. An $O(E_{tree})$ complexity can also be achieved if the $q_t$ and $q_n$ probabilities are discrete values. For example, imagine that there were three nonterminals with probabilities $\{q_1 = .2, q_2 = .3, q_3 = .5\}$. One can create an array $[n_1, n_1, n_2, n_2, n_2, n_3, n_3, n_3, n_3, n_3]$. At nonterminal-selection time, picking randomly from this array is $O(1)$.

### 8.6.2 Strongly-Typed PTC1

Under relaxed constraints, PTC1 can be extended easily to handle the "basic" form of strongly-typed genetic programming (STGP) [Montana 1995]. StronglyTypedPTC1 assumes that for each type, there exists at least one nonterminal and at least one terminal whose return values are of that type. The algorithm presented is for atomic STGP. In order to accommodate STGP, StronglyTypedPTC1 must place further constraints on user-specified probabilities, by dividing the set of functions $F$ into not just terminals and nonterminals, but also further subdividing these subsets by the functions' return types.

The algorithm is as follows: Let $Y$ be the set of types. The set of functions $F$ is divided into two disjoint subsets nonterminals $N$ and terminals $T$. These subsets are further divided by their return types into subsets $Ny_1, Ny_2, \ldots Ny_z$, one for each $y \in Y$, and $Ty_1, Ty_2, \ldots Ty_z$, one for each $y \in Y$. During tree-generation time, the algorithm will, for some $y$, alternately choose new child nodes from either that $N_y$ or $T_y$. For each nonterminal $n_y \in N_y$ the user provides a probability $q_{n,y}$ that $n_y$ will be chosen when the algorithm needs a nonterminal with return type $y$. Similarly, for each terminal $t_y \in T_y$ the user provides a probability $q_{t,y}$ that $t_y$ will be chosen from $T_y$ when the algorithm needs a terminal with return type $y$. The user also provides a return type $y_r$ for the tree, and a maximum depth bound $D$, though this bound is used only to enforce an absolute, and preferably large, memory restriction. Lastly, the user indicates an expected tree size $E_{tree}$.

Before generating any trees, the algorithm computes $p_y$ for each $y \in Y$. $p_y$ is the probability of choosing a nonterminal over a terminal of return type $y$ in order to produce a tree with an expected tree size $E_{tree}$:

$$p_y = \frac{1 - \frac{1}{E_{tree}}}{\sum\limits_{n_y \in N_y} q_{n,y} b_{n,y}}$$

where $b_{n,y}$ is the number of arguments for nonterminal $n_y$. This computation need be done only once offline. Then the algorithm proceeds to create the tree:

**Algorithm 5** StronglyTypedPTC1

*Given:*
- maximum depth bound $D$
- disjoint nonterminal subsets $N_y$ of nonterminal set $N$ for each $y \in Y$
- disjoint terminal subsets $T_y$ of terminal set $T$ for each $y \in Y$
- computed nonterminal-choice-probabilities $p_y$ for each $y \in Y$
- for each $T_y$ and $N_y$,
  - probabilities $q_{n,y}$ and $q_{t,y}$ for each $t_y \in T_y$ and $n_y \in N_y$
- return type for the tree $y_r \in Y$

*Do:*
- new tree $\mathbf{T}$ = StronglyTypedPTC1$(0, y_r)$

*StronglyTypedPTC1*(depth $d$, return type $y \in Y$)
- *Returns:* a tree of depth $\leq D - d$ and of return type $y$

- If $d = D$, return a terminal from $T_y$ (by $q_{t,y}$ probabilities)
- Else if, with probability $p_y$ a nonterminal must be picked,

Choose a nonterminal $n_y$ from $N_y$ (by $q_{n,y}$ probabilities)
For each argument $a$ of $n_y$ of argument type $y_a$
Fill $a$ with StronglyTypedPTC1($d + 1$,$y_a$)
Return the completed nonterminal $n_y$ with filled arguments
Else return a terminal from $T_y$ (by $q_{t,y}$ probabilities)

Because of the user-provided type constraints of strongly-typed genetic programming, this version of PTC1 cannot guarantee that each terminal $t$ will appear in the tree with some probability $q_t$ relative to other terminals (or likewise a nonterminal $n$ appearing with probability $q_n$ relative to other nonterminals). Instead, it makes sure that each terminal $t_y$ of a type $y \in Y$ will appear with probability $q_{t,y}$ relative to other terminals of type $y$, and similarly that each nonterminal $n_y$ of type $y \in Y$ will appear with probability $q_{t,y}$ relative to other nonterminals of type $y$.

This algorithm guarantees an expected tree size of $E_{tree}$ for all STGP trees by determining the necessary probability $p_y$ for each return type $y$ such that subtrees returning that return type will each be of $E_{tree}$ size. Theorem 7 shows that the algorithm's method of picking of each $p_y$ is correct and invariant over of $y$.

**Theorem 7** Let $Y$ be is the set of STGP return types. Let the set of nonterminals $N$ be divided into nonempty subsets $N_{y_1}, N_{y_2}, \dots N_{y_z}$, one for each $y \in Y$. At tree-building time, let $p_y$ be the probability that a nonterminal will be chosen as a new child node for a particular type $y \in Y$. Given some $y$, for each $n_y \in N_y$, let $q_{n,y}$ be the probability that $n_y$ will be chosen given that a nonterminal of type $y$ is to be chosen, and $b_{n,y}$ be the number of arguments to $n_y$. Let $b_y$ be the expected number of children of a nonterminal node of return type $y$ in the tree, that is, $b_y = \sum_{n_y \in N_y} q_{n,y} b_{n,y}$. Then under StronglyTypedPTC1, the nonterminal-choice probability $p_y$ necessary to guarantee that a tree or subtree $\mathbf{T}$ of return type $y \in Y$ will be of expected finite size $E_{tree} > 0$ is: $p_y = \frac{1 - \frac{1}{E_{tree}}}{\sum_{n_y \in N_y} q_{n,y} b_{n,y}}$

**Proof** Let $g \geq 0$ be the expected number of children of a node newly generated by the algorithm. From Theorem 2, $E_{tree} = \frac{1}{1-g}$ if and only if $g < 1$. Thus $g$ may be determined from $E_{tree}$ as $g = 1 - \frac{1}{E_{tree}}$.

For any $y \in Y$, since $p_y$ denotes the likelihood that the newly generated node function will be a nonterminal, and the expected number of children to a terminal is 0, then the expected number of children of the newly generated node is $g = p_y b_y + (1 - p_y)(0) = p_y b_y$. Therefore $p_y = \frac{1 - \frac{1}{E_{tree}}}{b_y}$.

Since $N_y$ is nonempty, $b_y > 0$, so for any given $b_y$ and requested $E_{tree}$, an appropriate $p_y$ may always be determined. Replacing $b_y$ with $\sum_{n_y \in N_y} q_{n,y} b_{n,y}$ yields $p_y = \frac{1 - \frac{1}{E_{tree}}}{\sum_{n_y \in N_y} q_{n,y} b_{n,y}}$ ■

The algorithm can be modified to permit types for which there exists no nonterminal with a return value for that type; for each such type $y$, simply set $p_y$ to 0. However, this does not guarantee that the expected size of the tree will remain $E_{tree}$, only that it will be no larger than $E_{tree}$.

The complexity of this algorithm is dependent largely on the size of each set of functions by type, and the combinations of types of arguments to each function. However, the complexity is no worse than the complexity for PTC1 under ordinary GP (that is, ignoring the types to the functions in question). This is because the number of nonterminals that must be searched is no more than $\|N\|$, and the number of terminals is no more than $\|T\|$.

The algorithm can also be modified to accommodate set-based STGP. In this case the various sets $N_{y_x}$ are not required to be disjoint, nor are the various sets $T_{y_x}$. Choosing a nonterminal from some $N_{y_x}$ is still no more difficult than $O(\|N\|)$ and choosing a nonterminal from some $T_{y_x}$ is no more difficult than $O(\|T\|)$. However, since $\|Y\|$ may now be greater than $\|N\|$ and $\|T\|$; the complexity of PTC1 with set-based STGP is bounded by the maximum of the PTC1 complexity bound and $\|Y\|$. Of course $\|Y\|$ may be any size the designer likes, but it is rarely larger than $\|F\|$; at any rate, $\|Y\| \leq 2^{\|F\|}$, else $Y$ will contain duplicate types.

| | Natural | # Size 1 Trees when $E_{tree}$ is... | | | |
|---|---|---|---|---|---|
| Domain | $E_{tree}$ | Natural | 4 | 16 | 256 |
| Cart Centering | $\infty$ | $\frac{1}{4}$ | $\frac{13}{22}$ | $\frac{43}{88}$ | $\frac{643}{1408}$ |
| Ant | $\infty$ | $\frac{1}{2}$ | $\frac{19}{28}$ | $\frac{67}{112}$ | $\frac{1027}{1792}$ |
| Regression | $\infty$ | $\frac{1}{9}$ | $\frac{1}{2}$ | $\frac{3}{8}$ | $\frac{43}{128}$ |
| 11-Multiplexer | $\frac{15}{7}$ | $\frac{11}{15}$ | $\frac{5}{8}$ | $\frac{17}{32}$ | $\frac{257}{512}$ |
| 6-Multiplexer | 5 | $\frac{3}{5}$ | $\frac{5}{8}$ | $\frac{17}{32}$ | $\frac{257}{512}$ |
| 3-Multiplexer | 6 | $\frac{1}{2}$ | $\frac{11}{20}$ | $\frac{7}{16}$ | $\frac{103}{256}$ |

**Legend.** "Natural" indicates numbers if tree generation is made without restrictions on tree size (as in GROW). Other columns give numbers when PTC1 restricts the expected tree size $E_{tree}$ to various values. The probability of generating a size-1 tree is the probability of generating a terminal (e.g., $1 - p$).

Table 8.2: Expected Number of Size-1 Trees Generated, as a Percentage of the Whole Population

## 8.7 The PTC2 Algorithm

PTC1 generates trees with expected sizes around a specific user-defined value. A serious problem with PTC1 is that it does not give the user control over *variance* in tree size. PTC1, like GROW, produces a large number of small trees; there is little the user can do about it. For example, consider the large number of trees of size 1 (equal to $1 - p$) generated under the previous example (five nonterminals and five terminals, and an average nonterminal arity of 2). Using PTC1 enforcing an expected tree size of 10, about 11/20 of all new trees would be of size 1. Similarly, under GROW (no enforcement), exactly half of the trees generated would be of size 1, even though the expected tree size is infinity!

In general, when $E_{tree}$ is restricted to be less than GROW's expected tree size, then PTC1 generates more trees of size 1 than GROW would. If the enforced expected tree size is larger GROW's expected tree size, PTC1 will generate fewer small (or size 1) trees than GROW. Table 8.2 illustrates this for the introductory domains from [Koza 1992].

PTC2 avoids this problem by allowing the user to provide beforehand a probability distribution of requested tree sizes. Like PTC1, PTC2 also guarantees user-provided distributions of nonterminals and terminals appearing in each tree. And like PTC1, PTC2 is very fast. However, PTC2 is not as elegant as PTC1: when it picks a tree size from the distribution, it may produce a tree of that size or slightly larger. In effect, while PTC2 *guarantees* the user-provided nonterminal and terminal probabilities of appearance, it *approximates* the user-provided tree-size distribution.

PTC2 is as follows: the set of functions $F$ is divided into two disjoint subsets: nonterminals $N$ and terminals $T$. For each nonterminal $n$ in $N$ the user provides a probability $q_n$ that $n$ will be chosen from $N$ when the algorithm needs a nonterminal. Similarly, for each terminal $t$ in $T$ the user provides a probability $q_t$ that $t$ will be chosen from $T$ when the algorithm needs a terminal. The user also provides a maximum depth bound $D$, a maximum size bound $S$, and a probability distribution of desired tree sizes $w_1, ..., w_S$ for each tree size from 1 to $S$. Then the algorithm proceeds to create the the tree:

**Algorithm 6** PTC2

*Given:*
  maximum depth bound $D$
  maximum size bound $S$
  function set $F$ consisting of nonterminal set $N$ and terminal set $T$
  probabilities $q_t$ and $q_n$ for each $t \in T$ and $n \in N$
  probabilities $w_1, ..., w_S$ of generating a tree of size $\{1, ..., S\}$
*Do:*
  new tree $\mathbf{T}$ = PTC2()

*PTC2*()
  *Returns:* a tree of depth $\leq D$ and a size between $1$ and
    $S + b_{max}$ inclusive, where $b_{max}$ is the largest number
    of arguments to any nonterminal in $N$
  Let $r$ be a random tree size from the probability distribution $w_1, ..., w_S$

  If $r = 1$ return a random terminal from $T$ (by $q_t$ probabilities)
  Else
    Choose a nonterminal $n$ from $N$ (by $q_n$ probabilities)
    Tree root $root \leftarrow n$
    Node depth $d \leftarrow 1$
    For each argument position $a$ of $n$, Enqueue($\{a, d\}$)
    Current tree size $s \leftarrow 1$
    Until Size()$+s \geq r$ or Size()$= 0$,
      $\{$argument position $a$, depth $d\} \leftarrow$RandomDequeue()
      If $d = D$, Fill $a$ with a terminal $t$ from $T$ (by $q_t$ probabilities)
      Else
        Choose a nonterminal $n$ from $N$ (by $q_n$ probabilities)
        Fill $a$ with $n$
        For each argument position $a$ of $n$, Enqueue($\{a, d + 1\}$)
      $s \leftarrow s + 1$
    Until Size()$= 0$,
      $\{$argument position $a$, depth $d\} \leftarrow$RandomDequeue()
      Choose a terminal $t$ from $T$ (by $q_t$ probabilities)
      Fill $a$ with $t$
    Return $root$

The algorithm begins by picking a random tree size from the the user-provided tree-size probability distribution. It then attempts to build a tree of that size or slightly greater. The algorithm builds the tree by starting with a single node, $root$, extending the tree with nonterminals at random places along the current tree boundary. It continues this until the size of the unfilled positions along the boundary plus the number of nonterminals currently in the tree is greater than or equal to the requested size. Then the algorithm fills the remaining boundary positions with terminals, and returns the result.

Whenever the boundary extends beyond depth $D$, the offending boundary positions are automatically filled with terminals; this means that if $D$ is so small that a full tree of $S$ nodes might be greater than depth $D$, then PTC2 may return a tree smaller than expected.

To maintain the tree boundary, the algorithm stores in a global random queue the position and depth of each unfilled argument along the boundary, picking random items from this queue as needed. To do this, the algorithm relies on three random-queue functions: Size, which returns the size of the queue, Enqueue, which enqueues an item, and RandomDequeue, which dequeues and returns a random item. As shown below, the random queue can be implemented so that all three functions run in $O(1)$ time.

**Algorithm 7** Size, Enqueue, and RandomDequeue

*Given:*
array of slots $U = \{u_0, ..., u_{S+b_{max}-1}\}$
array-fill value $h \leftarrow 0$

*Size*()
*Returns:* the size of the queue

return $h$

*Enqueue*(item $i$)
*Returns:* nothing

$u_h \leftarrow i$
$h \leftarrow h + 1$

*RandomDequeue*()
*Returns:* the value of a random slot in $U$, or nil if $U$ is empty

If $h = 0$, return nil
Else
$h \leftarrow h - 1$
Let $r$ be a random integer, $0 \leq r \leq h$
Swap the values of $u_h$ and $u_r$
Return $u_r$

The random-queue implementation relies in this case on a maximum queue value of $S + b_{max}$, the largest returnable tree size. It can be instead implemented with a small initial queue array, extended when needed by doubling its size. This also yields an amortized complexity of $O(1)$ for all three operations.

### 8.7.1 Complexity of PTC2

Because random-queue operations can be done in $O(1)$ time, and either a nonterminal or terminal is chosen at each iteration, the complexity of building a tree of requested size $r$ is the time it takes to pick a random terminal or nonterminal (from the $q_t$ and $q_n$ distributions) multiplied by the number of iterations.

The first until-loop in PTC2 runs until Size()$+s \geq r$ or Size()$= 0$. In the first case, consider the last iteration of the first until-loop. As this iteration starts, Size()$+s < r$. The iteration may perform one last Enqueue before the iteration ends. Since the largest number of arguments to a nonterminal in $N$ is $b_{max}$, this last enqueing operation will increase Size()$+s$ to no more than $r + b_{max}$. At the point between the two until-loops, the first loop has run for exactly $s - 1$ iterations, and the second loop will run for exactly Size() iterations. Hence the total number of iterations is $O(r + b_{max})$.

In the exceptional second case (which will only occur when $D$ is inappropriately small relative to $S$), the first until-loop runs for no more than $r$ iterations, else the first case would have been triggered. The second until-loop will then run for 0 iterations, hence the total number of iterations is $O(r) \leq O(r + b_{max})$.

As discussed, the complexity of choosing a nonterminal from $N$ or a terminal from $T$ is $O(lg(||N||))$ and $O(lg(||T||))$ respectively, or both $O(1)$ under reasonable constraints. Since at each iteration either a nonterminal or a terminal is chosen, a loose complexity bound for choosing nonterminals and terminals in the algorithm is $O((r + b_{max}) \times \max(lg(||N||), lg(||T||)))$, or $O(r + b_{max})$ under reasonable constraints.

Likewise, picking randomly from the tree-size probability distribution takes at most $O(lg(||S||))$ time, or $O(1)$ under reasonable constraints. Let $r_{mean}$ be the mean tree size given the provided probability distribution. Then PTC2 has an average complexity of

$$O((r_{mean} + b_{max}) \times \max(lg(||N||), lg(||T||)))$$

Under reasonable constraints as discussed earlier, this reduces to $O(r_{mean} + b_{max})$. Since the largest possible tree is of size $S + b_{max}$, the worst-case complexity is therefore

$$O((S + b_{max}) \times \max(lg(||N||), lg(||T||)))$$

which under reasonable constraints reduces to $O(S + b_{max})$, effectively linear. If $D$ is too small relative to $S$ and the exceptional second case is triggered, then the complexity may be even lower.

### 8.7.2 Strongly-Typed PTC2

PTC2 can also be extended to handle "basic" strongly-typed genetic programming, assuming relaxed constraints similar to StronglyTypedPTC1. And just like StronglyTypedPTC1, StronglyTypedPTC2 assumes that for each user-provided type, there exists at least one nonterminal and at least one terminal whose return values are of that type.

The algorithm works as follows: the set of functions $F$ is divided into two disjoint subsets: nonterminals $N$ and terminals $T$. These subsets are further divided by their return types into subsets $Ny_1, Ny_2, \ldots Ny_z$, one for each $y \in Y$, and $Ty_1, Ty_2, \ldots Ty_z$, one for each $y \in Y$. During tree-generation time, the algorithm will, for some $y$, alternately choose new child nodes from either that $N_y$ or $T_y$. For each nonterminal $n_y \in N_y$ the user provides a probability $q_{n,y}$ that that function will be chosen when the algorithm needs a nonterminal with return type $y$. Similarly, for each terminal $t_y \in T_y$ the user provides a probability $q_{t,y}$ that $t_y$ will be chosen from $T_y$ when the algorithm needs a terminal with return type $y$. The user also provides a maximum depth bound $D$, a maximum size bound $S$, a requested return type for the tree $y_r$, and a probability distribution of desired tree sizes $w_1, ..., w_S$ for each tree size from 1 to $S$.

**Algorithm 8** StronglyTypedPTC2

*Given:*
maximum depth bound $D$
maximum size bound $S$
disjoint nonterminal subsets $N_y$ of nonterminal set $N$ for each $y \in Y$
disjoint terminal subsets $T_y$ of terminal set $T$ for each $y \in Y$
for each $T_y$ and $N_y$,
probabilities $q_{n,y}$ and $q_{t,y}$ for each $t_y \in T_y$ and $n_y \in N_y$
return type for the tree $y_r \in Y$
probabilities $w_1, ..., w_S$ of generating a tree of size $\{1, ..., S\}$
*Do:*
new tree $\mathbf{T}$ = StronglyTypedPTC2($y_r$)

*StronglyTypedPTC2*(return type $y{\in}Y$)

*Returns:* a tree of depth $\leq D$, of return type $y$, and of size between $1$ and $S + b_{max}$ inclusive, where $b_{max}$ is the largest number of arguments to any nonterminal in $N$

Let $r$ be a random tree size from the probability distribution $w_1, ..., w_S$

If $r = 1$ return a random terminal from $T_y$ (by $q_{t,y}$ probabilities)
Else
    Choose a nonterminal $n_y$ from $N_y$ (by $q_{n,y}$ probabilities)
    Tree root $root{\leftarrow}n_y$
    Node depth $d{\leftarrow}1$
    For each argument position $a$ of $n_y$, Enqueue($\{a, d\}$)
    Current tree size $s{\leftarrow}1$
    Until Size()$+s{\geq}r$ or Size()$= 0$,
        $\{$argument position $a$, depth $d\}{\leftarrow}$RandomDequeue()
        $y{\leftarrow}$argument type of $a$
        If $d = D$, Fill $a$ with a terminal $t_y$ from $T_y$ (by $q_{t,y}$ probabilities)
        Else
            Choose a nonterminal $n_y$ from $N_y$ (by $q_{n,y}$ probabilities)
            Fill $a$ with $n_y$
            For each argument position $a$ of $n_y$, Enqueue($\{a, d + 1\}$)
        $s{\leftarrow}s + 1$
    Until Size()$= 0$,
        $\{$argument position $a$, depth $d\}{\leftarrow}$RandomDequeue()
        $y{\leftarrow}$argument type of $a$
        Choose a terminal $t_y$ from $T_y$ (by $q_{t,y}$ probabilities)
        Fill $a$ with $t_y$
    Return $root$

This algorithm works identically to PTC2, except that like StronglyTypedPTC1 it too must loosen the guarantees on probability of occurrence of nonterminals and terminals. Namely, each terminal $t_y$ of a type $y{\in}Y$ will appear with $q_{t,y}$ probability relative to other terminals of type $y$, and each nonterminal $n_y$ of type $y{\in}Y$ will appear with $q_{t,y}$ relative to other nonterminals of type $y$.

Like StronglyTypedPTC1, the complexity of this algorithm is dependent on the size of each set of functions by type, and the combinations of types of arguments to each function. However, the complexity is no worse than the complexity for PTC2. This is because the number of nonterminals that must be searched at any time is no more than $\|N\|$, and the number of terminals is no more than $\|T\|$. And like StronglyTypedPTC1, this algorithm can be adapted to set-based STGP, with a complexity bounded by the maximum of the PTC2 complexity and $\|Y\|$.

## 8.8 Algorithm Summary

Unlike GROW, PTC1 and PTC2 add robustness to tree creation by permitting rigorous control over expected or actual tree size, giving the user a much-needed handle on tree growth during subtree mutation. Unlike other tree-creation algorithms which provide uniformly distributed tree structures but have high computational complexity, PTC1 and PTC2 provide uniform distribution of functions and have very low computational complexity. Both of these algorithms compare well with GROW in terms of runtime; GROW runs in $O(E_{tree})$ time, but permits $E_{tree}$ to be infinite in many common configurations.

PTC1 guarantees trees will be generated around an expected tree size, but does not provide control over variance in size. If the function set demands continuous-valued probabilities of appearance, PTC1 runs in $\leq O(\lg(\|F\|)E_{tree})$ time, where $\|F\|$ is the number of total functions, and $E_{tree}$ is the (finite) expected tree size. With reasonable constraints, PTC1 can run in $O(E_{tree})$ time.

PTC2 takes a user-provided probability distribution by tree size, and approximates generating trees from this distribution. PTC2 runs in $O((r+b_{max}) \times \max(\lg(||N||), \lg(||T||)))$ time, or $O(r+b_{max})$ with reasonable constraints, where $r$ is the average tree size in the probability distribution, $||N||$ is the number of nonterminals and $||T||$ the number of terminals in the function set, and $b_{max}$ is the largest number of children of any nonterminal in the function set.

# Chapter 9

# Code Bloat: An Inside Look

What causes code bloat in general? Although Chapter 8 showed one reason for code bloat under subtree mutation, Chapter 7 shows that trees often express much greater bloating characteristics, and especially under subtree crossover. This chapter peeks under the hood of tree evaluation and breeding, directly examining causes of bloat and suggesting that the common wisdom about bloat may be incorrect. It then presents a new, more general theory of the causes of bloat.

## 9.1 Introns

Much of the code bloat literature in genetic programming revolves around discussion of so-called "introns" — extraneous regions in an individual which neither add nor detract from its fitness, because they are ignored or do nothing. Evolutionary computation borrows many of its notions from biology, and genetic programming's use of the term "introns" is a classic example.

In biological terms, regions of DNA on chromosomes may be roughly divided into two classes: *genes* containing *intragenic DNA*, and *extragenic DNA*. Genes make up roughly 20 to 30 percent of the human genome [Brown 1992]. A gene is a set of DNA nucleotides which are responsible for *transcribing* a given string of RNA. This RNA may in turn be used to produce protein through a complex process called *translation*.

Not all of the nucleotides in a gene are actually used to produce the protein or RNA endproduct. Much of the gene is often spliced out during transcription; these spliced-out portions, known as *introns*, do not play a significant role in the production process. The portions of a gene actually transcribed into RNA are known as *exons*[1]. It is generally believed that introns and other non-coding DNA in a gene make up over 90% of a typical gene in the human genome [Brown 1992]. Thus the amount of genetic material actually active in the human genome is under 3%.

### 9.1.1 Introns in Genetic Programming

[Angeline 1994] appears to be the first publication to identify genetic programming introns and give them their nickname. Angeline defined introns as areas of code that "are unnecessary since they can be removed from the program without altering the solution the program represents". This general definition has been used, and abused, in a variety of ways throughout the code bloat literature, and so it is very important to provide a specific definition. Three uses of the term seem to be mixed up in the literature, along with many variant terms:

1. Introns are areas of code which can be trivially simplified without modifying the individual's operation.

2. (A subset of #1) Introns are subtrees which cannot be replaced by anything that can possibly change the individual's operation. Such code is associated with an *invalidator*, a structure elsewhere in the individual

[1] In fact, not all of a transcribed exon is finally used in producing protein: some portions of an exon, known as *untranslated regions* (UTRs), do not participate in translating the RNA to protein

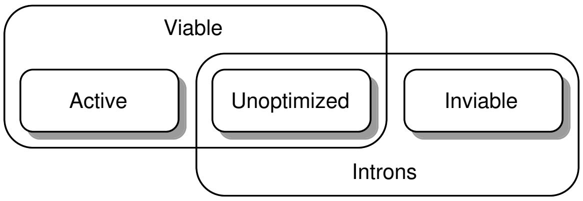



Figure 9.1: A Venn Diagram of Labels Used to Describe Various Kinds of Code

which is responsible for nullifying the intron's effect. [Soule and Foster 1998] and [Langdon *et al.* 1999] call these introns "inviable code", while [Banzhaf *et al.* 1998] call them "absolute introns", [Blickle and Thiele 1994] and [Blickle 1996] call them "redundant nodes", and [Rosca 1996] calls them "ineffective code". [Nordin, Francone, and Banzhaf 1995] call them "type 1" and "type 2" introns. [Angeline 1998] calls them "syntactic introns". It is possible for two invalidators to make each other inviable. I call this "co-inviable code".

3. (Those introns in #1 but not in #2) Introns are subtrees which perform no function in the individual, but can be replaced with subtrees which do perform a function. [Langdon *et al.* 1999] call this "inoperative code", though they unfortunately restrict their term in such a way as to exclude plausible candidates such as redundant parents, as in (not (not *foo*)), or redundant subtrees, as in (and d1 d1). [Nordin, Francone, and Banzhaf 1995] similarly make this exclusion, calling them "type 3" and "type 4" introns. [Tackett 1994] calls them "inert subexpressions". [Angeline 1998] calls them "semantic introns".

Although definition #2 is the closer analog to biological introns, definition #1 seems more common in the literature and so I will use it as the meaning of "introns". For definition #2 I adopt the term *inviable code*. For definition #3, I will use the term *unoptimized code*. Code which is not part of an intron will be termed "active". Active and unoptimized code will be termed "viable". Figure 9.1 illustrates this with a Venn diagram. For specific examples of inviable and unoptimized code, see Appendix A.

### 9.1.2 Theories of Code Bloat

[Angeline 1994] argued that individuals with large numbers of introns stood a better chance of surviving crossover intact, but he viewed this feature as beneficial to the evolutionary process. [Tackett 1994], citing personal communication from Andrew Singleton in 1993, also noted this argument, though Tackett himself disagreed with it. Subsequent literature, however, has argued strongly that introns are a chief culprit in code bloat, and thus are not desirable.

Generally speaking, there are three overlapping theories which implicate introns in code bloat: hitchhiking, defense against crossover, and removal bias.

#### Hitchhiking

The first theory, *hitchhiking* [Tackett 1994] says that if introns (the hitchhikers) are attached to parents of "important" active code (though Tackett did not use the term, this is effectively building blocks), then crossover which preserves this active code is likely to take some of these introns along with it and thus propagate introns throughout the population. Tackett noted that if the evolutionary computation system was more selective, code growth would increase. He argued that this was because, with the increase in selectivity, the value of important building blocks increased, and as these building blocks were replicated more rapidly through the population, introns hitchhiking on these building blocks would spread all the faster.

### Defense Against Crossover

In the second theory, *defense against crossover*, introns are propagated because they act to make it difficult to destroy an individual by increasing the number of crossover points which have no effect on the individual. This gives rise to *neutral crossover*, where the child is identical in function (and fitness) to its parent. Despite its name, this theory is equally applicable to a wide variety of non-crossover tree-manipulation mechanisms, including subtree mutation.

This general theory has been cited, in one way or another, by a large chunk of literature: [Blickle and Thiele 1994; McPhee and Miller 1995; Nordin and Banzhaf 1995; Rosca 1996; Blickle 1996; Rosca 1997; Banzhaf *et al.* 1998; Langdon *et al.* 1999; Andre and Teller 1996]. The chief theoretical support for defense against crossover was outlined in [Blickle and Thiele 1994] (later [Blickle 1996]), and [Nordin and Banzhaf 1995] (later [Banzhaf *et al.* 1998]). The argument is roughly as follows: let $s_i$ be the expected number of times an individual $i$ will be selected from a given generation to undergo crossover or reproduction. Let $p_c$ be the probability that crossover will be the breeding mechanism. Let $C_{ai}$ be the number of crossover points in the individual (its *absolute complexity*), and let $C_{ei}$ be the number of crossover points which might result in something other than neutral crossover (its *effective complexity*). Let $d_i$ be the probability that individual $i$ will be damaged by crossing over in non-neutral crossover points. Obviously the probability of damage by crossing over in neutral points is 0. Then the expected number of children $n$ of individual $i$ that are better than or equal to $i$ in fitness is equal to:

$$n = s_i(1 - p_c(\frac{C_{ei}}{C_{ai}}d_i + \frac{1 - C_{ei}}{C_{ai}}0)) = s_i(1 - p_c\frac{C_{ei}}{C_{ai}}d_i)$$

From this equation [Blickle and Thiele 1994] and [Nordin and Banzhaf 1995] independently argue that initially crossover is equally likely to be constructive as destructive, but as an evolutionary run progresses, individuals become fitter and finding a better solution becomes rarer. In this later situation, the most survivable individuals are the ones with a small ratio of effective to absolute complexity, so most crossover doesn't do anything at all, maintaining the individual's existing high fitness.

It is notable, however, that this equation is only applicable to inviable code when applied to tree-based GP. Indeed, one surprising feature of the defense-against-crossover literature is that while most proponents of this theory argue that *introns* increase the number of ineffective crossover points in an individual, in reality the lion's share of experiment and all theoretical justification have dealt solely with inviable code for tree-based GP.

While for linear string GP systems all kinds of introns can decrease effective complexity (as noted in work in AIMGP [Nordin and Banzhaf 1995; Banzhaf *et al.* 1998]), in tree-based GP, effective complexity can only be reduced through inviable code, as noted by [Blickle and Thiele 1994]. Similarly, [Blickle and Thiele 1994] and [Blickle 1996] use only inviable code in their experimental examples, as does [Rosca 1996]. Some of the literature ([Langdon 2000; Rosca 1997]) simply defines introns to be inviable code. Other experiments ([Langdon *et al.* 1999]) support the theory in abstract but are not specific enough to distinguish between inviable and unoptimized code.

In summary, while the defense against crossover theory is typically stated in terms of introns in general, proofs of the theory consistently deal with inviable code only, and experiments with the theory generally focus on inviable code in tree-based genomes. I think it is fair to say that, at least for its history in the tree-based genetic programming literature so far, defense against crossover is primarily an inviable code theory.

### Removal Bias

A recent third theory, *removal bias*, eschews unoptimized code entirely and focuses solely on inviable code [Soule and Foster 1998; Langdon *et al.* 1999]. In this theory, the presence of inviable subtrees provides safe harbors for tree growth. The theory first proposes that if an individual contains inviable subtrees, it is more likely to survive if it performs modifications (crossover, mutation) within these subtrees. In some sense this is similar to defense against crossover. But removal bias goes further, suggesting an actual mechanism which prefers larger trees within these inviable subtree areas. In order to guarantee preservation of the individual, the subtree removed during modification must be no larger than the inviable subtree area; hence there is a bias towards removing small subtrees from the individual. But the *replacing* subtree has no such bias at all — it can be any size and still have no effect on the individual. From this the theory predicts that inviable subtrees will grow without bound.

Recently [Soule and Foster 1998; Langdon *et al.* 1999] have a form of hillclimbing to argue for the removal bias theory of tree bloat. This hillclimbing technique, originally proposed by [O'Reilly 1995], rejects crossover results which decrease the fitness of an individual (or more strongly, do not increase the fitness of the individual). If crossover fails this standard, then the parent is replicated into the new population in lieu of the failed child. By rejecting crossover which decreases fitness, tree growth is slowed. By rejecting crossover which does not increase fitness, tree growth is slowed even further. The claim is that the difference in growth rates between these techniques is due to the rejection of any crossover events which occur in inviable code areas (and thus do not change the individual's fitness). [Langdon and Poli 1998] use a related technique and report similar results. One weakness in this argument is that this technique also acts to replicate large numbers of parents into future generations. Further, rejecting children with no increase in fitness serves to replicate many more parents than simply rejecting children which decrease fitness. If some (unknown) code bloat force is causing children to be larger than their parents, then this technique may be doing little more than artificially dampening code growth by filling the population with parents and ancestors, which are generally smaller then their descendents.

#### A Non-Intron Code Bloat Theory

The chief non-intron bloat theory in genetic programming is due to Langdon and Poli. I call this the *diffusion* theory applied in various ways in [Langdon and Poli 1997b; 1997c; Langdon 1998; Langdon and Poli 1997a; Langdon *et al.* 1999; Langdon 2000]. According to this theory, in the space of programs, there are generally many more large-sized highly-fit trees than there are small-sized ones, if only because there are many more large trees than small ones overall. If genetic programming searched uniformly in this space, average tree size would be expected to remain constant. But genetic programming runs start out with small trees initially, and so code bloat might be explained simply as the system moving towards equilibrium.

This theory is supported with an argument borrowed from Price's Covariance and Selection Theorem [Price 1970] from population genetics. The diffusion theory adapts this theorem to suggest a relationship between how selective the system is and the growth in tree size, taking a cue from [Tackett 1994]. When the evolutionary system suddenly chooses individuals at random (selectivity drops to 0), trees begin to shrink rather than grow. [Langdon 2000] further argues that this code growth is not exponential but subquadratic in the number of evaluations.

## 9.2 Experiments

The two most prevalent intron theories of code growth (defense against crossover and removal bias) both rely on a similar thesis: that crossover in inviable subtree areas is the driving force behind tree growth in general. In this paper I present what I believe to be the first experimental evidence against this claim. The experiments do the surprisingly obvious: deny individuals the ability to cross over in inviable areas.

[Blickle and Thiele 1994; Blickle 1996] proposed exactly this, calling it *marking*: identifying inviable code areas and disallowing crossover within those regions. Unfortunately, no code growth results were presented, and [Blickle 1996] later wrote off the technique, stating "the marking method only avoids redundant crossover sites but does not address the bloating phenomenon directly as it leaves the redundant subtrees unchanged." In fact, marking does significantly decrease the amount of inviable code. But what is surprising is that marking does not appear to affect tree growth, even in inviable code-heavy domains.

| Inviable Code Examples | |
|---|---|
| true as invalidator | (or *true inviable*) |
| | (or *inviable true*) |
| | (if *true executed inviable*) |
| false as invalidator | (and *false inviable*) |
| | (and *inviable false*) |
| | (if *false inviable executed*) |
| Invalidation of test | (if *inviable* a0 a0) |
| Conspired invalidation (examples) | (and (not a0) (and a0 *inviable*)) |
| | (or d0 (or *inviable* (not d0))) |
| Co-inviable code (examples) | (or *inviable-returns-true inviable-returns-true*) |
| | (and *inviable-returns-false inviable-returns-false*) |

| Invalidator Examples | |
|---|---|
| Creating true (example) | (not (and a0 (not a0))) |
| Creating false (example) | (and d1 (not d1)) |

| Unoptimized Code Examples | |
|---|---|
| Redundant Subtrees (examples) | (and a0 (not a0)) |
| | (or d0 (and d0 d0)) |
| Redundant Parents | (not (not (not (not a0)))) |

Table 9.1: Intron Examples for the 6- and 11-Multiplexer Domains

To test this I performed experiments in three genetic programming domains: Symbolic Regression, 11-Bit Multiplexer, and 6-Bit Multiplexer, all with and without marking Each experiment consisted of 50 random runs.

All experimental runs lasted 64 generations, or until an ideal solution was found, using a population of 512, and 7-tournament selection. The only breeding mechanism used was one-child crossover: two children are picked from the population and crossed over. However, while the first child is placed into the next generation, the second child is discarded. One child crossover was illustrated in Figure 2.1. Rather than using the traditional ad-hoc node-selection scheme (picking terminals 10% of the time), node selection was uniform. However, unreported experiments were also run with 10% terminal and 90% nonterminal selection, with similar results. Additionally, no limit was placed on the size of trees, in an attempt to gauge the true, unbiased dynamics of subtree modification with respect to tree growth. Symbolic Regression did not use ephemeral random constants. ECJ was the genetic programming system used [Luke 2000], as described in Appendix B.

The various figures show eight graphs of data as the fifty evolutionary runs progress, with a solid black line showing the mean. These graphs are:

- Mean depth of individuals in the run.
- Mean number of nodes of individuals in the run.
- Number of individuals identical in fitness to their parents (neutral crossovers).
- Best fitness so far in the run.
- Inviable nodes as a percent of all nodes in the individual.
- Occurrence of invalidators (the structures responsible for nullifying the effects of inviable code) as a percent of all nodes in the individual.

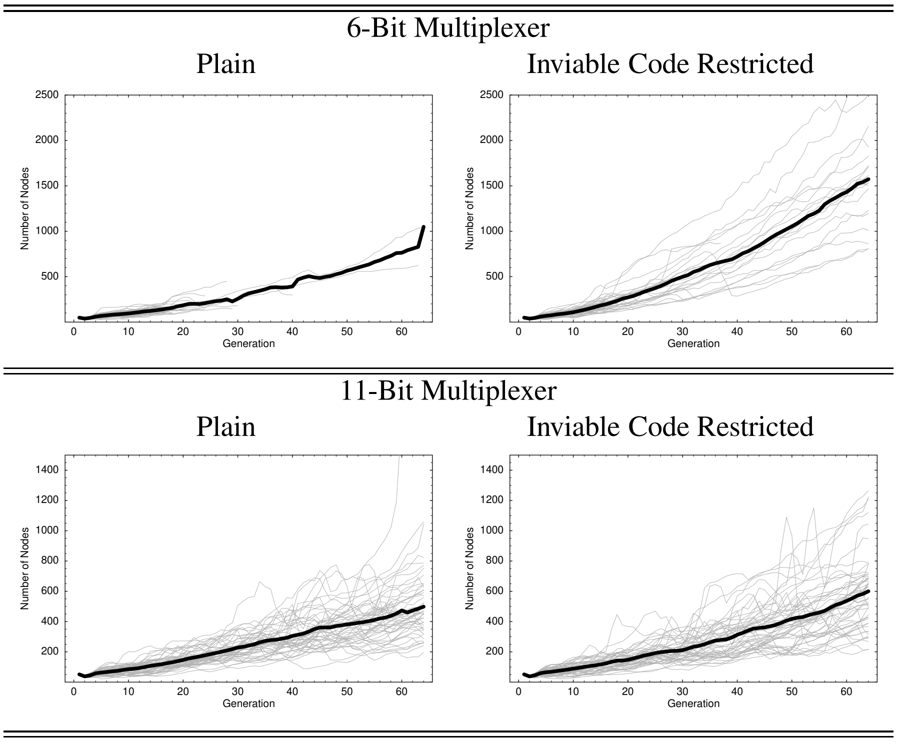



Figure 9.2: Mean Number of Nodes Per Individual for the 6-Bit and 11-Bit Multiplexer Domains

- Viable invalidators as a percent of all nodes in the individual. A viable invalidator is an invalidator structure which is located in a viable code region (and so is active).

- Base invalidators as a percent of all nodes in the individual. A base invalidator is an invalidator which propagates a series of invalidator events. For example, in (∗ (∗ 0 *inviable1*) *inviable2*), there are two invalidators, namely 0 and (∗ 0 *inviable1*), since both return 0 and are multiplied against inviable code. In this case, the base invalidator is 0.

### 9.2.1 Multiplexer

Table 9.1 shows many common Multiplexer introns and their causes. One nice feature of Multiplexer is that it is possible, if expensive, to identify all inviable code. The most common forms of inviable code in Multiplexer are operations with true or false, such as (or *returns-true inviable*). In a less common form, parents and parents' siblings work together to eliminate the effect of inviable code. A simple example of this is (and (not a0) (and a0 *inviable*)). For 11-Bit Multiplexer, these less common forms are so rare that it is unlikely a single one will occur during a run. For 6-Bit Multiplexer, the less common forms are also rare, though they occur more often.

For each Multiplexer domain, two sets of runs were performed, each with fifty independent runs. In the first set, runs were performed normally as described earlier. In the second set, the crossover point in the first parent was specially chosen through marking: a node was picked at random from the set of viable nodes in the individual. Since the child of the second parent was discarded, this meant that all children would be generated from a crossover point chosen from among viable nodes only.

The full results of these experiments, shown in Figures C.9 through C.16, were surprising. A selection of these figures, shown in Figure 9.2, shows just the tree growth results with and without marking inviable code for both 6-Bit and 11-Bit Multiplexer.

| Inviable Code Examples | |
|---|---|
| 0 as invalidator | (∗ 0 *inviable*) |
| | (∗ *inviable* 0) |
| | (% *inviable* 0) |
| | (% 0 *inviable*) |
| $\pm\infty$ or *NaN* as invalidator | (∗ $\pm\infty$*-or-NaN inviable*) |
| | (% $\pm\infty$*-or-NaN inviable*) |
| | (+ $\pm\infty$*-or-NaN inviable*) |
| | (- $\pm\infty$*-or-NaN inviable*) |
| | (∗ *inviable* $\pm\infty$*-or-NaN*) |
| | (% *inviable* $\pm\infty$*-or-NaN*) |
| | (+ *inviable* $\pm\infty$*-or-NaN*) |
| | (- *inviable* $\pm\infty$*-or-NaN*) |
| Decimation (example) | (rlog (+ (sin *inviable*) (exp (exp *returns-about-6*)))) |
| Co-inviable code (examples) | (∗ 0 $\pm\infty$) |
| | (+ *NaN* $\pm\infty$) |
| | (% 0 0) |

| Invalidator Examples | |
|---|---|
| Creating 0 (examples) | (- x x) |
| | (rlog (% x x)) |
| Creating $\pm\infty$ (example) | (exp (exp (exp (exp (% x x))))) |
| Creating *NaN* (example) | (sin $\pm\infty$) |

| Unoptimized Code Examples | |
|---|---|
| Redundant Subtrees (example) | (+ (- x x) (rlog (% x x))) |
| Redundant Parents (example) | (rlog (exp (rlog (exp (rlog x))))) |

Table 9.2: Intron Examples for the Symbolic Regression Domain

These results were surprising. After denying the ability to cross over in inviable regions, code growth in 11-Bit Multiplexer *increased*. In 6-Bit Multiplexer, code growth also increased. However, these increases were not statistically significant (using an ANOVA with an alpha value of 0.05). Nonetheless, they were not the expected *decreases* predicted by the intron theories. Tree depth similarly increased. At the same time, the number of inviable nodes, as a percent of each individual, dropped dramatically. Note that the growth in neutral crossovers was unchanged. This is expected: an unusual feature of the Multiplexer domains is the very high likelihood of performing crossover which, while dramatically changing the functional semantics of an individual, does not change its overall score. Fitness change was negligible, except in 6-Bit Multiplexer, which found perfect scores much more rapidly without marking.

### 9.2.2 Symbolic Regression

Table 9.2 shows common Symbolic Regression introns and their causes. Symbolic Regression inviable code takes three forms: multiplication or division by zero, multiplication, division, addition or subtraction with infinity/NaN, and decimation.

Multiplying and dividing by zero is by far the most common cause of inviable code in Symbolic Regression: for example, (∗ *always-returns-zero inviable*). Less obvious is how to achieve operations involving infinity or

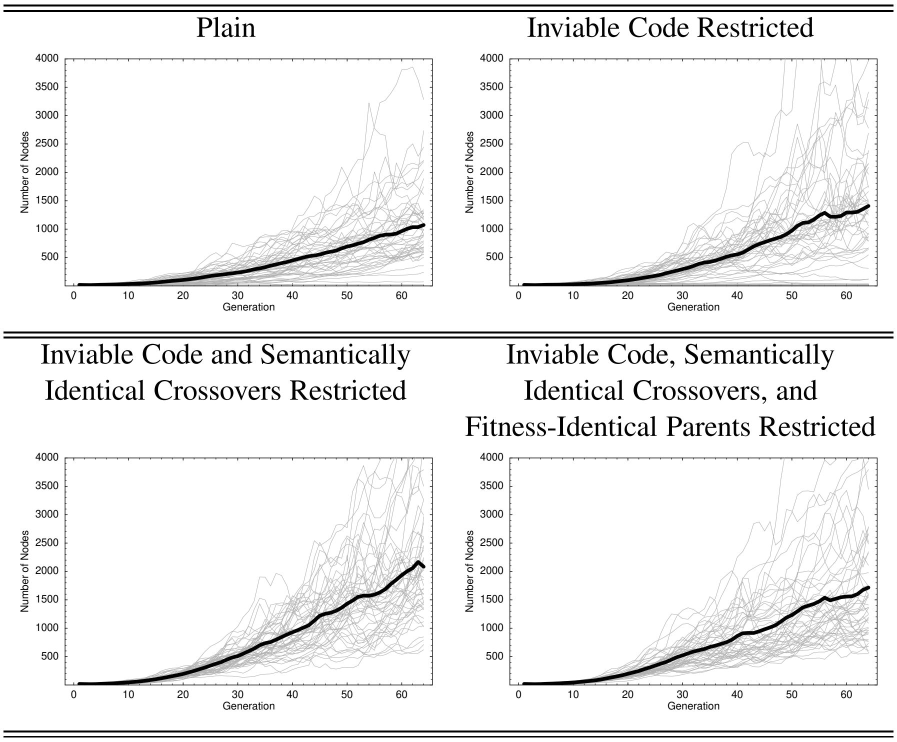



Figure 9.3: Mean Number of Nodes Per Individual for the Symbolic Regression Domain

NaN. Structures returning infinity can be achieved with successive calls to exp, as in (exp (exp (exp (exp (exp *foo*))))). Structures returning NaN can be achieved with (sin *infinity*). However, infinity and NaN dominate the return value of an individual; as such when they appear, they give the individual the worst possible fitness, and so cannot propagate.

Decimation is more insidious. In decimation, parent nodes work together to eliminate the effect of a child by dropping its contribution below the precision of the data type. An example of decimated inviable code for the Java double type is (rlog (+ (sin *inviable*) (exp (exp *6*)))). While most forms of decimation are nearly impossible to identify directly, their effect can be ascertained indirectly by tracking the growth of neutral crossovers for which decimation can be the only possible culprit. In this regression experiment, such tracking determined that decimation never occurred prior to generation 15, and rarely occurred prior to generation 25.

Symbolic Regression differs from Multiplexer in that it is extremely unlikely that crossover of semantically different subtrees (that is, subtrees which return different values) will result in identical fitness, except within inviable code regions. This means that restricting semantically-identical crossover can have a significant impact on the number of neutral crossovers.

In order to gauge the effects of introns on tree growth in Symbolic Regression, four sets of runs were performed, each with fifty independent runs. In the first set, as usual, runs were performed unencumbered. In the second set, the crossover point for the first parent was chosen through marking. Note that this marking does not eliminate inviable code due to decimation.

In the third set, the crossover point for the first parent was chosen through marking, and the system rejected semantically identical crossover points. This means that once the system had selected two individuals for crossover, the system would try 500 times to find two semantically dissimilar subtrees to cross over. It would reject semantically identical subtrees such as x and (+ (- x x) x). If it failed 500 times (which only occurred when both individuals were simply the terminal x), the first parent was replicated in lieu of its child. The goal of this set of runs was to eliminate any possibility that unoptimized code could be "increasing" neutral crossover points through semantically-identical crossover, which will be discussed more later. After applying these restrictions, there are only two possible causes for neutral crossover left: crossover inside decimated inviable code, and crossover between two trees consisting of the single terminal x (very rare after generation 7).

In the final set, the first parent was chosen through marking, the system rejected semantically identical crossover points, and decimated inviable code was controlled through a special technique: disallowing an individual to be selected if it had the same fitness as its parent (caused by decimation).

The full results are shown in Figures C.1 through C.8. A selection of these figures, shown in Figure 9.3, shows just the tree growth results using these four sets of runs.

In all three restriction approaches, tree growth increased. Using an ANOVA with Fisher LSD at 0.05, this difference was statistically significant. Using an ANOVA with Tukey at 0.05, the tree growth increase was statistically significant in the third and fourth sets. Again, the surprising result is that the expected hypothesis (that tree growth would decrease) was not fulfilled. Tree depth similarly increased. Changes in fitness were again negligible.

### 9.2.3 Discussion

Proponents of intron theories point to the increase in inviable nodes, neutral crossovers, and tree growth and suggest that the correlation among them is in fact causation: inviable code growth is driving tree growth. But these experiments suggest a more likely relationship. Associated with each inviable subtree is an invalidator, a chunk of code responsible for making the subtree inviable. For example, in ($*$ 0 *inviable*), 0 is the invalidator. If invalidators were uniformly but randomly distributed, their effect on large trees would be much higher than on small trees, since in large trees there is a higher probability that an invalidator is proportionally closer to the root. Thus if trees grow but invalidator distribution remains constant, then the percentage of inviable code should grow naturally towards 100%. As each figure shows, invalidators generally remain constant throughout the run. Additionally, Figure C.7 shows that even though individuals with neutral crossover (caused by decimation) were immediately terminated, decimation events continued to rise (indicated by the rise in neutral crossovers). This suggests that growth in decimated inviable code is also driven by overall tree growth. In other words, these experiments suggest that *tree growth is the cause of inviable code growth.*

One theoretically possible source of tree growth is an increase in neutral crossover due to unoptimized code propagation. The idea here is that perhaps many unoptimized code subtrees are semantically identical, and so their propagation increases the likelihood that crossover might accidentally trade two such subtrees, resulting in neutral crossover; however in Figures C.5 and C.6, such propagation was entirely eliminated and code continued to grow unabated.

## 9.3 Exploratory Analysis

If not introns, then what is causing these trees to grow? Some exploratory statistics gathered from these results reveal interesting trends. For crossover events in the previous experiments, the following data was gathered from the first parent:

- The depth of the crossover point chosen.
- The size of the parent.
- The size of the subtree removed.
- The size of the subtree inserted.
- The number of times the child was later selected for crossover (its *child survivability*). Note that this is *not* the fitness of the child; it is roughly equivalent to the quality of the child relative to its peers in its generation.

From this data I selected five generations' slices of data to view: 4, 8, 16, 32, and 64. Each data slice is labeled by the generation in which survivability data was gathered for each child, that is, when each child's children were evaluated. For example, "generation 4" means that in generation 2 the original parent was evaluated, in generation 3 its child was evaluated, and in generation 4 the child's children were evaluated.

**Goal and Caveats** If there is a positive relationship between child survivability and some factor in code bloat, then the population should be expected to move towards increasing that factor. The goal of this analysis is to compare candidate factors for tree bloat (parent size, crossover point depth, size of removed and inserted subtrees) against their effect on child survivability, with or without marking. Additionally, crossover point depth is compared against parent tree size and against removed subtree size to gauge relationships between them (which would suggest that crossover point depth drives the other factors in tree growth).

However, this data analysis is exploratory only. As such, there are three caveats which must be taken into consideration. First, because this is a post-hoc analysis of the run data, the data slices are dependent from generation to generation. This means that conservative analysis of this data would take into consideration only the results for one generation, as the other results are strongly tied to that generation. I provide all five generations, but suggest considering only either generation 8 or 16. This is because generation 4's results are probably too reflective of initialization phenomena (its evaluation occurred only in generation 2), and generations later than 16 are probably not statistically relevant for 6-Multiplexer due to the drop in sample size. Second, the data is overdispersed, as expected. Regression analysis of this data was scaled to compensate for this, but nonetheless this must be taken into consideration. Third, crossover point depth, parent size, and removed subtree size are multicollinear, that is, there is a high correlation between them — deeper crossover point depths are related to larger parents and smaller removed subtrees.

### 9.3.1 Graphs

For each problem domain and generation, graphs were produced relating child survivability with crossover point depth, parent size, removed subtree size, inserted subtree size, and net change in size from parent to child (inserted minus removed). Child survivability is considered the dependent variable. This data is noisy and overdispersed, and child survivability follows a Poisson, not normal, distribution. Thus certain measures were taken in an attempt to make the trends in the data visualizable. This varied depending on what independent variable was being shown, as follows:

- **Crossover Point Depth.** The independent variable was normalized by dividing it by the mean for its population at that generation. The data was then sorted and divided evenly into 50 bins by the independent variable.

- **Parent Tree Size, Removed Subtree Size, and Inserted Subtree Size.** The independent variable was normalized by dividing it by the mean for its population at that generation. Because tree size is at least superlinear and at most exponential with tree depth, the independent variable was transformed with a logarithmic scale. The data was then sorted and divided evenly into 50 bins by the independent variable.

- **Net Difference in Inserted and Removed Tree Size.** Because this data centers around 0, and contains negative values, it cannot be normalized by the mean, nor is a log transform appropriate. Hence, the data was simply sorted and divided evenly into 50 bins by the independent variable. The downside of this is that much of the data is clumped close to the $Y$ axis.

Each point on the graph represents the means of a bin both for the independent variable and for survivability. In all graphs, the $Y$ axis marks the mean for the population (or very close to the mean, for the net-difference graphs).

Figures C.17 through C.19 show this data arranged for the fifty plain Symbolic Regression runs. Figures C.20 through C show the data for Symbolic Regression with inviable code restricted. Similarly, Figures C.23 through C.25 and C.26 through C.28 show data arranged for 11-Bit Multiplexer with and without inviable code restriction. Figures C.29 through C.31 and C.32 through C.34 do likewise for 6-Bit Multiplexer.

### 9.3.2 Regression Analysis

For each set of runs, nine subtables are presented showing exploratory regression analysis of the data. Tables C.1 and C.2 show analysis of Symbolic Regression. Tables C.3 and C.4 show analysis of Symbolic Regression with inviable code restricted. Tables C.5, C.6, C.7, and C.8 show analysis of 11-Multiplexer with and without inviable code restricted. Similarly, Tables C.9, C.10, C.11, and C.12 show analysis of 6-Multiplexer with and without inviable code restricted.

For each set of runs, the first subtable presents multiple regression results for child survivability as a function of crossover point depth, parent tree size, removed subtree size, and inserted subtree size. The second subtable presents multiple regression results for child survivability as a function of crossover point depth, the log of parent tree size, the log of removed subtree size, and the log of inserted subtree size. The reason for the second table is to make certain that crossover point depth doesn't get an unfair advantage, if tree size is at most exponential in depth. Both of these tables regress with a Poisson distribution and logarithmic link function, because child survivability follows a Poisson curve. The tables are scaled with the square root of Pearson's $\chi^2$/DoF.

The third and fourth tables present simple regression results for parent tree size as a function of tree depth, and removed subtree size as a function of tree depth, respectively. These two tables regress with a normal distribution and a linear link function, and are scaled by maximum likelihood.

The next five tables present simpler regression analyses of child survivability against crossover point depth, parent tree size, removed subtree size, and inserted subtree size. The analyses present these factors independently, rather than jointly as in the first two multiple regression tables. These five tables roughly correspond to the five graph columns for each domain.

Those estimates whose alpha values exceed 0.05 (the test value used) are marked with asterisks. These estimates are statistically insignificant. Because of the high number of observations used (typically about 25,000), if an estimate's alpha value did not exceed 0.05, it was almost always less than 0.01 and usually much less than 0.0001. The deviance per degree of freedom prior to scaling is also given. As can be seen, the data is overdispersed (it much more or much less than 1.0). After scaling, the deviance was consistently close to 1.0.

### 9.3.3 Discussion

Although this is only exploratory data, five features appear to be almost invariant. They are (followed by the relevant regression analysis subtables, found in Appendix C):

1. The strongest relationship with survivability is almost always its positive relationship with crossover point depth, even when other factors are placed on a log scale (first and second subtables).

2. There is consistently a positive relationship between parent tree size and child survivability (sixth subtable).

3. There is consistently a negative relationship between removed subtree size and survivability. This remains so even after marking (seventh subtable). This effect is always stronger than that of inserted subtree size, with or without marking (ninth subtable).

4. There is consistently a positive relationship between subtree point depth and parent tree size (third subtable).

5. There is consistently a negative relationship between subtree point depth and removed subtree size (fourth subtable).

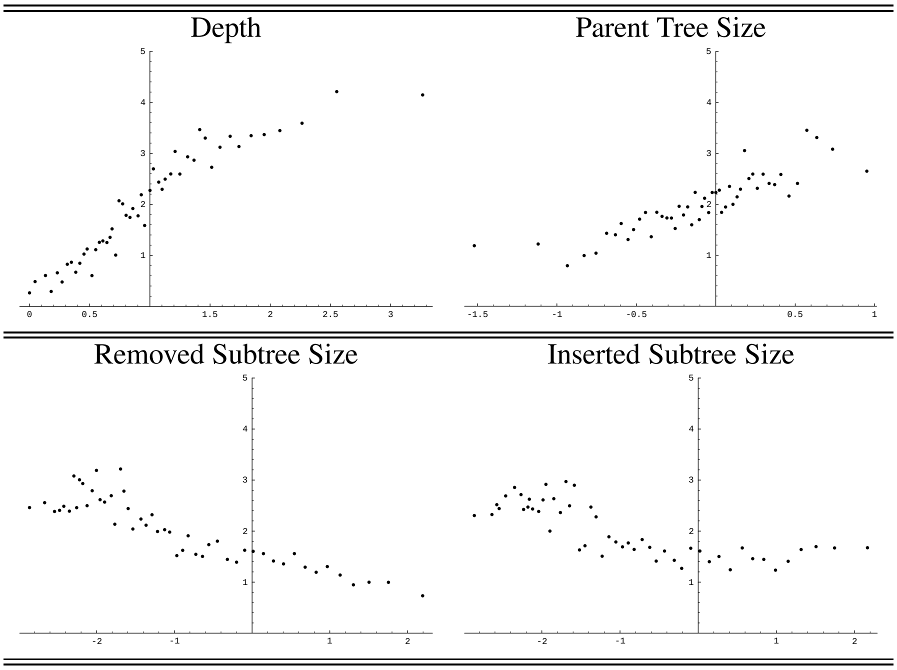



Figure 9.4: Relationship of Depth, Parent Tree Size, Removed Subtree Size, and Inserted Subtree Size to Child Survivability, Symbolic Regression Domain, Generation 16

The first three of these relationships is illustrated in Figure 9.4. The full set of related figures is found in Appendix C.

Surprisingly, in the Symbolic Regression domain there is *also* a consistent negative relationship between inserted subtree size and child survivability. However, in the Multiplexer domains, there is only a weak positive relationship between inserted subtree size and child survivability.

One oddity in the multiple regression tables (the first and second for each domain) is that when crossover point depth has a stronger estimate than parent tree size, the partial estimate for parent tree size goes negative. This doesn't mean that parent tree size is negatively correlated with child survivability, however. It is likely due to parent tree size being strongly multicollinear with crossover point depth.

Also note the high selectivity when the net change in subtree size equals 0, as shown in the graphs. This is due to subtree crossovers which only swap a terminal for an identical terminal, effectively reproducing the parents into the population.

## 9.4 A More General Theory of Tree Growth

In genetic programming, all nodes are equal, but some are more equal than others. Removing some subtrees results in dramatic changes to an individual, while removing other subtrees results in minor changes, or in the case of inviable code, inconsequential changes.

One strong predictor of node "importance" is its depth within the tree. The reason for this is straightforward: an s-expression is evaluated starting at the root, and the return value is obtained from the root. Tree nodes near the root often have a dramatic effect on the s-expression's operation, whereas tree nodes far away from the root have proportionally less effect, because their return values are decimated or they are rarely chosen from among a web of ancestor nodes' if-then constructs. Thus changing a tree node near the root can dramatically modify an individual, but changing a tree node far from the root has a correspondingly lower likelihood of modifying the individual by much.

Ordinarily this might be considered a good feature of genetic programming. The principle of *causality* [Rechenberg 1973; Rosca and Ballard 1995] argues that for evolutionary computation it is good to have the magnitude of genotypical change positively correlated with the magnitude in phenotypical (and fitness) change. Since deep nodes often root small subtrees, depth seems likely to be related with the amount of genotypical change. [Igel and Chellapilla 1999] analyzed causality in genetic programming both with the depth of the crossover point and with a metric of difference between the inserted and swapped out trees (called the *edit distance*), and found that while depth had a strong causal relationship in an individual, the relationship for edit distance was more tenuous.

However, this relationship between node depth and survivability can also explain tree bloat. Note that causality only relates the *magnitude* of change, not its direction (for better or for worse). But it is generally believed that in most common forms of genetic programming, modifying the individual is destructive most of the time (see [Teller 1996; Nordin and Banzhaf 1995]). If a decrease in node depth decreases the magnitude of change, and usually change is for the worse, then the expected survivability of a child is better if deep nodes are chosen for crossover or mutation points.

Such a bias towards node depth might have two effects on tree growth. First, it can promote larger parents, which have deeper crossover points. These larger parents in turn tend to produce larger children. Second, deeper-rooted removed subtrees are more likely to be small, but the inserted subtrees have no such size bias (a generalization of the removal bias theory).

### 9.4.1 Predicted Effects

One expected result of this theory is the emergence of *pseudoinviable code*, subtree areas for which crossover has a very low (but nonzero) probability of effecting an individual in a significant ways. The deeper a node is in the tree, the higher likelihood its rooted subtree will be forced into pseudoinviable code by its parents and ancestors. Since crossover in a pseudoinviable code region is unlikely to modify the individual by much, it is at one end of the gamut of strength of relationship between tree depth and survivability. Further, large trees would be expected to have correspondingly larger amounts of pseudoinviable code, just as they are expected to have larger amounts of inviable code. Both [Blickle 1996] and [Nordin, Francone, and Banzhaf 1995] noted the possible existence of pseudoinviable code. [Smith and Harries 1998] have recently performed experiments suggesting a strong relationship between pseudoinviable code and code bloat. True inviable code may be viewed as simply the most extreme end of the pseudoinviable spectrum.

Other effects might be explained by this theory. [McPhee and Miller 1995]'s inc-dec and [Smith and Harries 1998]'s R2 experiments both present contrived but important examples of code growth due to unoptimized code, but *not* due to neutral crossover — rather, trees seem to be simply filling up with junk. As it turns out these experiments both appear to be arranged in such a way that deeper crossover points are less likely to cause destruction of the individual; however this effect deserves more examination.

The theory also suggests a reason why restricted 6-Bit Multiplexer was so much less likely to find a solution. 6-Bit Multiplexer is a simple domain with no formal suboptima, and so it benefits from an incremental strategy. Crossing over in deeper subtrees lowers the likelihood of changing an individual significantly. But one effect of restricting inviable subtree crossover is that subtrees closer to the root have a higher likelihood of being chosen, resulting in more radical semantic jumps; in effect, this underdampens the search.

Lastly, the Edge Encoding domain has some interesting bloating characteristics which deserve some mentioning. Edge Encoding has many forms of unoptimized code but only one unusual invalidator: a subtree which creates an NFA subgraph which trivially accepts all strings. Such subtrees are easy to create, as in (loop ... (start (accept $\epsilon$))). The presence of this subtree anywhere in an individual will guarantee that the individual will achieve exactly a 0.5 fitness score no matter what. Edge Encoding's invalidator is not causally dependent on depth; it can appear anywhere and invalidate the entire tree. But as Tomita Language #5 showed (in Figure 6.5) strong bloating would still occur even when it was unlikely that individuals with such an invalidator could live long (because a 0.5 fitness score was too low).

## 9.5 Summary

Previous theories of code growth have suggested that the phenomenon is due to the propagation of inviable code. But the experiments in this chapter contradict this prevailing wisdom. After denying inviable code the ability to propagate, code growth actually increases. Further, the uniform distribution of node invalidators throughout the evolutionary run suggests that in fact not only is code growth not due to inviable code, but that it's the other way around. An increase in inviable code is an expected natural result of code growth.

Statistics gathered from the experimental work in this study suggest a more general explanation of code bloat: the depth of the modification point chosen in the parent is strongly related to the survivability of the child. This bias in tree depth can result in a preference for larger parents (which beget larger children) and also a preference for smaller removed subtrees (with no corresponding preference for the size of replacing subtrees).

# Chapter 10

# Conclusion

Genetic programming is a promising technique, but bloat is currently presenting a serious hindrance to GP's scalability. This thesis introduced genetic programming and evolutionary computation, illustrated the application of genetic programming to difficult problems, and addressed current issues in scaling genetic programming to tackle such problems in the future.

The primary challenge to scaling genetic programming is dealing with its strong tendency to bloat. Uncontrolled growth in variable-length genomes is not restricted to genetic programming, though GP has the lion's share of literature on the subject. Bloating is significantly hampering GP's ability to search for long periods of time. With the option of long runtimes closed off, the only realistic way to increase the number of individuals tested is to raise the population size. This effectively moves GP in the direction of random search, assuming enough memory is even available to do so. Genetic programming has of late been applied to some very difficult problems; but unrestrained bloating puts further its scalability in question.

The causes of genome growth can come from features inherent in the search procedure, and also from poor decisions on the part of the experimenter. The choice of breeding strategy has a significant effect on getting the best fitness for the smallest tree sizes: in general, subtree crossover favors large populations and short run lengths, whereas subtree mutation is superior for longer runs with smaller populations. While crossover in general bloats more rapidly than subtree mutation, ill-considered choices in function sets and the widespread use of the GROW tree-generation algorithm can also exhibit bloating effects special to mutation and independent of any selection pressure. Traditional standard GP methods must counter this in an ad-hoc fashion by enforcing arbitrary depth limits on initial tree generation and subtree mutation, biasing the search space with near-full tree structures.

The standard explanations for bloat center around inviable code areas, wherein modification cannot possibly change the fitness of an individual. Subtree growth in these areas, the traditional reasoning goes, is what drives tree growth as a whole in the population. However, as this thesis has shown, denying crossover in these regions (thus depriving them of the ability to grow) does not stunt tree growth in the population; rather, it actually increases it. Furthermore, the uniform distribution of invalidator structures in trees suggests that the more likely hypothesis is that bloat is causing the rise of inviable code areas, and not the other way around.

Exploratory data analysis suggests that, inviable code or no, subtree crossover retains a strong bias towards deeper crossover points. This bias can drive tree growth in two ways. First, it prefers larger trees, which have more deep nodes to choose from. Second, deeper nodes generally form the root of smaller subtrees, but no similar bias exists for the size of replacing subtrees. These results beg a serious reexamination of intron-based bloat-control techniques and the real reasons behind their success at slowing code growth.

**Contributions** The contributions of this thesis to the body of GP work are as follows:

- It presents a very large and comprehensive comparison of the common ways genetic programming can modify candidate programs to create new ones (subtree crossover and subtree mutation). This comparison covers four problem domains and two hundred common parameter settings, and reveals those areas where each breeding technique yields higher fitness and lower bloat. These areas are problem-specific, but do follow interesting trends. At 3.3 billion individual evaluations, this is the largest single comparison experiment in the genetic programming literature.

- It formally analyzes (GROW), the traditional algorithm used in genetic programming to create and mutate the trees which form GP individuals. The results of this analysis show that the GROW algorithm has very serious bloating characteristics, necessitating ad-hoc techniques to control the size of its generated trees. The thesis shows conditions under which subtree mutation can cause bloating in the population even with no driving selective force. It then introduces and analyzes two new algorithms, PTC1 and PTC2, which give the user control over tree size (which can counter bloat) and uniform tree node distribution.

- It gives data results which directly contradict the prevailing theories explaining the causes of bloat in genetic programming. This is the first such data of its kind. Supported by this and other data, the thesis proposes a more general theory of bloat applicable to most common genetic programming domains. These results shed light on why existing techniques work (or don't), giving further insight to the search for a cure for bloat.

Additionally, this thesis has two other items important to the GP community:

- It presents two case studies in genetic programming, both with nontrivial and difficult real-world problems to solve. In one of these case studies, evolving virtual soccer robot programs, the results of the evolutionary computation run performed surprisingly well against hand-coded programs in an international soccer robotics competition. Both case studies illustrate the problem of bloat in complex problem domains.

- It introduces a new genetic programming and evolutionary computation system, ECJ, designed to make possible complex genetic programming experiments, including ones discussed later in this thesis. ECJ provides a modular experimental framework and a rich set of built-in evolutionary computation features. As of July 2000, ECJ is the most sophisticated genetic programming system available in the public domain.

## 10.1 Future Work

This thesis touched on a variety of topics in genetic programming, evolutionary computation, and bloat, and so there are many areas to chose from for future directions of research. I list five below.

**Code Growth Limitation Techniques** If the reason for code growth is more general than intron explanations would suggest, then feature-independent techniques such as parsimony pressure may be the best overall solutions to the problem. There remain interesting parsimony pressure approaches to try that have not been tested yet. For example, an important feature of genetic programming recombination is that only one point is picked per individual. This means that as individuals grow, the likelihood that any given point $x$ will be chosen as a recombination site approaches zero. However, if it were possible to pick more than one point as a recombination site, a point's probability of selection can be fixed to a value invariant over individual size. This would lower the bias towards deep individuals in protecting against destructive crossover events.

Another interesting approach to try, suggested by [Delwiche 2000] is to apply parsimony pressure techniques selectively, either periodically in a cataclysmic fashion, or to only certain subpopulations using an island model. The idea is to allow parsimony pressure to effect population size but not dominate it. One interesting approach like this is to allow subpopulations to evolve asynchronously. Since bloat slows the rate of evaluation and breeding, smaller-sized subpopulations might be able to evolve more rapidly than large ones, displacing them.

**Picking the Right Breeding Operators** GP has of late branched out to use a wider range of breeding operators (for example, [Chellapilla 1997; Francone *et al.* 1999; Langdon 1999]), and a comprehensive comparison of these operators is needed to determine how effectively they search while keeping unnecessary code growth to a minimum. [Chellapilla 1997] suggests combining a large number of mutation operators for better results.

Much of Chapter 9 grew out of attempts to adaptively pick between subtree crossover and subtree mutation during the course of the run. These adaptivity experiments did not yield good results for such adaptation, but one discovery was that usually a half-half mix of mutation and crossover would perform as well as crossover in those conditions which favor crossover over mutation, and also as well as mutation when conditions favored mutation over crossover. Such results deserve further exploration.

**Determining When to Give Up** At some point in an evolutionary run, the minor increases gained from continuing on are not worth the extra effort expended to get them: it's better instead to give up and try again from scratch. This nominally occurs at generation $n$ when, in the probability distribution of best individuals for all runs for a given problem, the expected mean at generation $2n$ is worse than the expected $\frac{2}{3}$ percentile at generation $n$. At this point, it becomes advantageous on average to perform two $n$-generation runs rather than a single run $2n$ generations long.

This leads to a useful definition of problem difficulty. As the critical point $n$ approaches 0, the problem becomes more and more difficult for the search technique to solve in the sense that the technique outperforms random search by a smaller and smaller margin. When $n$ is 0, the technique is no better than random search on this problem.

Bloat figures prominently here. The previous computation assumes that all generations take the same amount of time to run. But if later generations bloat, and this bloat effects their breeding and evaluation time, then the meaning of "$n$ generations" becomes meaningless. In this case, the break-even point must be redefined to mean $n$ units of time rather than $n$ generations. Thus the effects of bloat must be estimated in determining the difficulty of the problem for a given technique.

This metric can have two uses. First, it gauges the effectiveness of a particular search technique at solving a given problem. Second, it might help determine at run time when to stop a run and start over, assuming sufficient run statistics exist (approximate population means and variances over several runs). Even if these statistics are unknown, the distribution of fitnesses within the population might possibly serve as (albeit poor) estimates.

**Estimating Tree Growth Under Subtree Mutation** While Section 8.3 suggested that it is unlikely exact solutions can be found for selection-independent tree growth due to subtree mutation, nonetheless it is possible that such growth can be estimated, if very roughly.

One example of this is binary trees. There are two obvious extremes in binary trees: full trees, and "snakes", that is, trees for which every nonterminal has at least one terminal as a child. For both of these extremes, it is possible to approximate the expected size $S$ to which populations will grow after repeated application of subtree mutation with an expected generated subtree size of $E_{tree}$.

For a "snake" of depth $D$, the longest path in such a tree consists of $D-1$ nonterminal nodes and one terminal node. The sum of subtree sizes rooted at each of these nodes, starting from the terminal and working up to the root, is $\sum_{i=0}^{D-1} 2i-1 = D^2$. The sum of subtree sizes in the remaining $D-1$ terminals is of course $D-1$. The total number of nodes in the tree is $2D-1$. Thus the average size of a subtree removed during mutation is $\frac{D^2+D-1}{2D-1}$. For subtree mutation to on average increase tree size, $E_{tree}$ must be smaller than this value. Solving $E_{tree} = \frac{D^2+D-1}{2D-1}$ for $D$ yields $D = \frac{1}{2}(2E_{tree} - 1 + \sqrt{4E_{tree}{}^2 - 8E_{tree} + 5}) < 2E_{tree}$. As $E_{tree}$ approaches infinity, $D$ rapidly approaches $2E_{tree}$ and so the tree size $S$, below which subtree mutation on average increases tree size, is approximately $S = 2D - 1 = 4E_{tree} - 1$.

In a full binary tree of depth $D$, there are $2^D - 1$ nodes. At each depth $d$, there are $2^d$ nodes, and each node roots a subtree of size $2^{D-d} - 1$. Thus the average tree size is $\frac{\sum_{d=0}^{D-1} 2^d(2^{D-d}-1)}{2^D-1} = \frac{(D-1)2^D-1}{2^D-1}$. For subtree mutation to on average increase tree size, $E_{tree}$ must be smaller than this value. There is no solution for $D$ in $E_{tree} = \frac{(D-1)2^D-1}{2^D-1}$, but as $D$ approaches infinity, $E_{tree}$ rapidly approaches $D$. Since there are $S = 2^D - 1$ nodes, at the limit with subtree mutation of size $E_{tree}$, the break-even point for full trees is approximately $S = 2^{E_{tree}} - 1$.

The typical binary tree lies somewhere between these two extremes. This suggests, but does not prove, that with subtree mutation with sizes of $E_{tree} > 4$, average tree size in the population will rise to $4E_{tree} - 1$ minimum, and possibly as much as $2^{E_{tree}} - 1$. What a typical population of binary trees actually grows to is yet to be determined, as is the growth patterns of more general tree populations, though such estimates can probably only be determined empirically.

**Biological Implications** Bloat isn't just an issue of abstract interest to genetic programming, or even more generally to variable-length optimization. It is also of interest to evolutionary genetics.

For many eukaryotes, including humans, much of the DNA appears to be "junk". In humans, less than 3% of the genome is actually used in coding for genes [Strachan 1992], the remainder being extragenic DNA, introns, pseudogenes, etc. While there do exist eukaryotes with tiny genomes — the the chlorarachniophyte nucleomorph genome has the smallest nuclear genome known (380K base pairs) [Palmer and Delwiche 1996] — nonetheless the trend seems to favor large, low-information chromosomes in eukaryotic organisms. In contrast, in most prokaryotes rather little of the genome is wasted. Prokaryotes are often so efficient that several different genes may share the same space, overlapping each other and using frame shifts to squeeze the maximum amount of information out of the fewest DNA nucleotides.

It is commonly held that prokaryotes have highly compressed DNA because to survive they must be able to divide quickly, and the larger the chromosome, the longer it takes to make a copy of it (see [Brown 1992], p. 296). Thus there is a selection pressure towards genome parsimony. But eukaryotes may not have a similar pressure in general. This might explain why prokaryotes don't have large strings of DNA, but it doesn't adequately explain why DNA bloats in eukaryotes in the first place. To a large extent, the reason behind eukaryotic bloating is not well understood.

One hypothesis suggested by bloat in GP is that, in the absence of some countering pressure, genomes grow because inserting new genic material is less likely to damage the individual than deleting material would. Bulk insertion of nucleotides affects only genes unfortunate enough to straddle the insertion point, whereas bulk deletion may damage or eliminate any genes located within the deleted region. Such insertion may also be important for a species to adapt through mutation. [Ohno 1970] introduced the popular notion that gene duplication, presumably caused by transposition and insertion events, permits a gene to undergo mutation without killing the individual, because it now has a backup gene filling in for it.

Real DNA has different dynamics than genetic programming. However, the similarities are tantalizing, and it is worthwhile considering if a generalized explanation for growth in variable-length representations can explain both abstract domains and ones in the real world.

Appendix A

# A Code Bloat Bestiary

[Nordin, Francone, and Banzhaf 1996] offered five flavors of introns: inviable code with regard to the problem domain, inviable code with regard to the given fitness cases, unoptimized code with regard to the problem domain, unoptimized code with regard to the given fitness cases, and code which "approximates" introns. The last flavor works by making it either highly unlikely that crossover will have an effect, or causes any crossover effect to be very small, or both. However, these five flavors in fact toss together three different dimensions of classifying code structures:

- **Level of Inactivity** There are three basic levels: either the code does in fact affect the individual (it is active), the code does not affect the individual (it is unoptimized), or *no* code could affect the individual in that location (it is inviable).
- **Degree of Ineffectiveness** There are five general possibilities here: either no code could possibly affect the individual (it is inviable), no code could affect the individual very much, there is a low probability of any given code affecting the individual at all, there is a low probability of any given code affecting the individual very much, or there is a significant probability of affecting the individual significantly. I group the middle three possibilities under the aegis of *pseudoinviable code*. Note that unoptimized code can also be pseudoinviable.
- **Domain of Ineffectiveness** I list four possible areas where code modification is ineffective. Modifying the code might not be able to: change the operation of the individual, change any conceivable fitness case, change any provided fitness case, or change the overall fitness assessment. These small differences are surprisingly important: in Multiplexer for example, many crossover events change the individual's boolean function dramatically but not its overall fitness. And in Symbolic Regression, there exist crossover events which change one fitness case a small amount, but this change is lost in the overall fitness assessment due to the limited precision of the floating-point type used.

As can be surmised, there are a wide range of interesting structures or events which may prevent individuals from changing in fitness, or which have been implicated in code growth, or both. What follows are four general categories of these structures and events: inviable code, inoperative code, pseudointrons, and important reproductive events. This catalog is hardly complete; but it illustrates the most common cases. For many of these cases, several permutations of the classifications listed above may be applicable.

## A.1 Inviable Code

Inviable code is code which cannot contribute to an individual, no matter what it consists of. Thus modification of this code, through subtree crossover or mutation, cannot change the individual's fitness assessment in any way. Two common reasons for inviable code are: because it is never executed; or because its return value is ignored. Inviable code is often due to the presence of an *invalidator*, a chunk of code responsible for nullifying the inviable code. Here are common forms of inviable code:

- **Sibling Interference** This is the most common inviable code form, where an invalidator forces a sibling's return value to be unused or causes it to be left unexecuted. Some examples: (∗ (- x x) *inviable*), (or (or d0 (not d0)) *inviable*), (if (and d1 (not d1)) *inviable always-executed*), (if *inviable* d0 d0).
- **Co-inviable Code** This unusual special case of sibling-interference inviable code deserves mentioning. Here two invalidators effectively force each other to be inviable code. For example in (∗ (- x x) (rlog (% x x))), both (- x x) and (rlog (% x x)) return 0.
- **Propagated Inviable Code** A single invalidator sets off a chain of inviable-code events. For example in (∗ *inviable2* (∗ (- x x) *inviable1*)), *inviable1* is nullified by the invalidator (- x x) which returns 0, and *inviable2* is in turn nullified by the propagated invalidator (∗ (- x x) *inviable1*) which also returns 0. (- x x) is called a *base invalidator*.
- **Denial of Action** In some domains (such as Artificial Ant), code can be made effectively inviable by tying its viability to some event in the evaluation period which never occurs. For example, in the code snippet (if-food-ahead *inviable always-executed*), the inviable code *could* be executed if there was ever food ahead, but as it turns out every time this food test was evaluated, there was never food ahead.
- **Delay** Inviable code can occur because by the time it is reached it is unneeded, typically because the individual's evaluation has ended. Such inviable code is often highly dependent on execution order in the tree. For example in the Artificial Ant problem, the ant is only permitted to move 400 times; if the tree consists of more than 400 move commands, then further nodes in the tree are ignored.
- **Remote Causes** Inviable code may occur because an earlier-executed remote invalidator caused the code to be inviable when it is reached later in execution. This occurs with domains which read and write to a global memory. For example, a subtree executed early on may execute (write-to-b 1); a later-executed code snippet might read (if (= 1 (read-from-b)) *always-executed inviable*).
- **Decimation** Parents and parents' siblings conspire to eliminate the effect of an inviable subtree's return value by dropping it below the effective precision of the computer system's data type. For example: (rlog (+ (sin *inviable*) (exp (exp 6)))) will return the same value (about 403.43) regardless of the value of the inviable subtree, when using the Java double type.
- **Boundary Conditions** Parents and parents' siblings conspire to eliminate the effect of an inviable subtree's return value by moving it beyond some boundary condition. For example, if my-log were defined to return 0 for any negative number, (my-log (- (sin *inviable*) 1.0)) would always return 0. Another example occurs infrequently in the Multiplexer domains: (and (not a0) (and *inviable* a0)).
- **Redundant Tests** Nested if-statements make it impossible for a subtree to ever be evaluated. For example in the Artificial Ant domain, (if-food-ahead *evaluated-when-true* (if-food-ahead *inviable evaluated-when-false*))
- **Uncalled Automatically Defined Functions and Macros** If an automatically-defined function or macro is not reachable in evaluating the individual, then that ADF branch is inviable code.

## A.2 Unoptimized Code

Unoptimized code is code which is excessively redundant in form and can be intelligently cleaned up without changing the operation of the individual. However, arbitrary modification of unoptimized code can change the individual's operation. Thus unoptimized code is viable.

- **Mutually Exclusive Parents** A chain of parents which together form the identity function on the data passed through them and eliminate each others' side effects when executed. For example, (not (not *foo*)) or (turn-left-then (turn-right-then *foo*)).

- **Redundant Subtrees** Two subtrees return or perform exactly the same thing, and so can be condensed into a single subtree. For example, (and (or d1 d2) (or d2 d1)).

- **Mutually Exclusive Subtrees** Two subtrees cancel each others' side effects. As in (prog2 (prog2 turn-left turn-left) (prog2 turn-right turn-right)).

- **Identity Conditions** The return value of a subtree is worthless because it converts its parent into the identity function and eliminates any side effects. Examples: (+ *foo* 0), (∗ *foo* 1), (set-position 3 (get-position 3)), or (and *foo bar*), where *bar* always returns true.

- **Other Unoptimized Code** A variety of other messy code structures can be cleaned up through copy propagation, constant folding, common subexpression elimination, and other code optimization procedures (for details, see [Aho, Sethi, and Ullman 1988]). For example: (+ 4.2 (+ 3.5 *foo*)).

## A.3 Pseudointrons

Pseudointrons are areas of code which are nearly unoptimized code or nearly inviable code within reasonable probability.

- **Pseudo-unoptimized Code** Code which has an effect, but which is minimal. For example, (+ *foo* 0.000000000000001).

- **Pseudoinviable Code** Regions where there is a low probability that any replacing code can change the fitness of the individual in any significant way. For example, (* *pseudoinviable* 0.000000000000001).

## A.4 Reproductive Events

These events round out other possible reasons for individuals to produce children whose fitness is identical, or nearly identical, to themselves.

- **Serendipitous Change** A non-pseudointron, non-inviable subtree is replaced with another that results in no change (or in inconsequential change): either the genotype is identical (very common when terminals are swapped), the phenotype or effective function is identical, the overall fitness is identical, or the fitness does not change significantly.

- **Failure to Meet Constraints** Because the breeding procedure is unable to form valid children, their parents are simply replicated instead. This happens, for example, when after some number of attempts, subtree crossover is unable to produce a child smaller than the maximum tree depth. Failure to meet constraints results in an interesting phenomenon: *protected code*. Protected code are subtrees so deep in the tree that it is difficult to internally modify the code (through crossover or mutation) and create a tree which meets depth constraints.

- **Direct Reproduction** Reproduction is added to the suite of breeding operators. In the traditional Koza formulation, reproduction produces on average 5.26% of a new population (it is chosen as an operator 10% of the time but only produces one child as opposed to subtree crossover's two children).

## Appendix B

# ECJ: Evolutionary Computation in Java

This appendix gives an overview of ECJ, the evolutionary computation research system developed to do the experiments described in Chapters 6 and 9. Other experimental work (in Chapters 5 and 7) was performed using the *lil-gp* genetic programming system [Zongker and Punch 1996].

ECJ is a Java-based evolutionary computation research system which runs on Unix, Windows, and MacOS. While ECJ is built for evolutionary computation in general, it comes with a comprehensive suite of libraries for doing Java-based genetic programming. As of its introduction, ECJ 4 is the most full-featured genetic programming distribution publically available. It provides:

- multithreaded and multiprocess evolution
- both steady-state and generational evolutionary computation
- multiple populations, inter-population exchanges, and coevolutionary hooks
- set-based strongly-typed genetic programming
- automatically defined functions and macros
- ephemeral random constants
- checkpointing, logging, and reading populations from files
- hooks for evolution with multi-objective fitness
- a highly flexible breeding architecture with six selection methods, twelve breeding operators, and four tree-generation algorithms
- very customizable individuals with multiple trees
- six pre-done problem domains (Artificial Ant, Multiplexer, Symbolic Regression, Parity, Lawnmower, Edge Encoding)
- abstractions for many evolutionary computation approaches, not just genetic programming

ECJ was designed to make many complex genetic programming experiments possible through an easily extensible design and a large suite of features. Most popular genetic programming systems are either small and custom-purpose, have few modern genetic programming features, or are difficult to extend.

## B.1 Why Java?

Because of its promise of distributed computation and its portable bytecode, Java is an increasingly popular language for doing genetic programming. There are presently ten Java-based genetic programming systems available, the most well known being: gpsys [Qureshi 1997], DJBGP [Lukschandl *et al.* 2000], and DGP [Chong 1998; Chong and Langdon 1999]. DJBGP and DGP were designed for special-purpose genetic programming work, and no existing system is easily extensible or generalizable.

ECJ's design criteria were primarily extensibility, portability, and a rich set of features, but some serious attention is also paid to keeping the system fast.

**Extensibility** Leveraging Java's ability to load classes arbitrarily at runtime, ECJ determines nearly every class, parameter, thread, and process at runtime from a set of user-provided parameter files which guide its operation. These parameter files tell ECJ what problem to attack, what technique to use against it, and how to report the results. Since they are determined at runtime, ECJ's classes have relatively few dependencies between objects; arbitrary classes may be substituted independent of each other. This makes ECJ easy to extend and maintain.

**Portability and Replicability Across Platforms** Java is designed to be portable. Because ECJ is written for Java 1.1, it runs with little or no modification across a wide range of platforms. A running ECJ process can even be serialized out to a file, moved to a foreign platform, and continue to run. But more importantly, because Java has formalized data types, math operations, and input/output libraries, ECJ can guarantee that its run results will be identical regardless of the platform, excepting for errors in the Java implementation. For scientific experiments, guaranteed replicability is a highly desirable attribute, and something that is not met by existing C, C++, or Lisp-based evolutionary computation systems.

**Speed** It is easy to be inefficient in Java, and ECJ takes significant pains not to be so. ECJ tries to do the minimum necessary allocation during evaluation and breeding. Though ECJ takes full advantage of Java's multithreaded environment, the ECJ distribution avoids situations which require Java's mutex synchronization, and rarely uses Java's heavily synchronized utility objects (such as java.util.Vector). ECJ also avoids Java's slow exception-handling system whenever possible. Lastly, ECJ's implementation of the Mersenne Twister random number generator produces random integers at three times the rate of the Knuth subtractive generator used in java.util.Random.

Despite its Java foundation, ECJ achieves speeds roughly comparable to lil-gp, the most popular C/C++-based genetic programming system. Because of differences in the internal dynamics of the systems, it is difficult to perform an exact comparison. But my experience has shown that ECJ is in general slower than lil-gp during breeding, but faster in evaluation. This means that for rapidly-evaluated problems like Symbolic Regression, ECJ is somewhat slower than lil-gp, but for slowly-evaluated problems like 11-Multiplexer, ECJ is generally faster than lil-gp. Since the trend in genetic programming is towards more complex evaluations, this seems to be a reasonable tradeoff.

**Memory Model Safety** Unlike C/C++, Java is garbage collected and provides run-time array length checking. Genetic programming generates a *very* large number of short-lived objects and arrays, and its complex breeding, node selection, and node-generation mechanisms make it easy to lose track of pointers or walk off the end of arrays. Since it uses Java, ECJ doesn't have to worry about these issues.

**Disadvantages** ECJ's primary disadvantage is memory consumption. Because it uses trees instead of arrays, and because Java has no static allocation, ECJ must allocate approximately 30 bytes per node. This is larger than

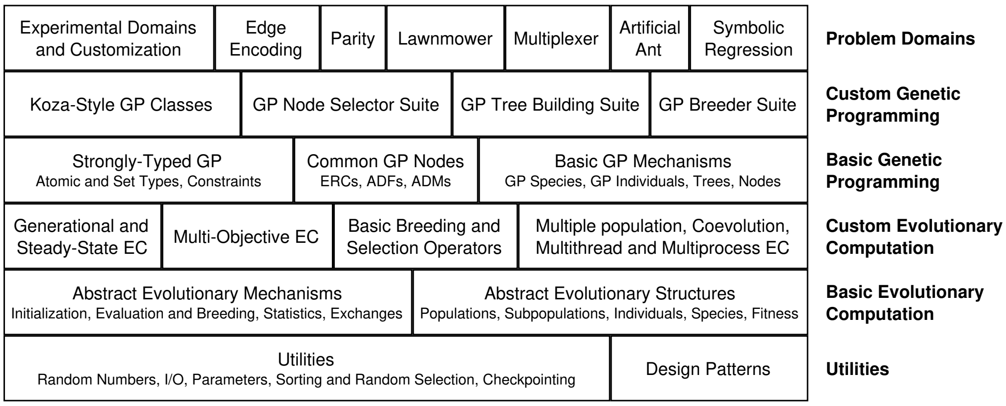



Figure B.1: Code Layers in ECJ

most non-Java genetic programming systems. For example, lil-gp uses 4 bytes per node, excepting ephemeral random constants which cost at least 17 bytes.

## B.2 Overview of ECJ

ECJ 4 has 228 classes and interfaces divided into roughly six layers of code abstraction: the utility layer, the basic and custom evolutionary computation layers, the basic and custom genetic programming layers, and the problem domain layer. Classes in these layers are highly modular; most can be removed and replaced with few dependencies from other classes. Figure B.1 illustrates the six code layers. Figures B.5 and B.6 show the core evolutionary computation and genetic programming classes respectively.

### B.2.1 The Utility Layer

The Utility Layer provides utility services and design patterns independent of evolutionary computation. This includes: Mersenne Twister random number generators, sorting and randomized selection algorithms, encoding and decoding data types to readable text, version information, threadsafe logging, a parameter database, and a checkpointing facility using Java's serialization mechanism.

Genetic programming uses a very large number of random numbers, and poor random number generation has been cited as the culprit behind many anomalous results in the field [Daida *et al.* 1997]. Recognizing this, ECJ sports the fastest and most complete Java implementation of Mersenne Twister, a popular and exceptionally high-quality random number generator [Matsumoto and Nishimura 1998]. Mersenne Twister has a period of length $2^{19937}-1$, is 623-dimensionally equidistributed, and runs faster than most common generators (including C's rand() and Java's java.util.Random).

With very few exceptions, all of ECJ's non-utility layer classes and their settings are determined at runtime from a hierarchy of user-provided parameter files loaded and stored in ECJ's parameter database. When ECJ loads an object determined from the database parameters, it gives the object a chance to set itself up. During this setup process the object determines its settings from the parameter database. This often includes loading additional object instances, which are in turn given a chance to set themselves up.

ECJ has four design patterns for objects which set themselves up in different ways. *Singletons* are objects for which only one instance exists for a given class. They are set up individually and stored globally, and are generally immutable once set up. For example, ECJ's evaluator object is a singleton. Similarly, *cliques* are small suites of unique objects of the same class which are each set up separately. Cliques are also global and usually immutable. The set of "types" in strongly-typed genetic programming is an example of a clique.

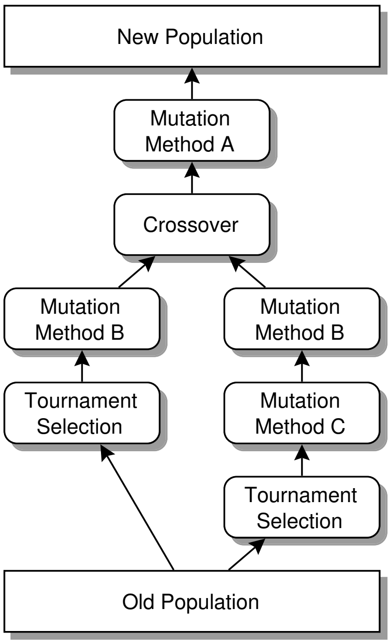



Figure B.2: An ECJ Breeding Pipeline

*Prototypes* and *groups* define objects for which a single prototypical instance is loaded and set up; then a great many instances are formed by cloning this prototypical instance. Most ECJ objects are prototypes or groups. For example, a genetic programming tree node is a prototype, and a population is a group.

ECJ begins by setting up the top-level singleton, the EvolutionState object, which triggers a cascade of setups until all classes in the system are loaded and set up. As EvolutionState is top-level object in the program, every other object in the program is reachable by tracing references from EvolutionState (except for the Evolve object, which exists solely to get the program up and running). Thus EvolutionState defines the entire current state of the evolutionary process at any time. Java's serialization facility writes out to disk an object and all objects reachable by it. Taking advantage of this, ECJ can write the entire state of the program out to a checkpoint file by serializing EvolutionState. This state can similarly be reloaded from the checkpoint file, resetting logs and streams as appropriate.

### B.2.2 The Basic and Custom Evolutionary Computation Layers

The Basic Evolutionary Computation Layer defines abstract evolutionary data types and mechanisms common to most evolutionary computation techniques. This includes abstract EvolutionState objects; initialization, evaluation, exchange, and breeding routines; problem domains; statistics loggers; populations, subpopulations, individuals, species, and fitnesses. The Custom Evolutionary Computation Layer defines specific usable versions of these abstract mechanisms for different kinds of evolutionary computation. This includes both generational and steady-state evolutionary computation, optimization with single and with multiple objectives, coevolution, multiple threading and multiprocess communication, and popular selection methods.

ECJ's basic unit of evolution is the *individual*. During evolution, each individual is assigned a *fitness*. Groups of individuals form a *subpopulation*; each subpopulation is assigned *species* which defines how individuals of the species are created and the constraints under which they may breed. Multiple subpopulations may have the same species. All subpopulations together form the *population*. In a typical non-coevolutionary ECJ run, the population will have only single subpopulation of individuals.

The EvolutionState object defines the high-level evolutionary loop. For generational evolutionary computa-

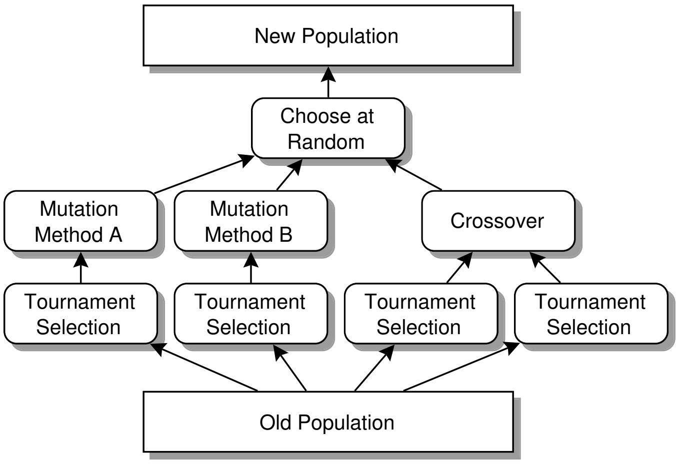



Figure B.3: An ECJ Breeding Pipeline with Subsidiary Pipelines Picked at Random

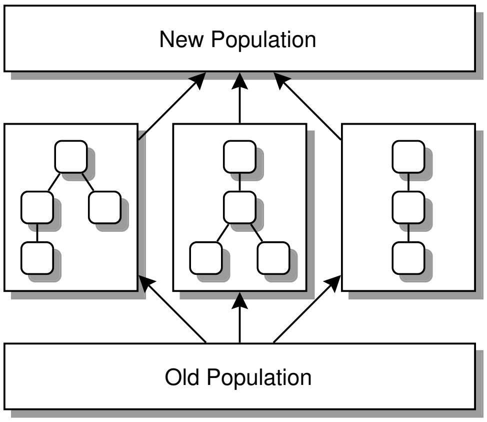



Figure B.4: Three Breeding Threads, Each with Its Own Breeding Pipeline, Breeding a New Population

tion, the loop is very simple. First the Initializer object is called to produce the population. Then the Evaluator is called to assess the fitness of individuals in the population. When evaluation is complete, the Breeder produces a new population bred from the original one. The new population replaces the old population, which is discarded. Then the Evaluator evaluates this new population, and it is in turn bred, and so on. At several points in the loop, the Statistics are called to report statistical results. Additionally, a user-defined Exchange object may be called before or after breeding to perform inter-subpopulation individual exchanges, or perhaps exchanges between simultaneous ECJ programs. Evolution ends when time runs out or when a sufficiently high-quality individual is discovered during an evaluation step. Before the program quits, the Finisher is called to do any necessary cleanup.

**Breeding Pipelines** Breeder uses a breeding mechanism known as *breeding pipelines*. Breeding pipelines are linked filters which pipe individuals from the previous generation, copy them, modify them, and add them to the new generation. These pipelines take the form of trees whose leaves are various selection methods picking individuals from an old population, and whose root produces the final individuals for the next generation population.

Breeding pipelines enforce a copy protocol which guarantees that members of the old population remain immutable. When using generational-style evolutionary computation, ECJ can take advantage of this protocol to spawn multiple breeding threads which operate without inefficient mutex locking. Each thread is fills a different section of the new population, using its own set of breeding pipelines.

Evaluator uses a similar mechanism to evaluate individuals in non-coevolutionary environments, spawning one or more threads, each of which evaluates a different chunk of the old population. Each thread uses its own copy of the Problem, which defines the domain against which to evaluate individuals. If the individuals can be evaluated independently of one another, then their fitnesses may be assigned by multiple threads without using mutex locking.

### B.2.3 The Basic and Custom Genetic Programming Layers

The Basic Genetic Programming Layer defines common data types for tree-based genetic programming. This includes genetic programming individuals, species, trees, and nodes, as well as ephemeral random constants (ERCs), and automatically-defined functions (ADFs) and macros (ADMs). This layer also provides types and constraints sufficient to implement set-based strongly-typed genetic programming. The Custom Genetic Programming Layer provides suites of mechanisms useful for most popular genetic programming techniques. One package in this layer includes Koza-style genetic programming algorithms sufficient to replicate most approaches used in [Koza 1992] and [Koza 1994], as well as those used by lil-gp. Other packages provide additional kinds of breeding operators, node-selection algorithms, and tree-generation algorithms from the literature, as well as a few unique to ECJ.

**The Structure of Genetic Programming Individuals** Like many Java-based genetic programming systems, ECJ stores its individuals' parse trees as actual trees, as opposed to the packed array approach popular in C/C++ systems. This is partly because Java's lack of static objects makes it difficult to store trees as arrays, and partly because packed arrays are very difficult to customize and also tough to code bug-free breeding and node-selection operators for. The disadvantage of ECJ's approach is that it consumes more memory, and tree copying and modification is inefficient, which makes breeding slow. On the other hand, this makes evaluation fast and extending the system intuitive.

ECJ's GPIndividuals each hold an array of one or more GPTrees. Each GPTree holds a tree made of various subclasses of GPNode, plus an index to a GPTreeConstraints object which defines the expected return type of the tree and the constraints that determine which other trees the tree may breed with. Each GPNode holds an array of zero or more children, a pointer to its parent in the tree (the root GPNode in the tree points to its owning GPTree), and an index to the node's particular GPNodeConstraints. GPNodeConstraints objects define the constraints placed upon node modification during breeding, including the return types and argument types of the nodes for strongly typed genetic programming, and the expected number of arguments to each node. GPNodes are usually evaluated recursively by calling a method in the root node, passing it a subclass of GPData which ferries data between parents and children.

### B.2.4 The Problem Domain Layer

The Problem Domain Layer holds packages defining various predefined problem domains, plus some auxiliary facilities for creating custom experiments. Six example problem domains come with ECJ 4:

**Symbolic Regression** Implements the Symbolic Regression problem, as defined in [Koza 1992], both with and without ephemeral random constants.

**Artificial Ant** Implements the Artificial Ant problem, as defined in [Koza 1992], using either the Santa Fe or Los Alamos trails.

**Multiplexer** Implements the 3-bit, 6-bit, and 11-bit Multiplexer problems, as defined in [Koza 1992], using the Sub-Machine Code technique [Poli, Page, and Langdon 1999] to speed evaluation by up to ten times.

**Lawnmower** Implements the Lawnmower problem, as defined in [Koza 1994], with any size lawn and with or without automatically-defined functions.

**Parity** Implements the Even- and Odd-Parity problems, as defined in [Koza 1994], with any number of bits from 2 to 31.

**Edge Encoding** Implements the Edge Encoding problems as described in this thesis.

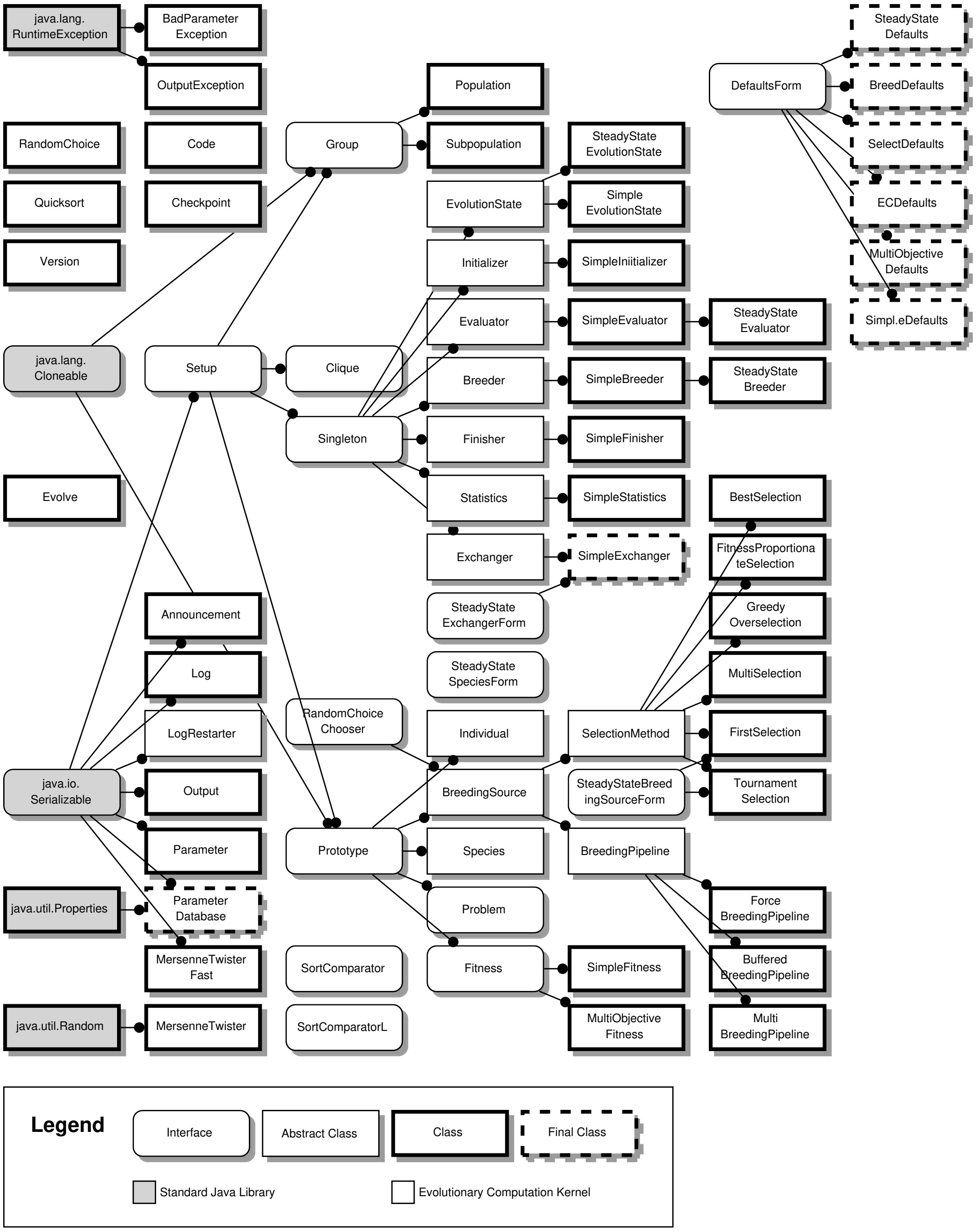



Figure B.5: ECJ Evolutionary Computation Kernel Class Hierarchy

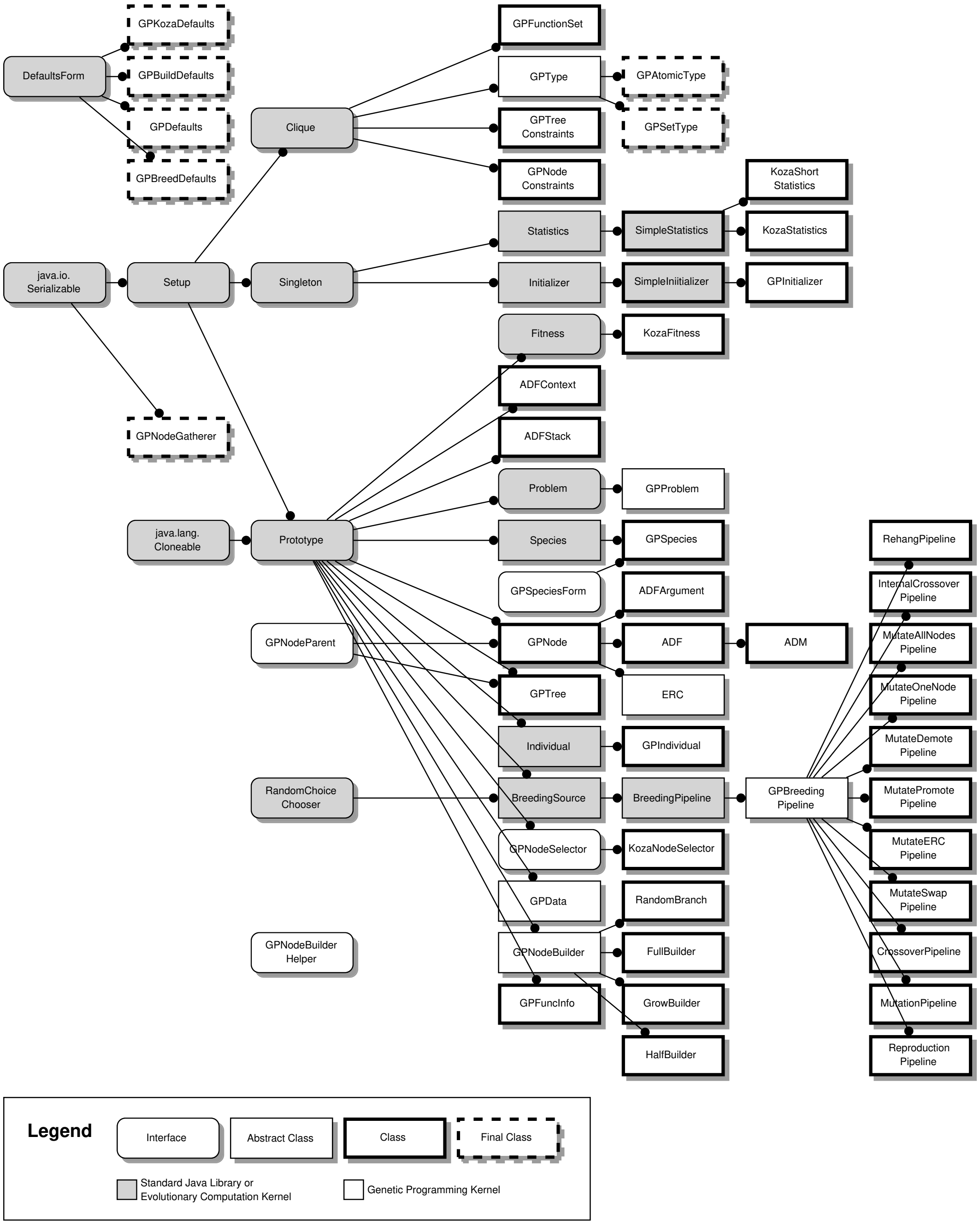



Figure B.6: ECJ Genetic Programming Extensions Class Hierarchy

## Appendix C

# Additional Tables and Figures

The following is the full collection of tables and figures discussed in Chapter 9; this chapter previously showed selections from this collection. There are three sets of figures and tables: graphs of general and inviable node statistics gathered at each generation of the runs; graphs of relationships between features gathered at selected generations during the runs; and tables showing regressions among these features.

The first set of figures display statistics gathered from eight kinds of evolutionary runs: 6-Bit Multiplexer, 11-Bit Multiplexer, and Symbolic Regression each with or without inviable code marking, and Symbolic Regression with two further counter-inviable code procedures: restrictions on semantically identical crossovers, and restrictions on fitness-identical parents. The other two sets of figures and tables omit Symbolic Regression's two further counter-inviable code procedures.

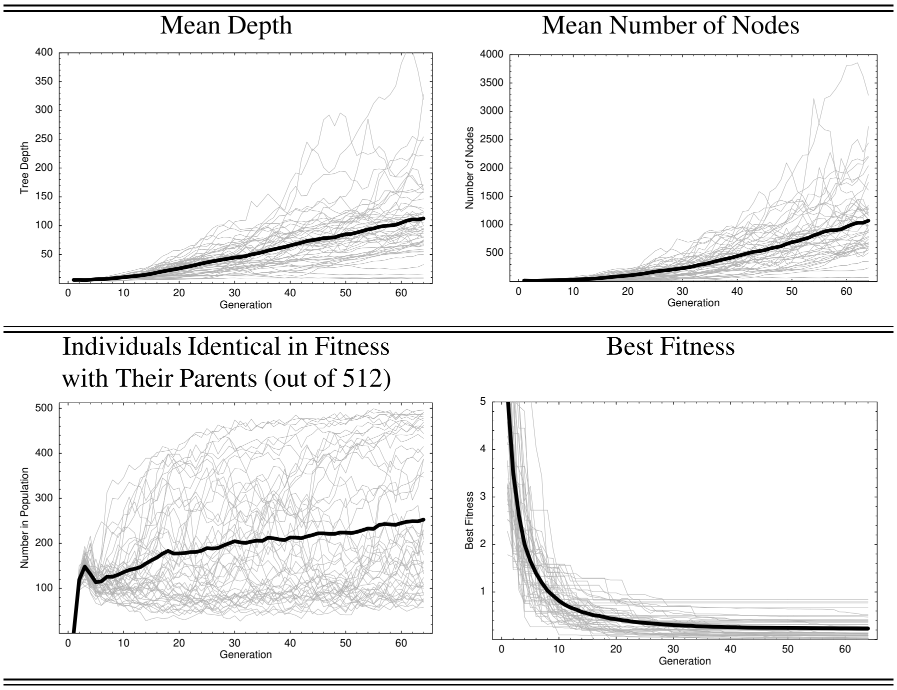



Figure C.1: General Data for Symbolic Regression Domain

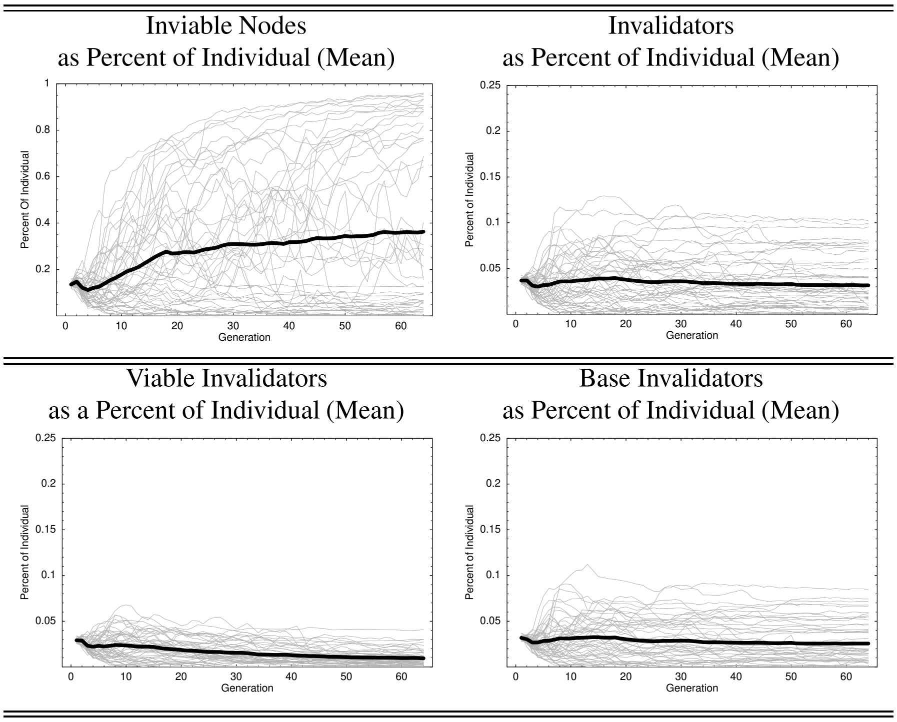



Figure C.2: Inviable Node Data for Symbolic Regression Domain

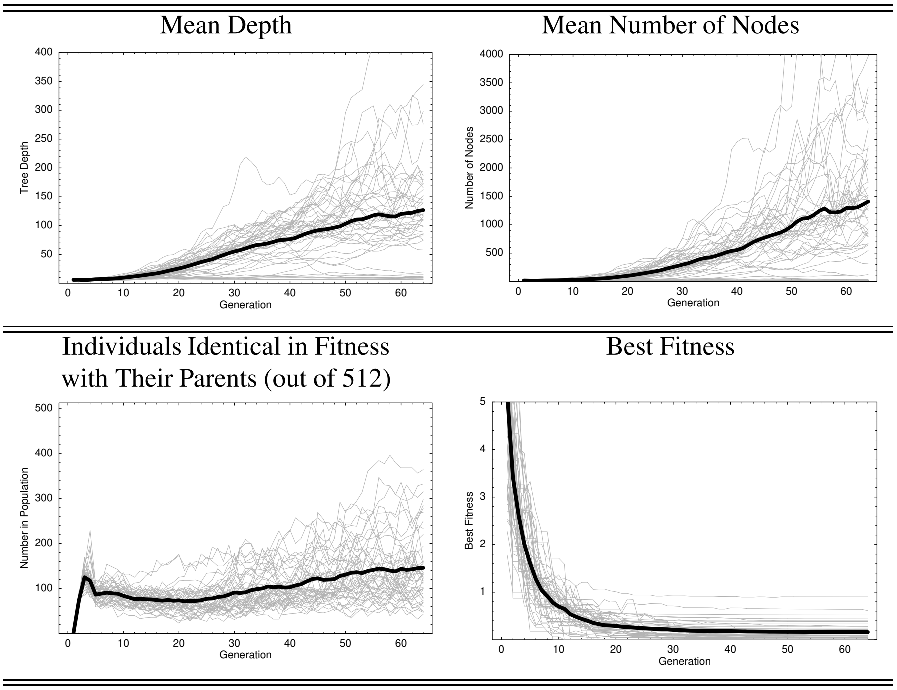



Figure C.3: General Data for Symbolic Regression Domain with Inviable Code Restricted

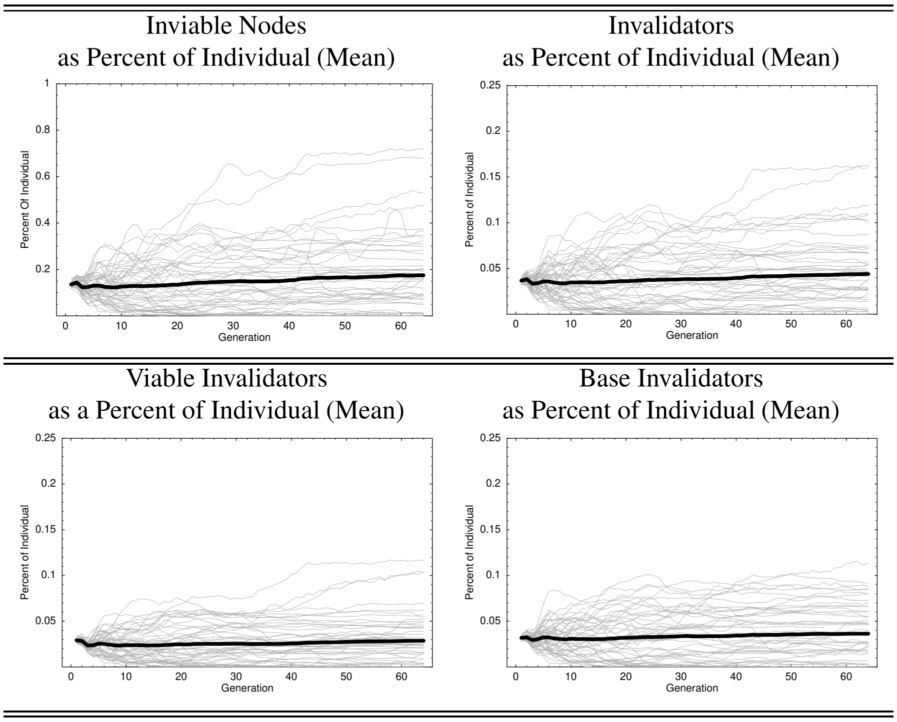



Figure C.4: Inviable Node Data for Symbolic Regression Domain with Inviable Code Restricted

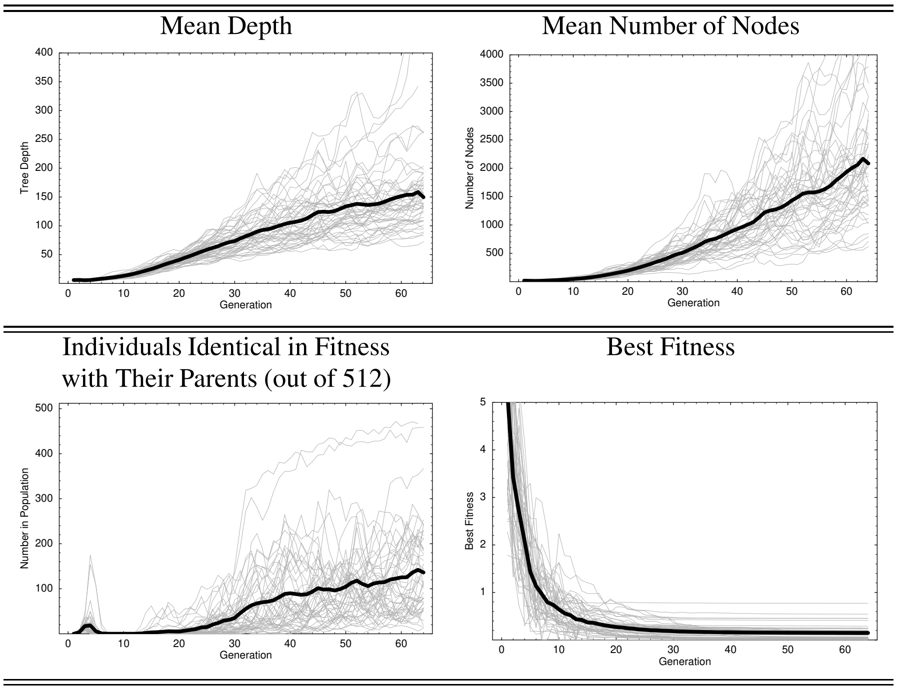



Figure C.5: General Data for Symbolic Regression Domain with Inviable Code and Semantically-Identical Crossovers Restricted

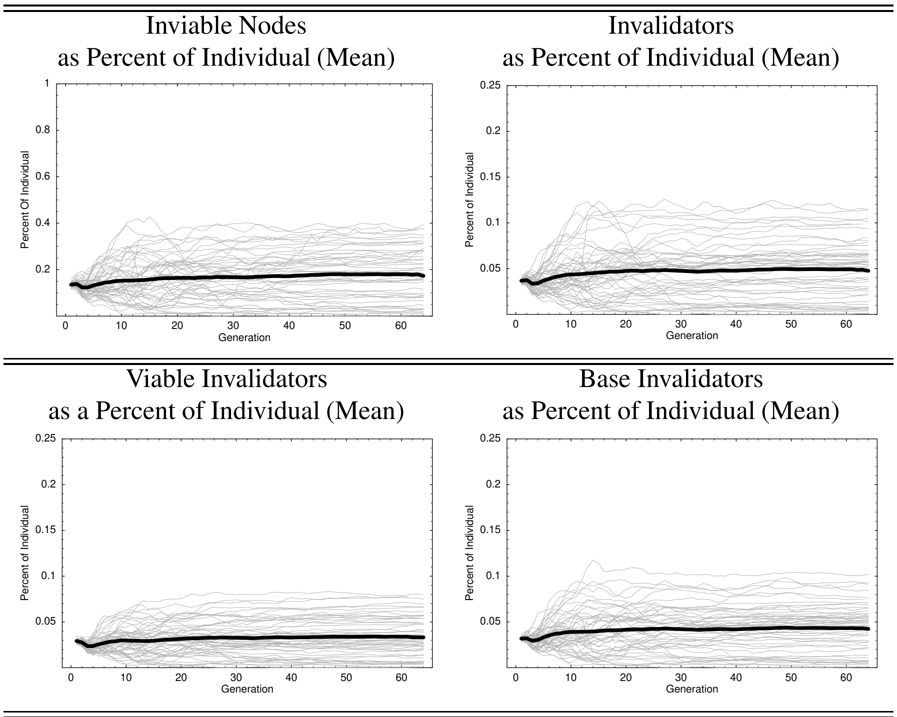



Figure C.6: Inviable Node Data for Symbolic Regression Domain with Inviable Code and Semantically-Identical Crossovers Restricted

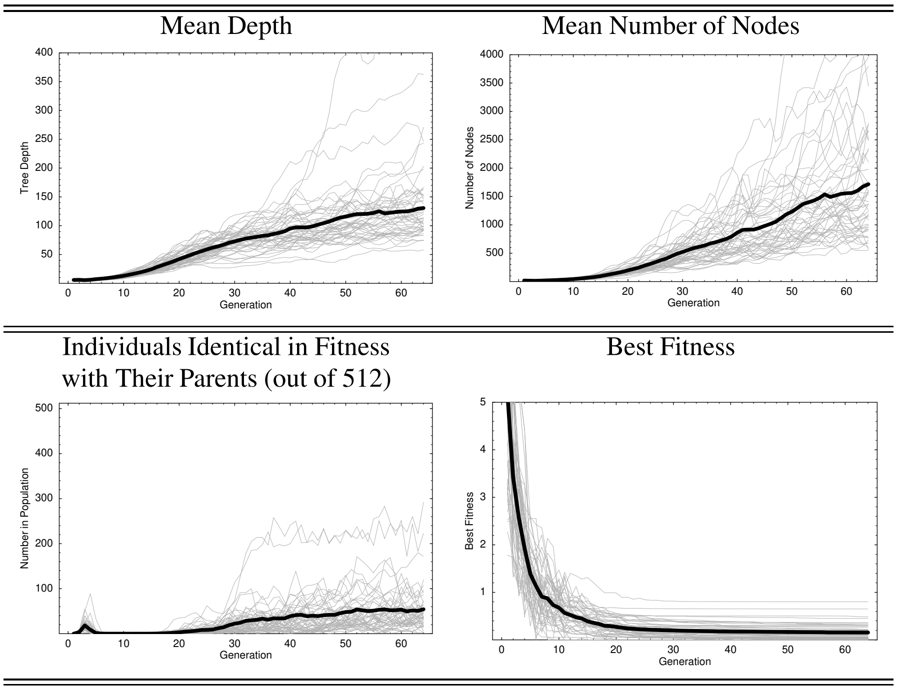



Figure C.7: General Data for Symbolic Regression Domain with Inviable Code, Semantically Identical Crossovers, and Fitness-identical Parents Restricted

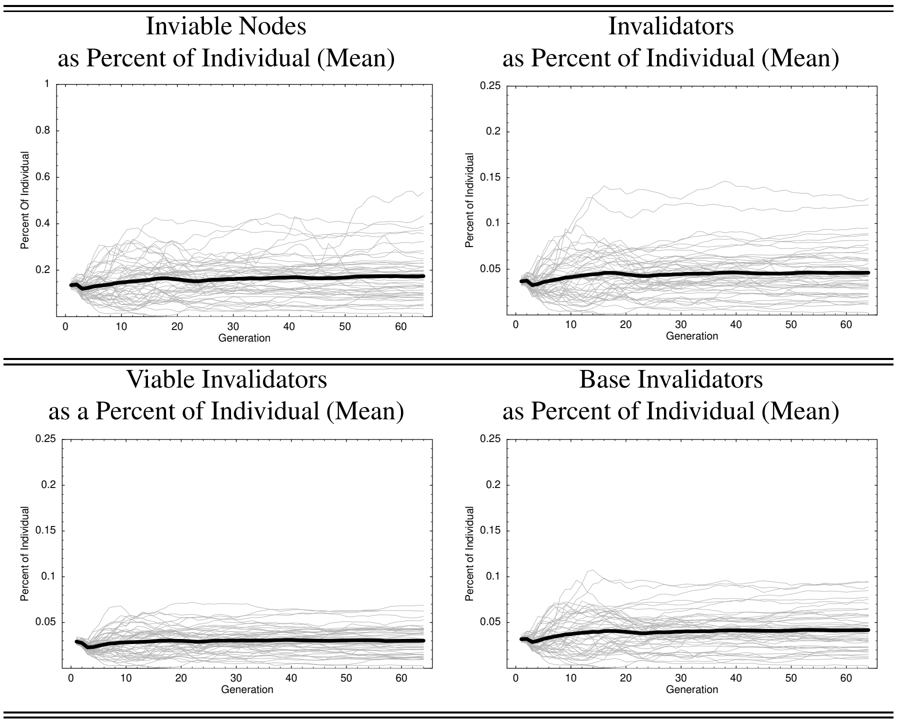



Figure C.8: Inviable Node Data for Symbolic Regression Domain with Inviable Code, Semantically Identical Crossovers, and Fitness-identical Parents Restricted

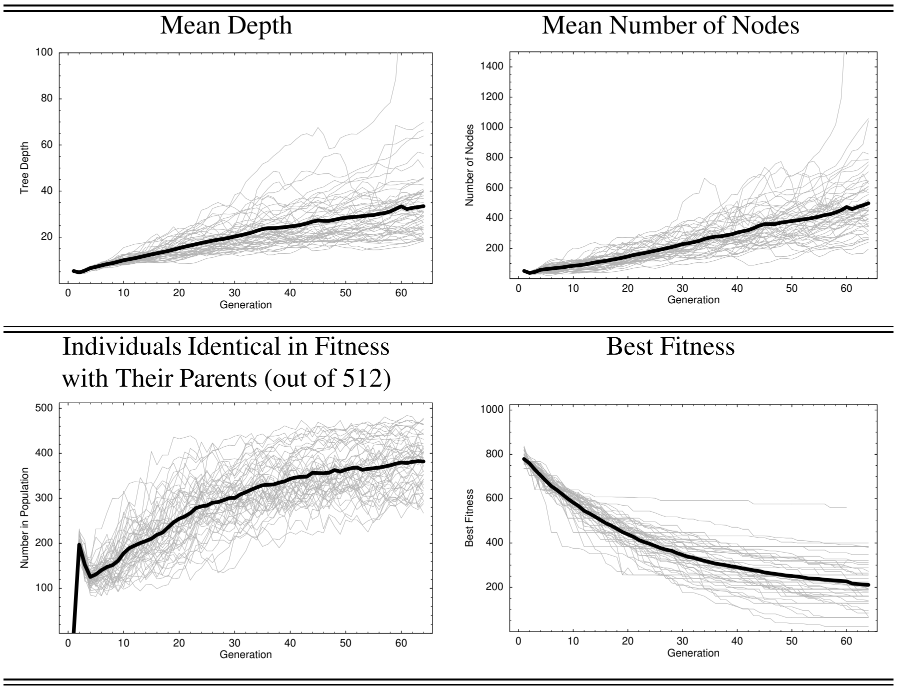



Figure C.9: General Data for 11-Bit Multiplexer Domain

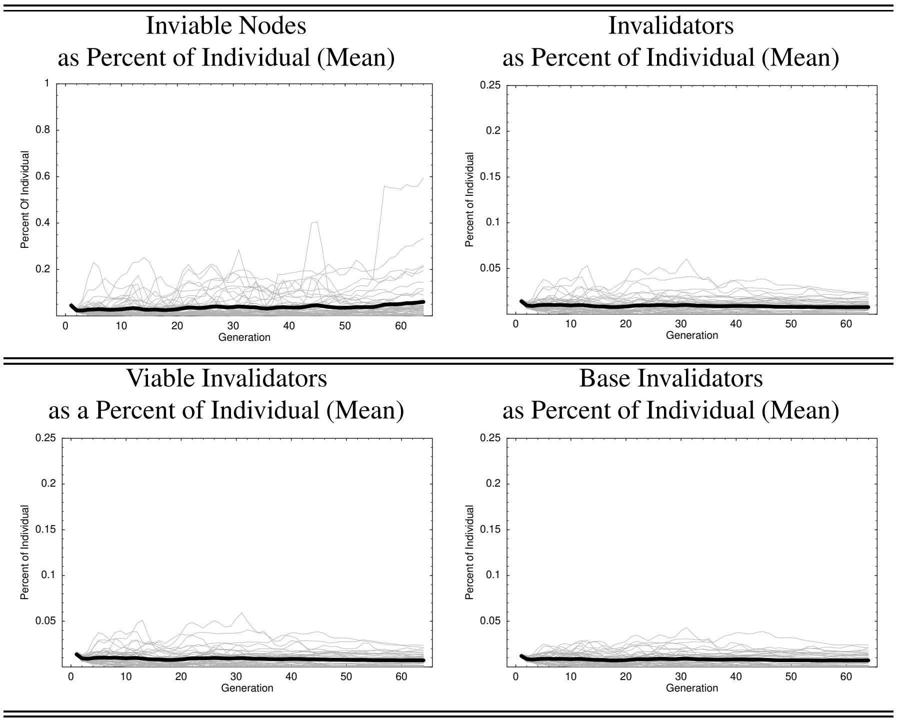



Figure C.10: Inviable Node Data for 11-Bit Multiplexer Domain

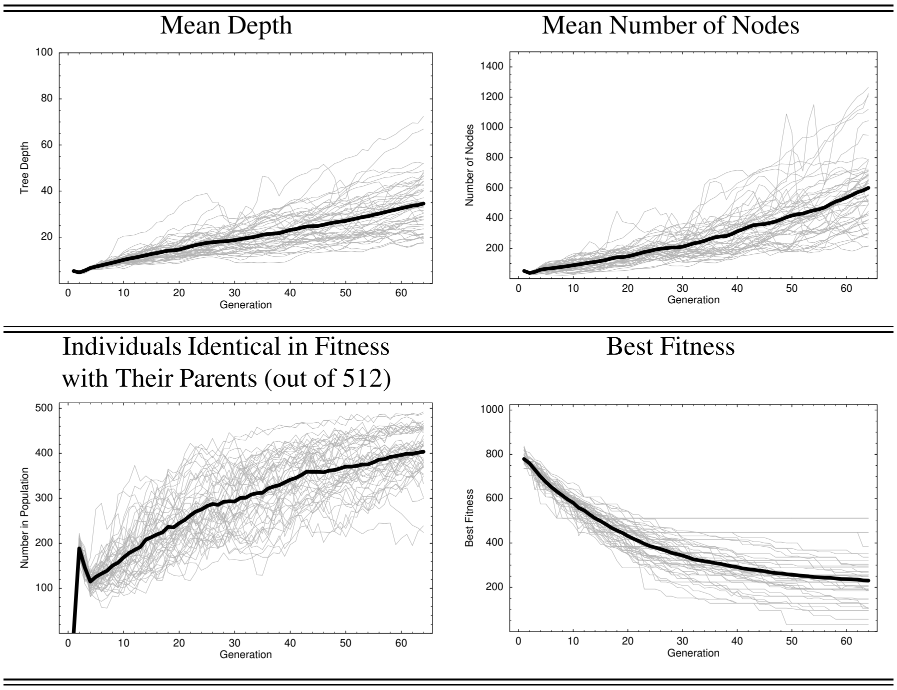



Figure C.11: General Data for 11-Bit Multiplexer Domain with Inviable Code Restricted

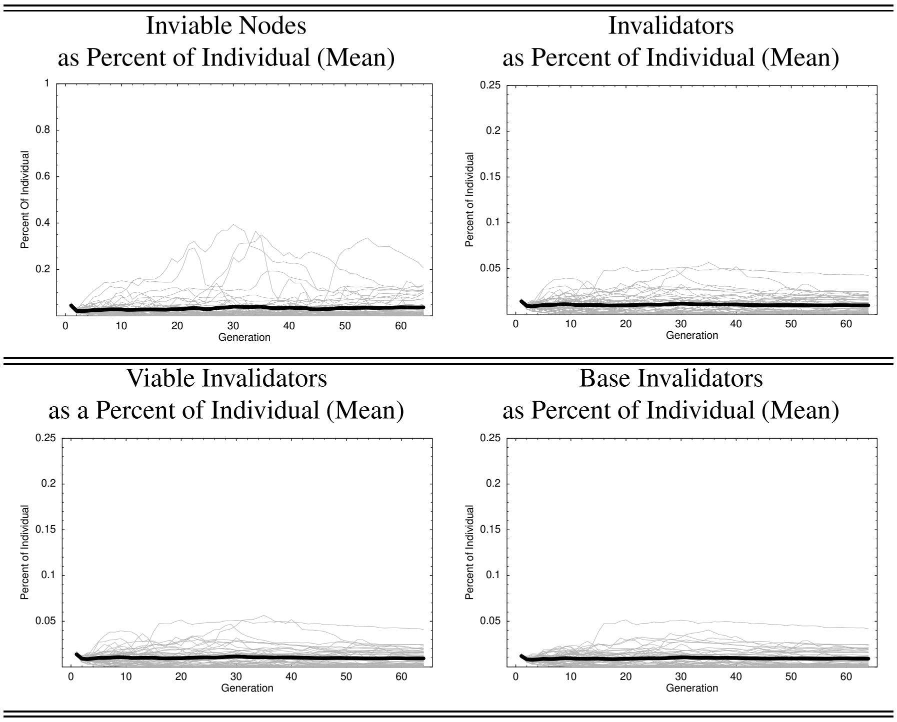



Figure C.12: Inviable Node Data for 11-Bit Multiplexer Domain with Inviable Code Restricted

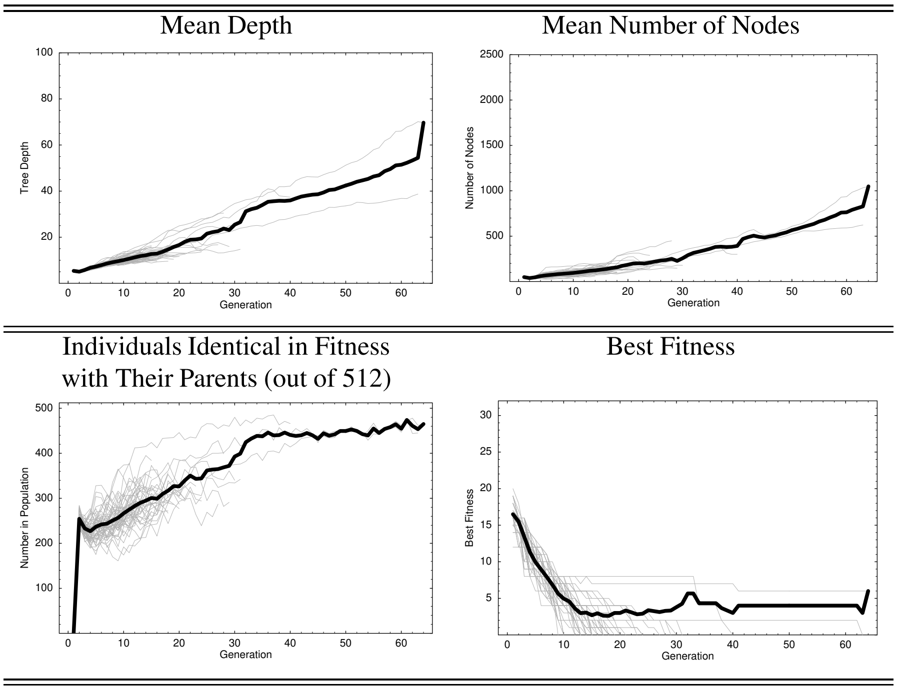



Figure C.13: General Data for 6-Bit Multiplexer Domain

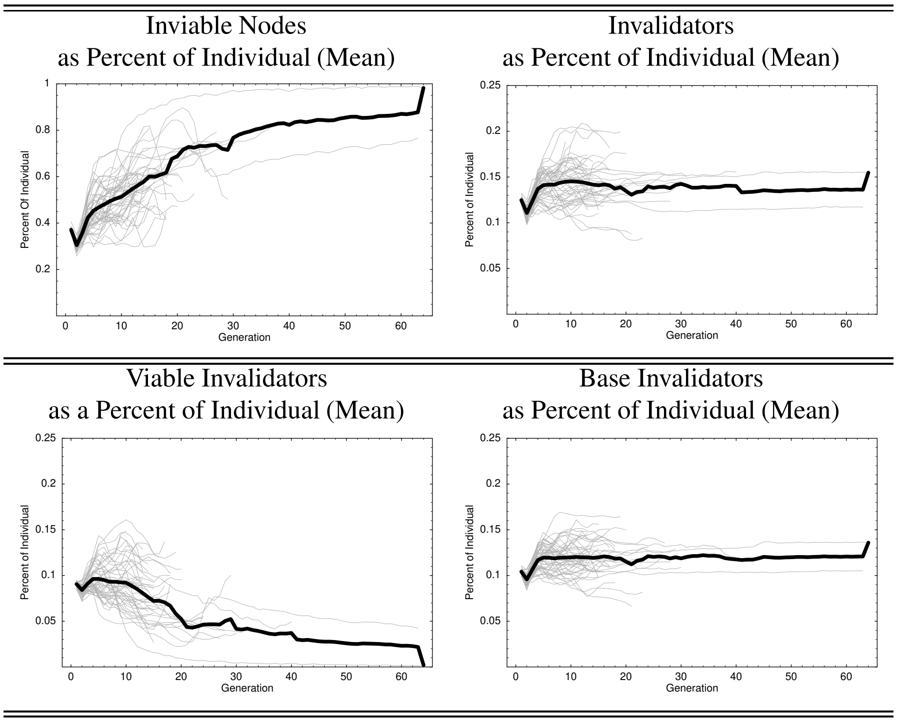



Figure C.14: Inviable Node Data for 6-Bit Multiplexer Domain

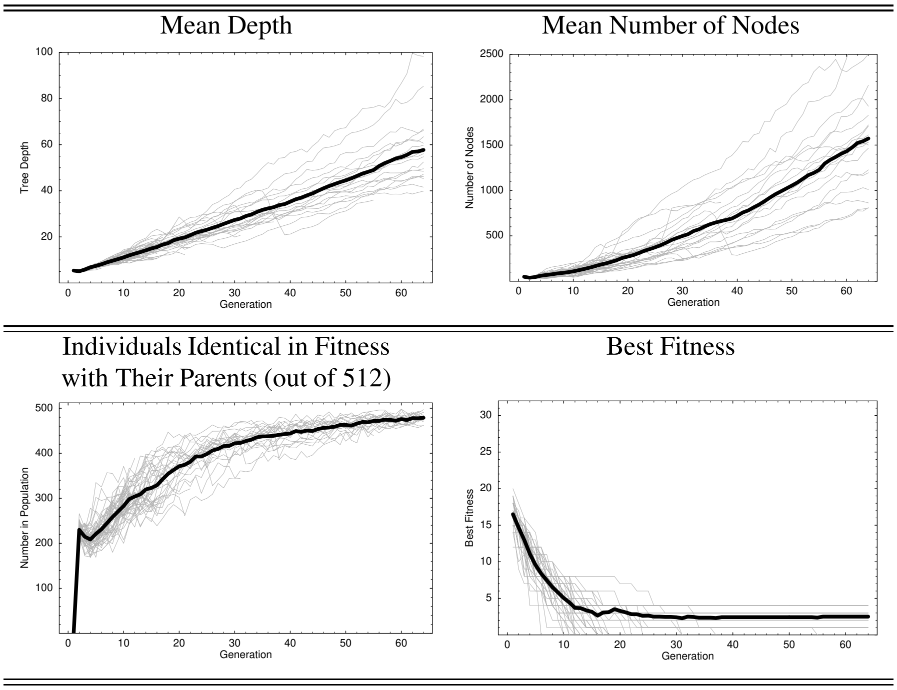



Figure C.15: General Data for 6-Bit Multiplexer Domain with Inviable Code Restricted

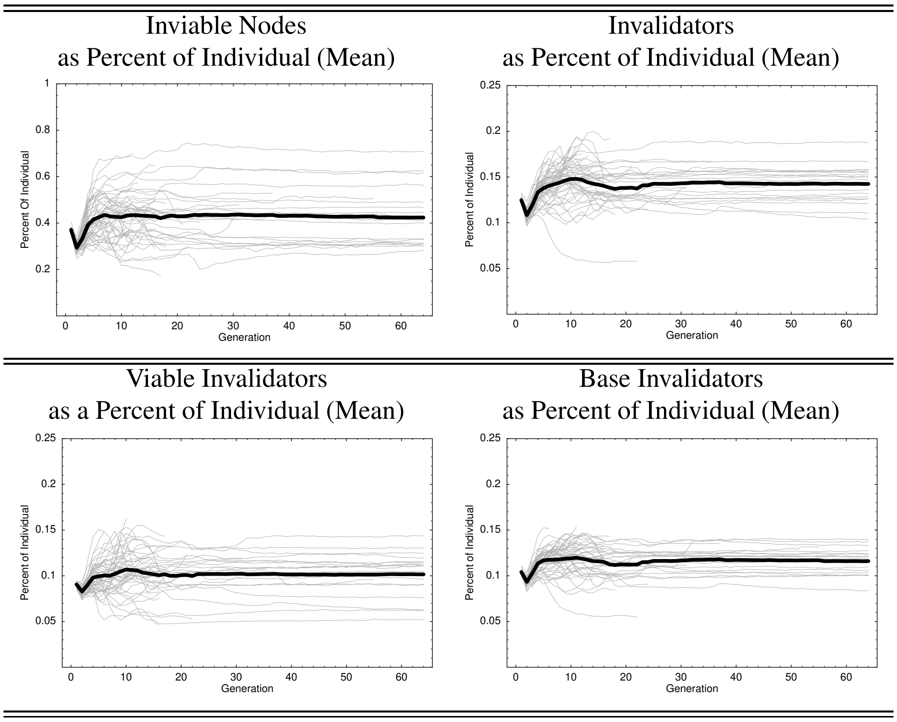



Figure C.16: Inviable Node Data for 6-Bit Multiplexer Domain with Inviable Code Restricted

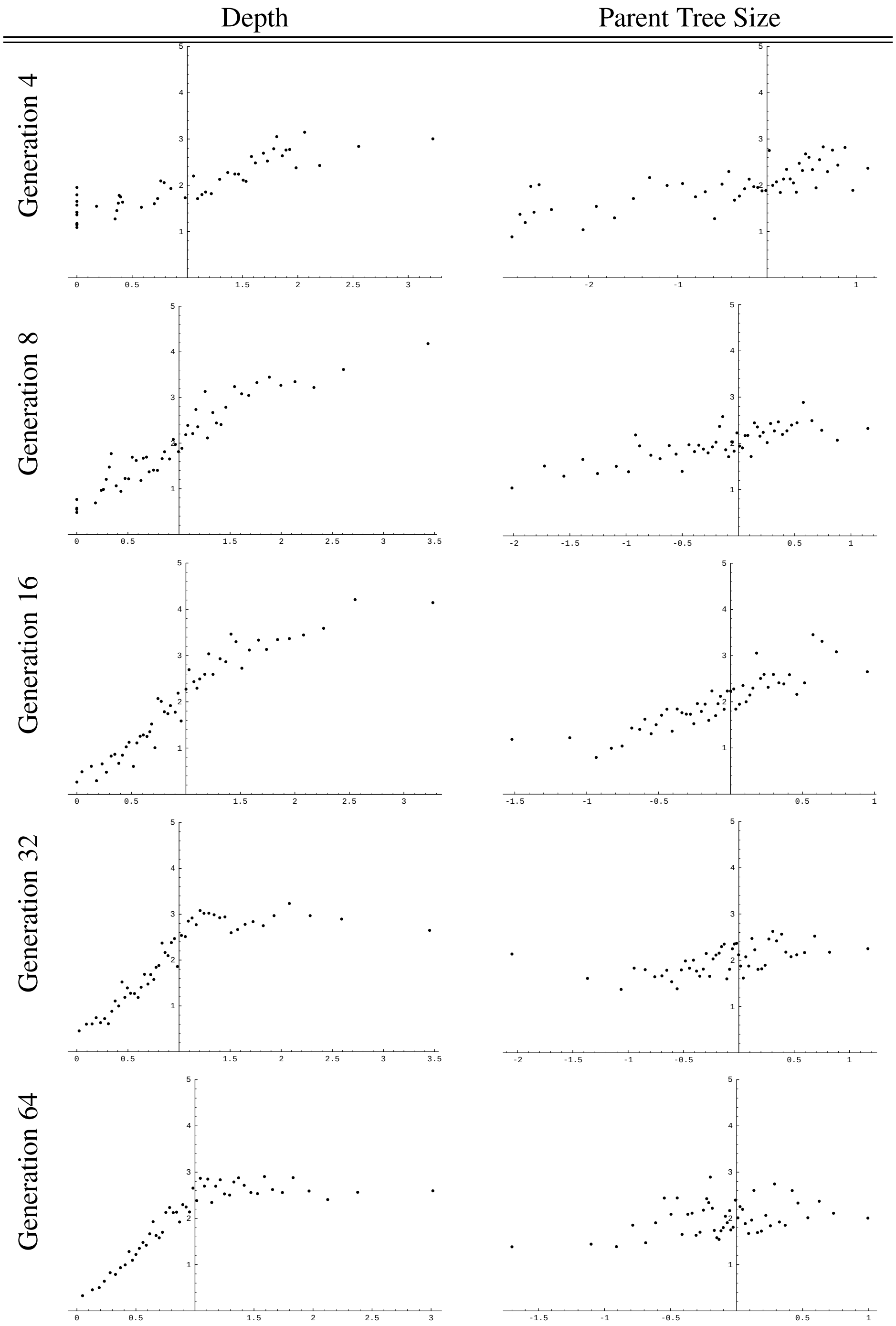



Figure C.17: Relationship of Depth and Parent Tree Size to Child Survivability, Symbolic Regression Domain

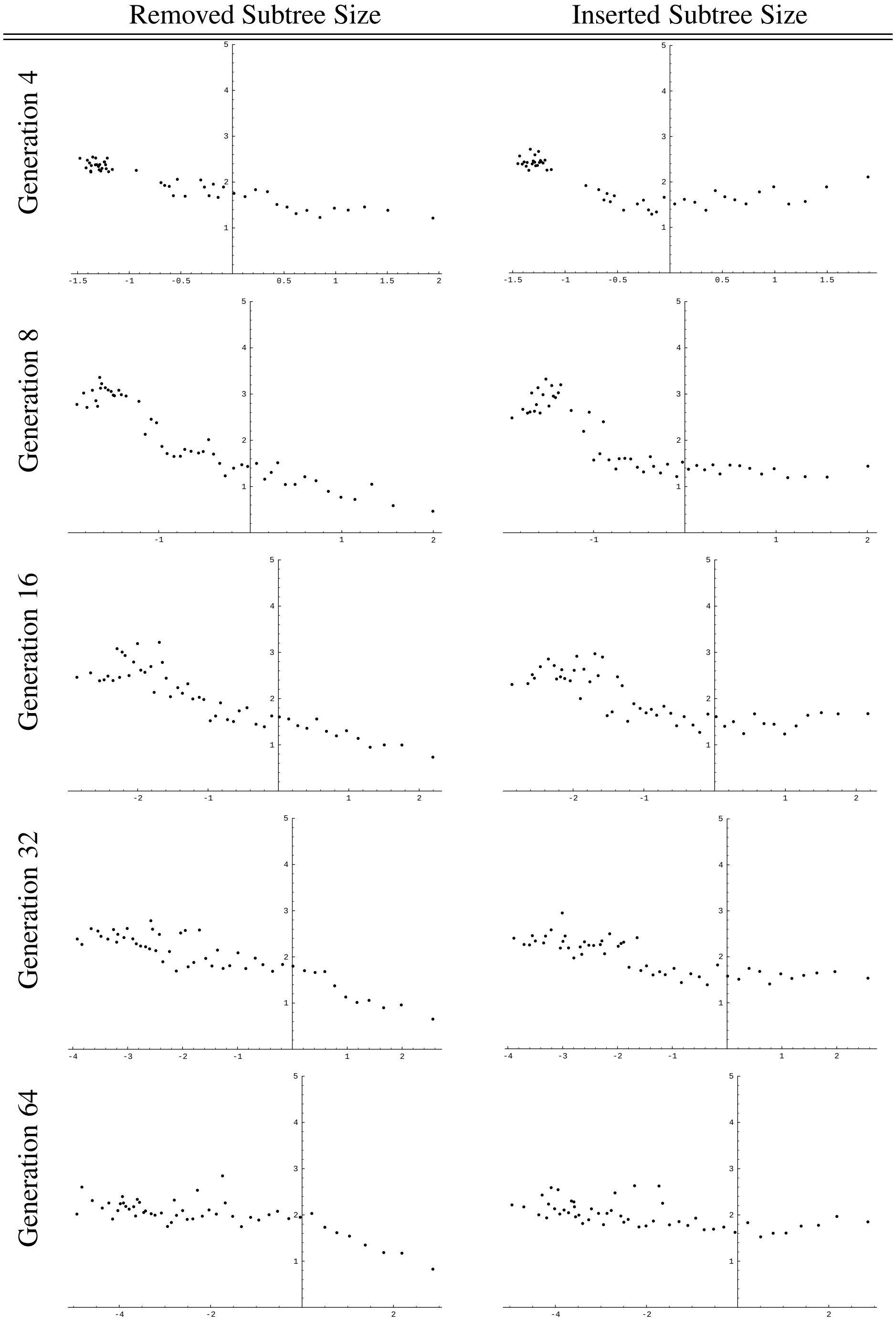



Figure C.18: Relationship of Removed and Inserted Subtree Size to Child Survivability, Symbolic Regression Domain

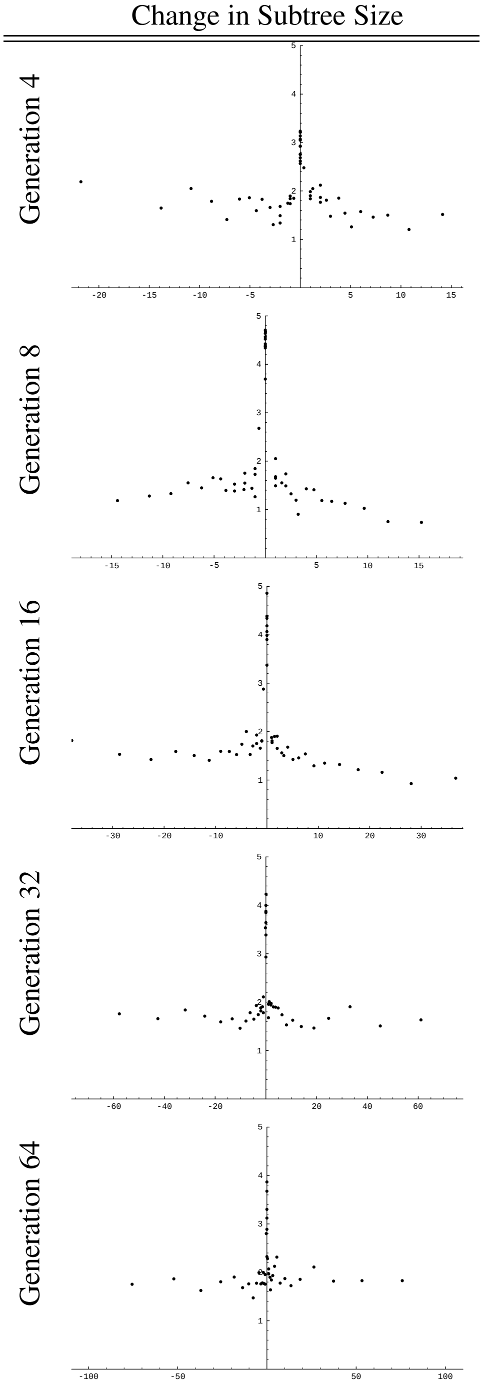



Figure C.19: Relationship of Change in Size to Child Survivability, Symbolic Regression Domain

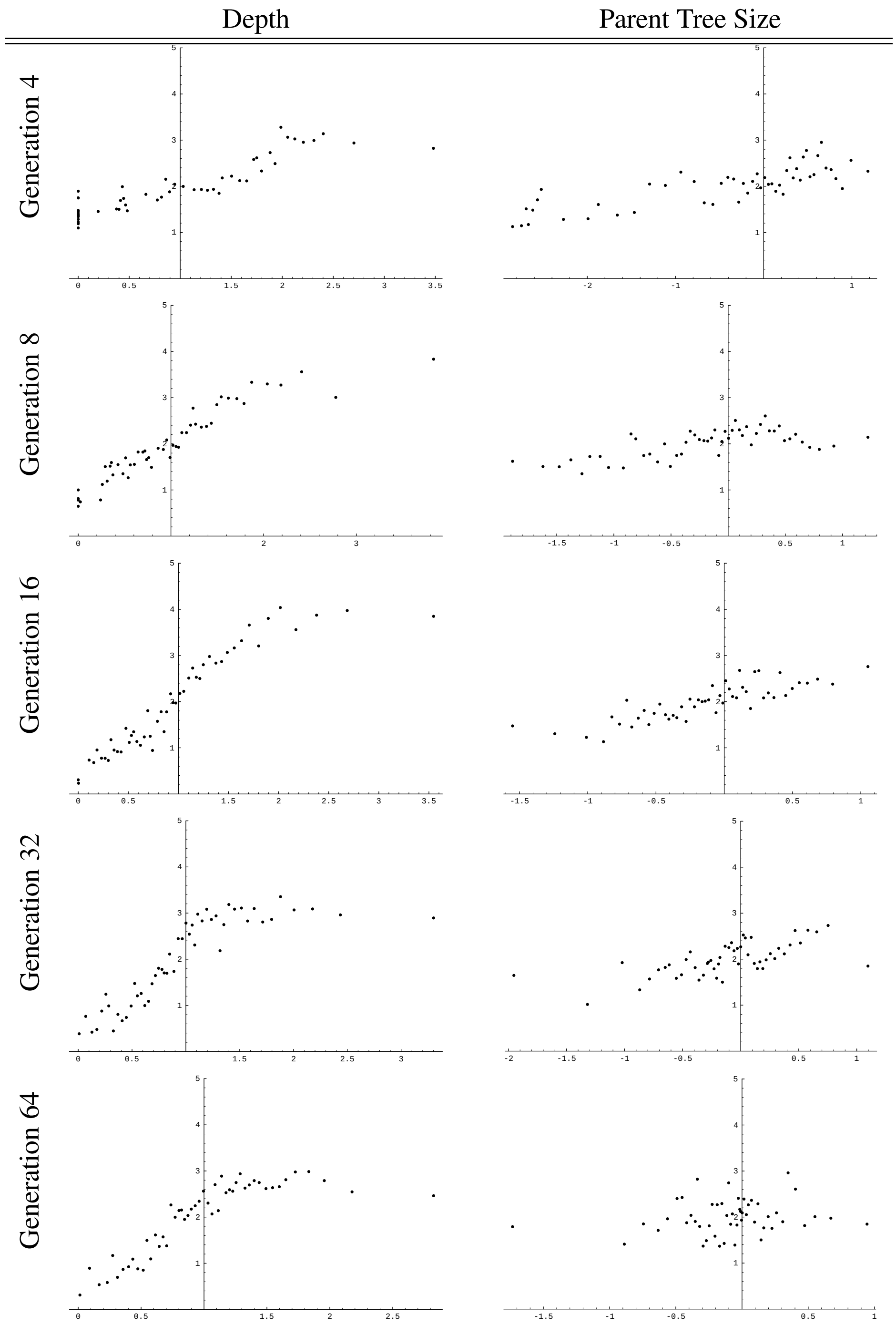



Figure C.20: Relationship of Depth and Parent Tree Size to Child Survivability, Symbolic Regression Domain With Inviable Code Restricted

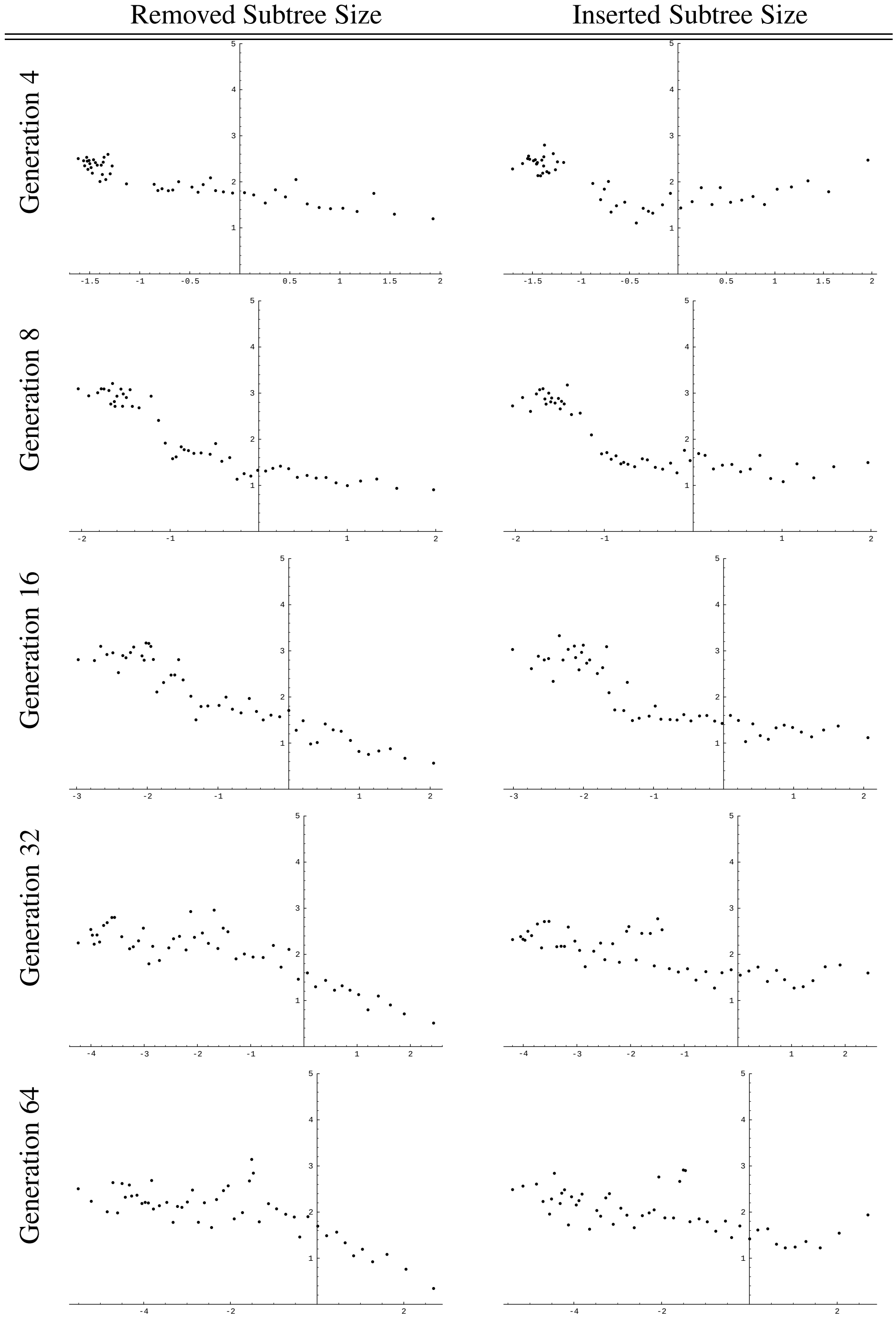



Figure C.21: Relationship of Removed and Inserted Subtree Size to Child Survivability, Symbolic Regression Domain With Inviable Code Restricted

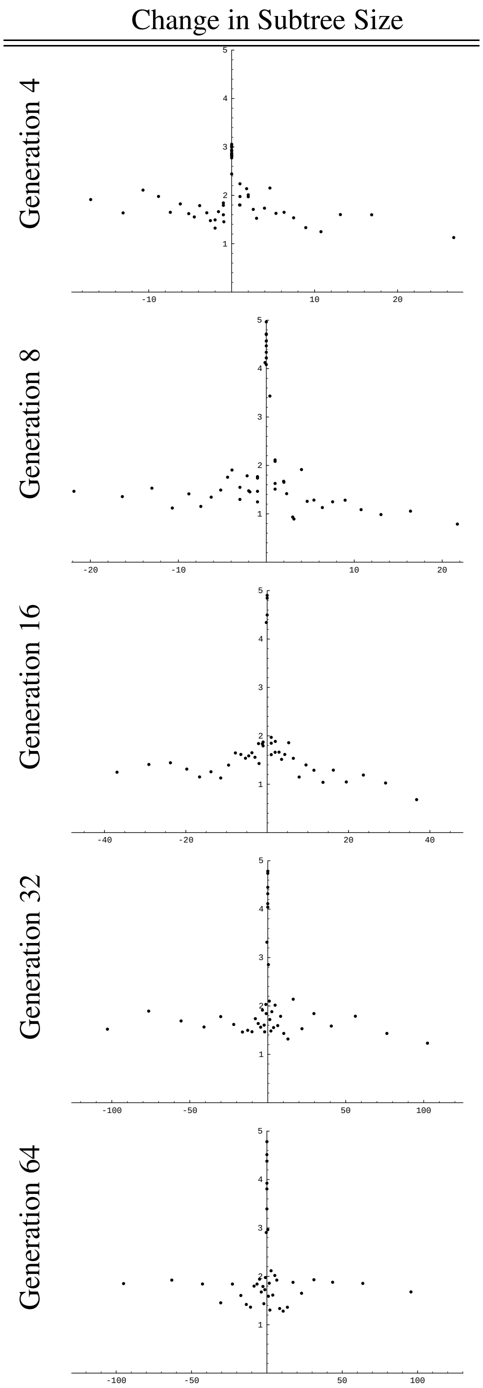



Figure C.22: Relationship of Change in Size to Child Survivability, Symbolic Regression Domain With Inviable Code Restricted

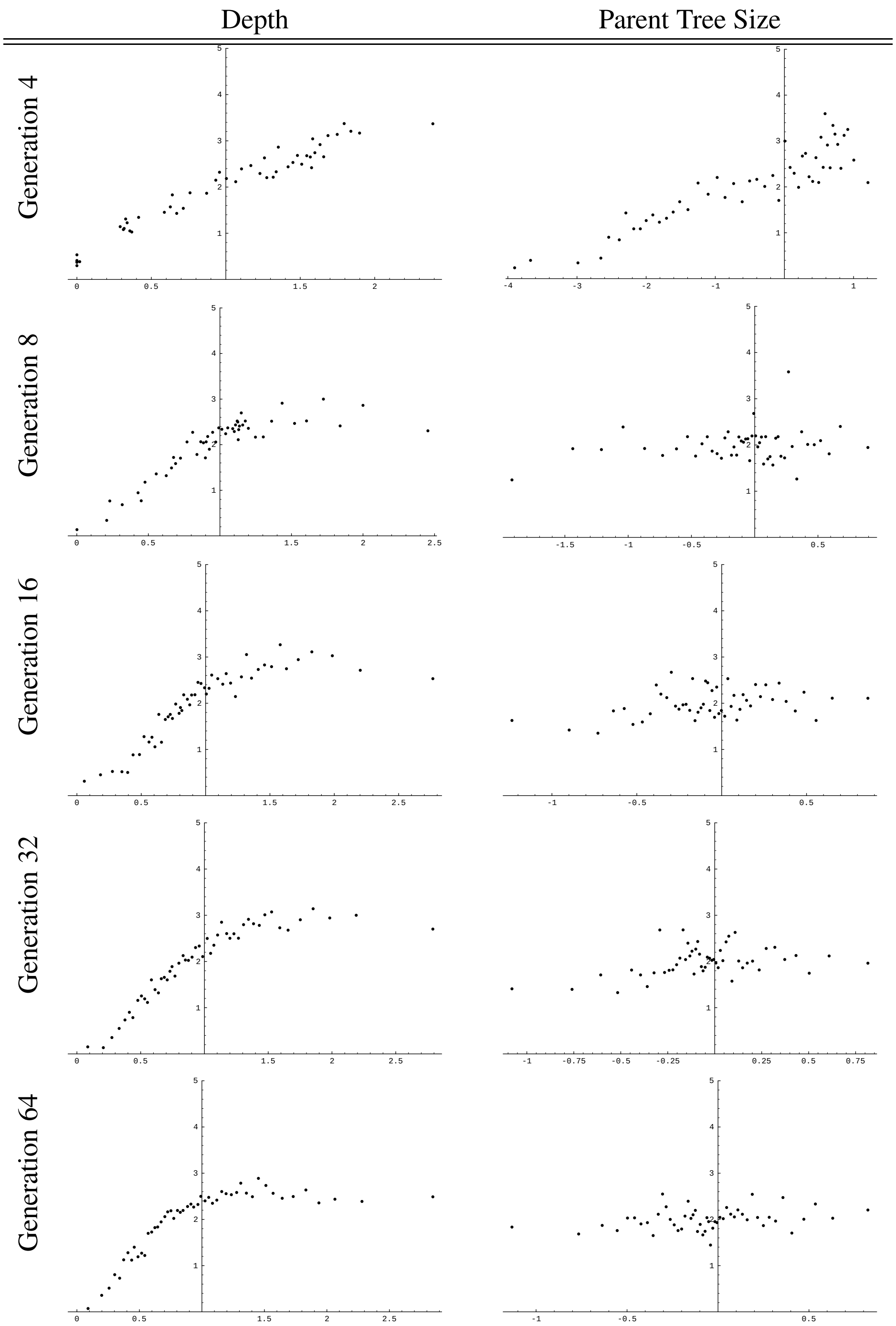



Figure C.23: Relationship of Depth and Parent Tree Size to Child Survivability, 11-Bit Multiplexer Domain

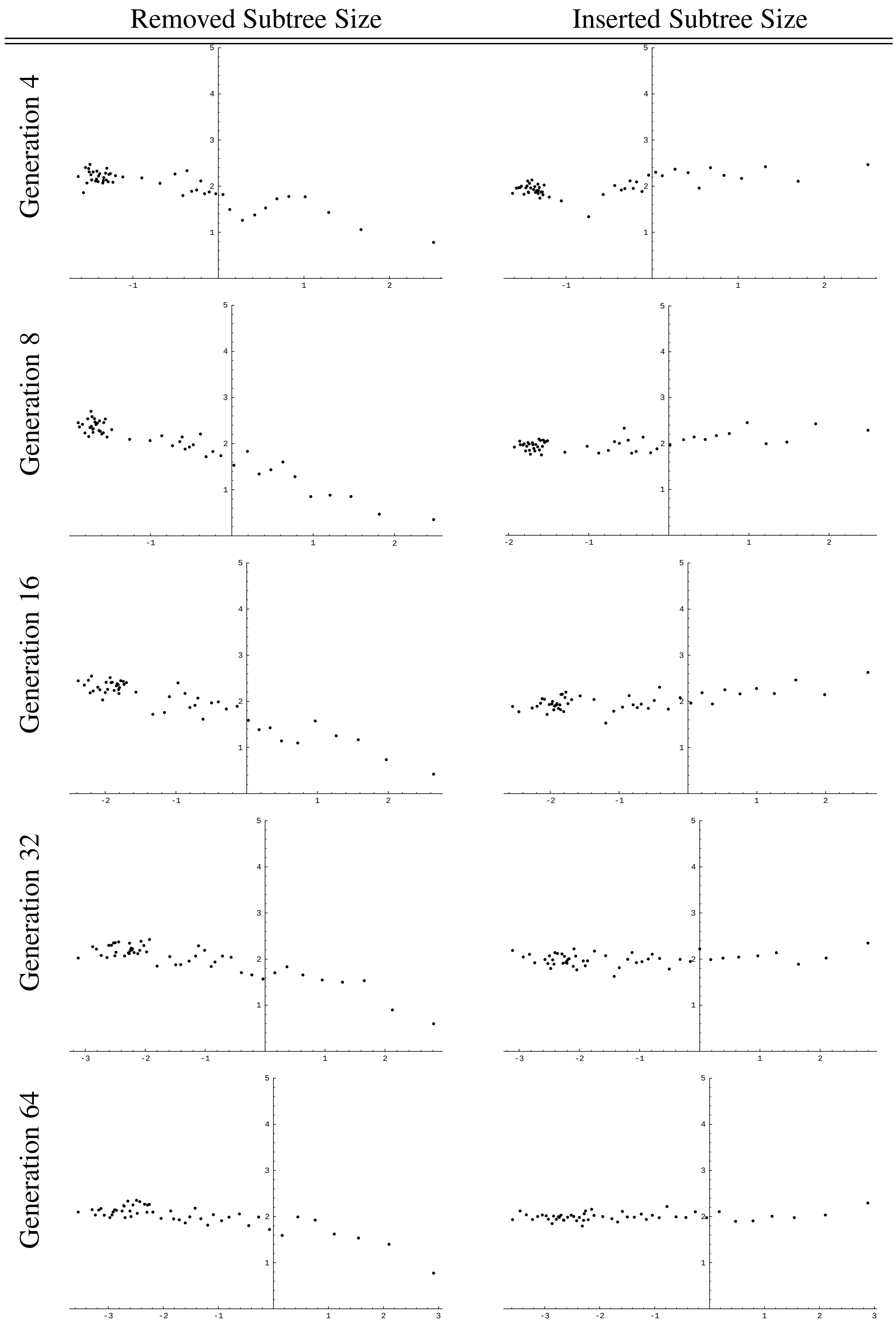



Figure C.24: Relationship of Removed and Inserted Subtree Size to Child Survivability, 11-Bit Multiplexer Domain

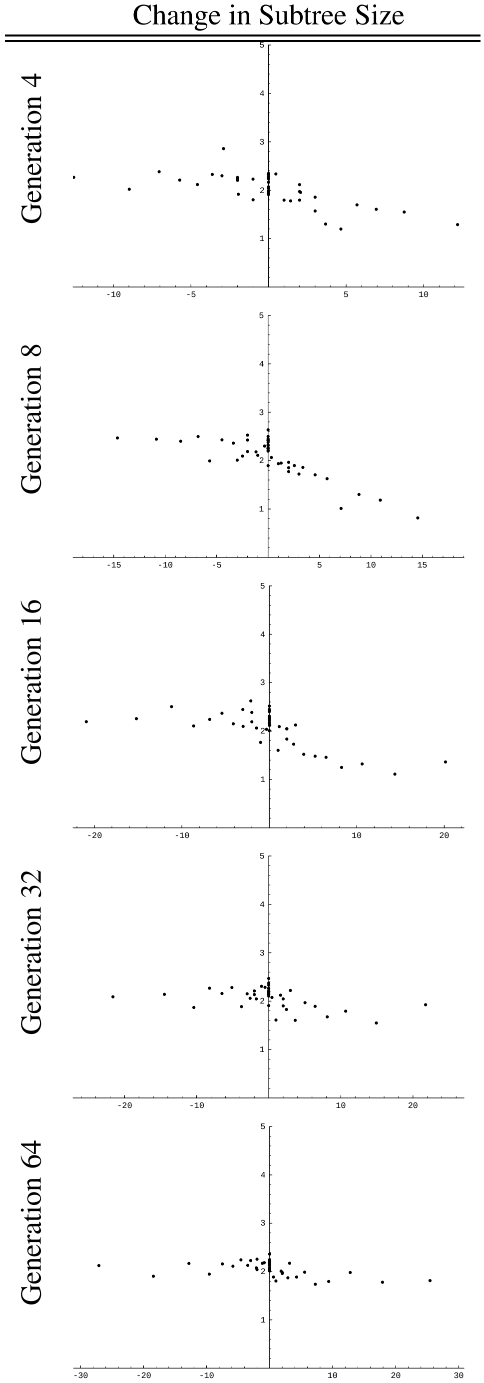



Figure C.25: Relationship of Change in Size to Child Survivability, 11-Bit Multiplexer Domain

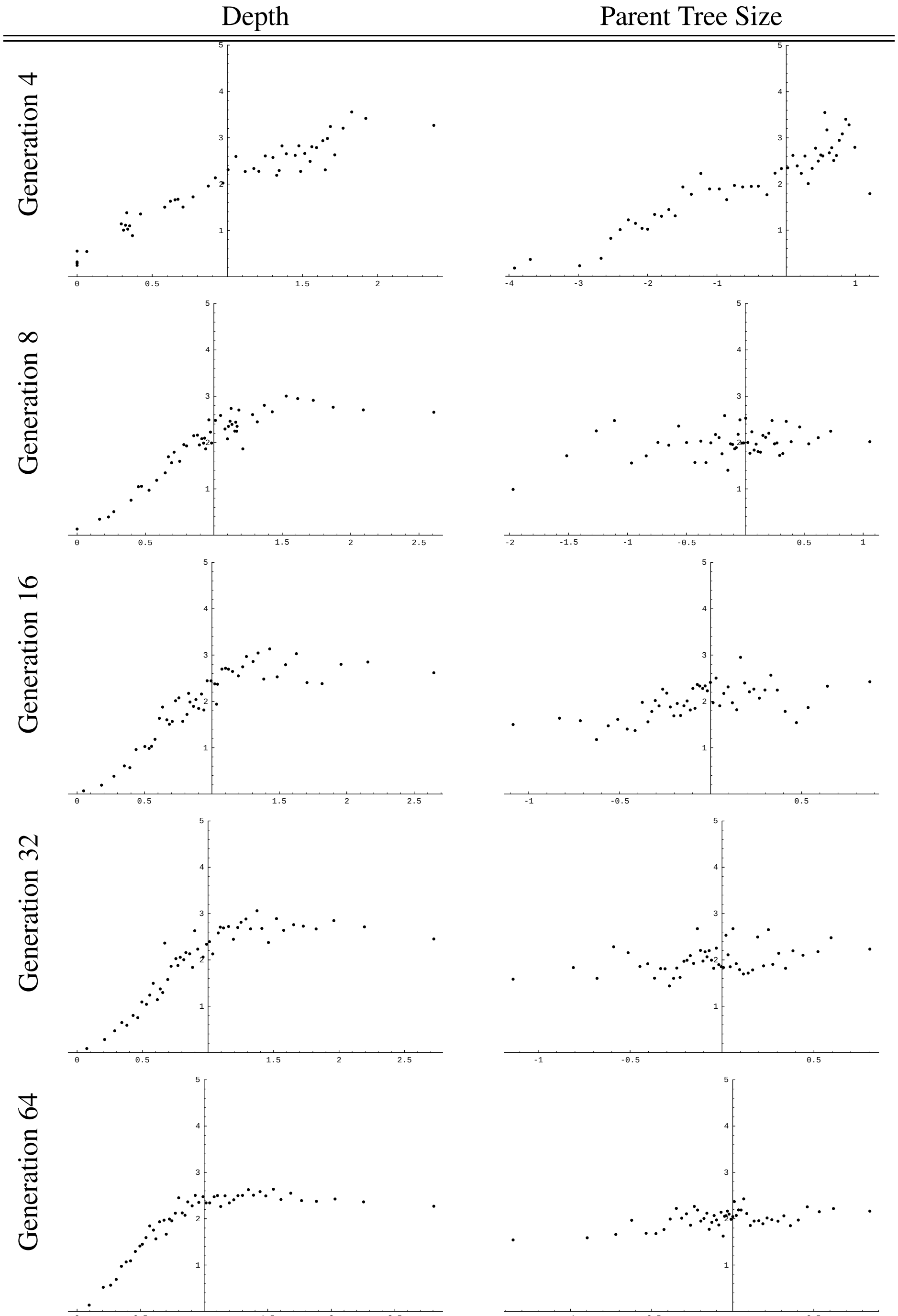



Figure C.26: Relationship of Depth and Parent Tree Size to Child Survivability, 11-Bit Multiplexer Domain With Inviable Code Restricted

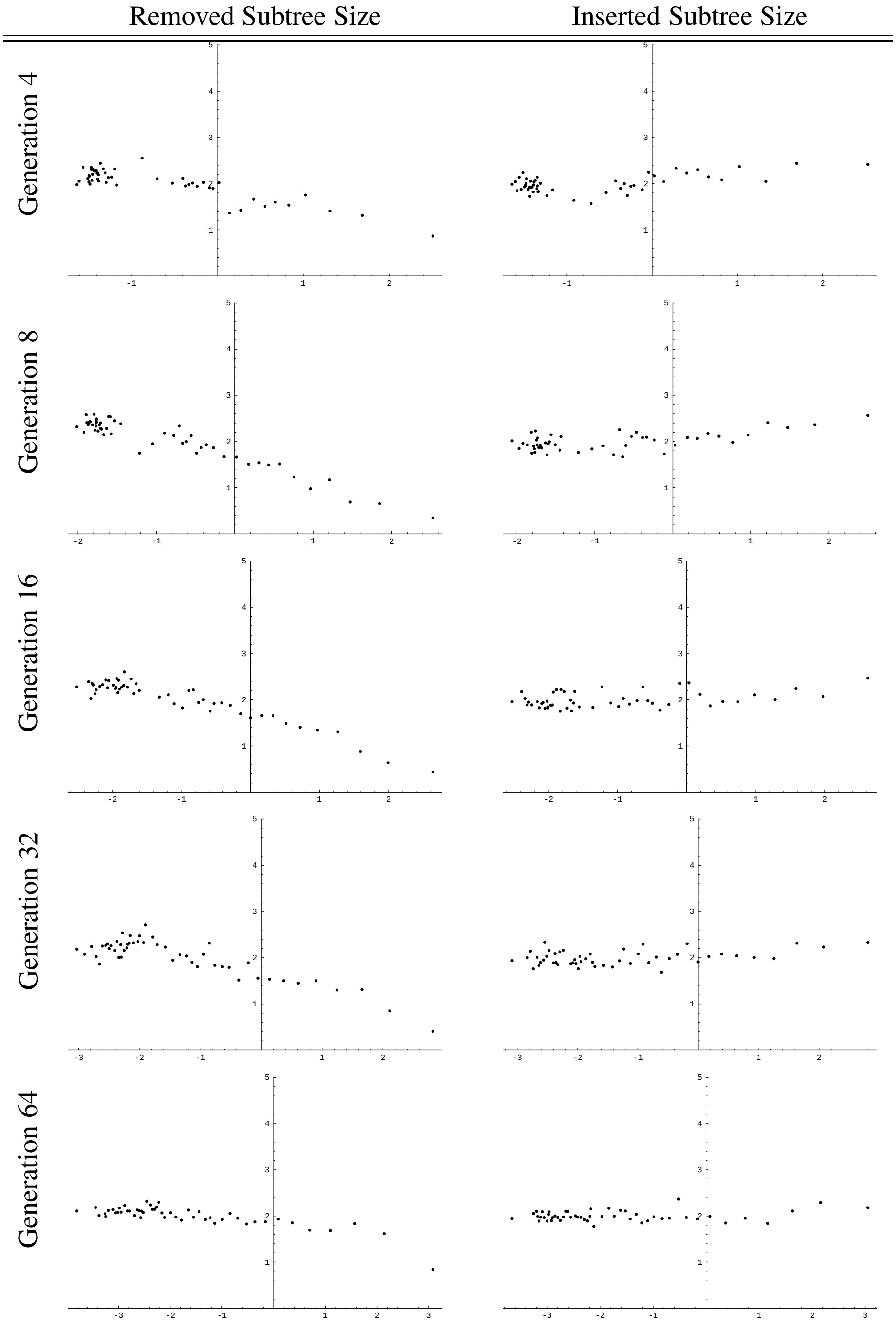



Figure C.27: Relationship of Removed and Inserted Subtree Size to Child Survivability, 11-Bit Multiplexer Domain With Inviable Code Restricted

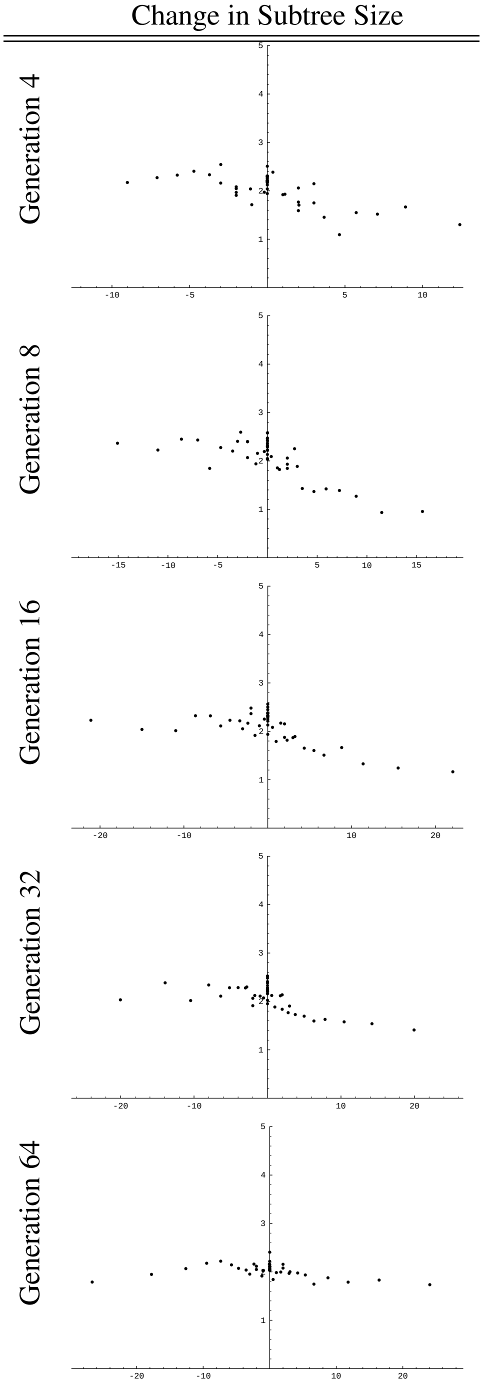



Figure C.28: Relationship of Change in Size to Child Survivability, 11-Bit Multiplexer Domain With Inviable Code Restricted

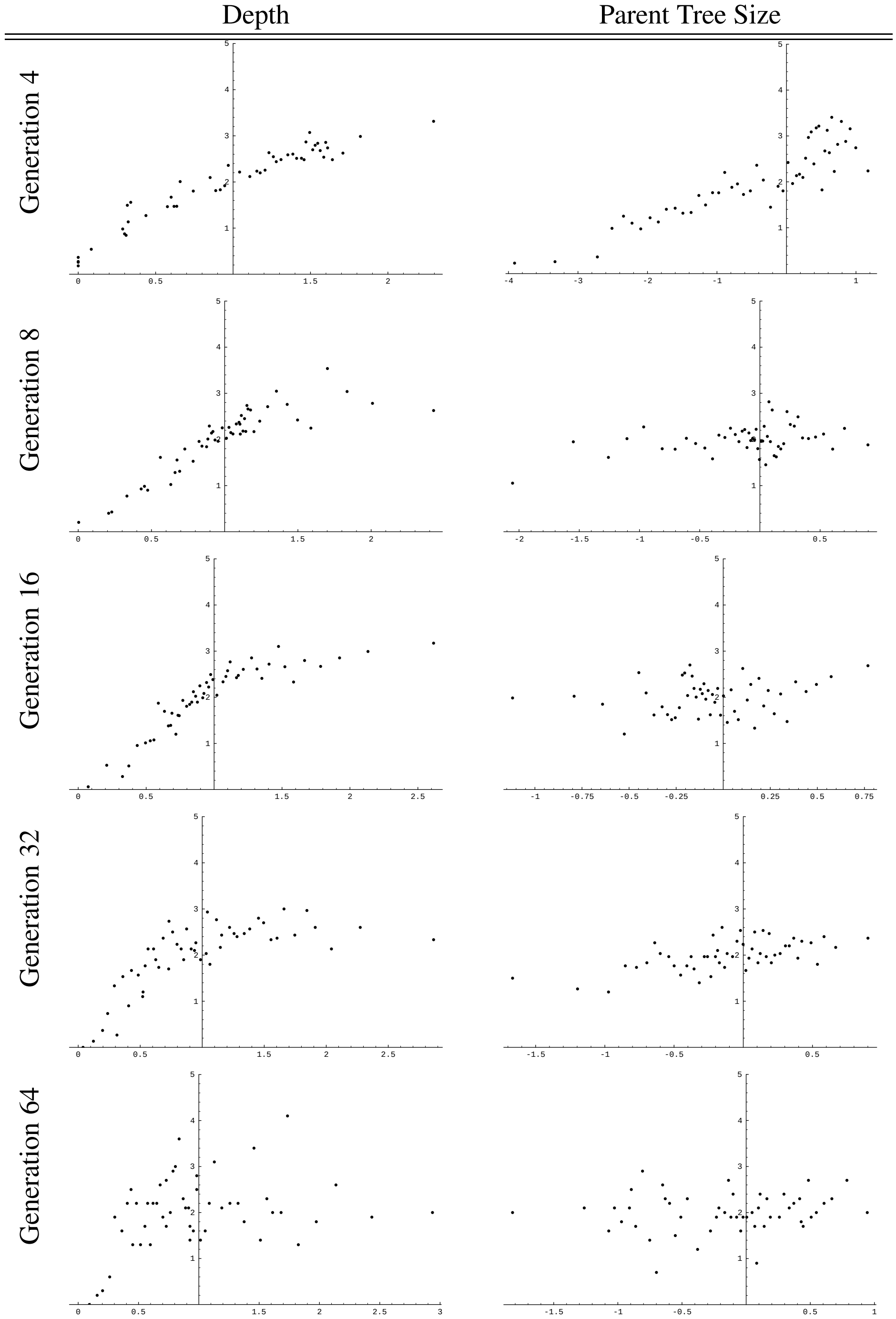



Figure C.29: Relationship of Depth and Parent Tree Size to Child Survivability, 6-Bit Multiplexer Domain

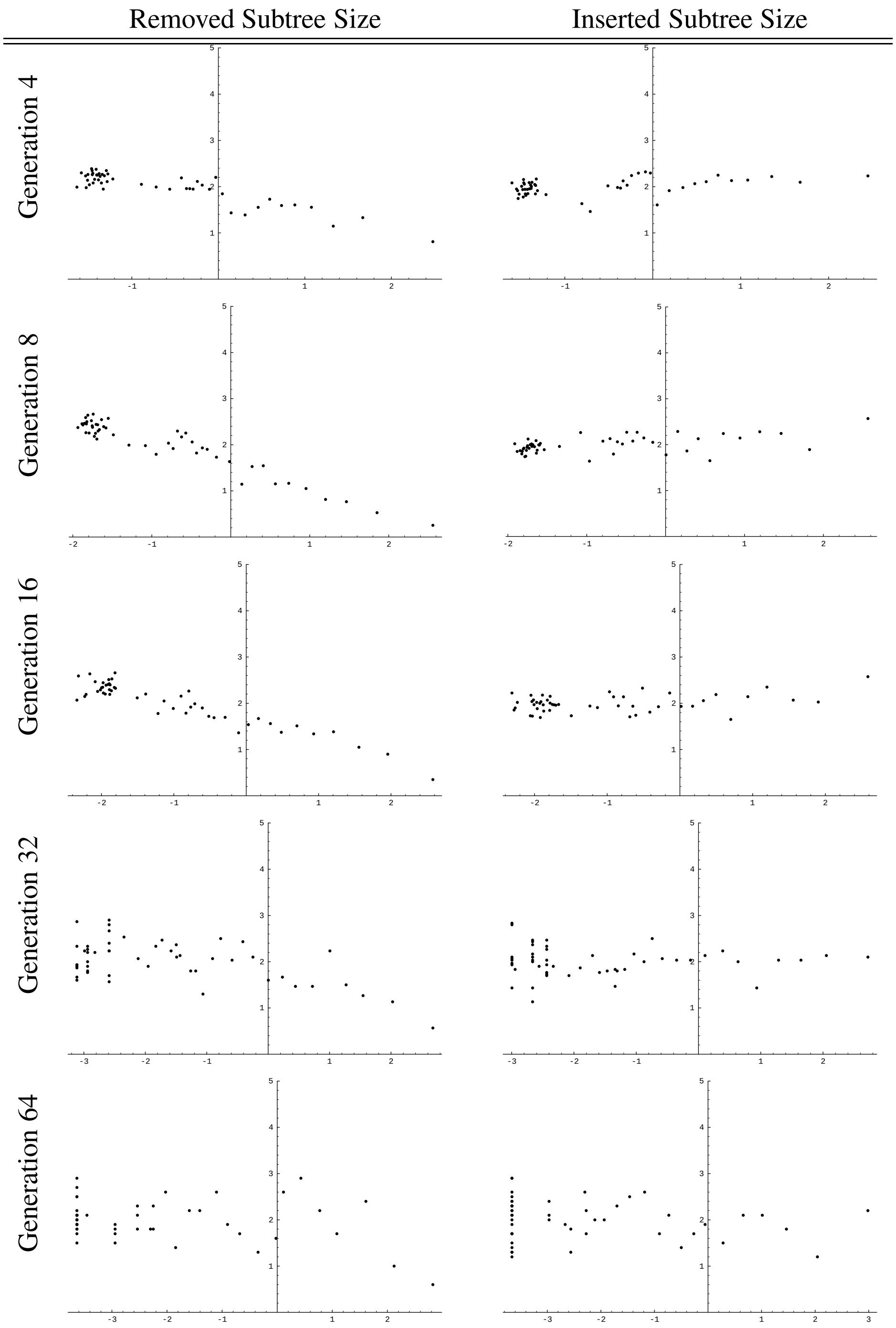



Figure C.30: Relationship of Removed and Inserted Subtree Size to Child Survivability, 6-Bit Multiplexer Domain

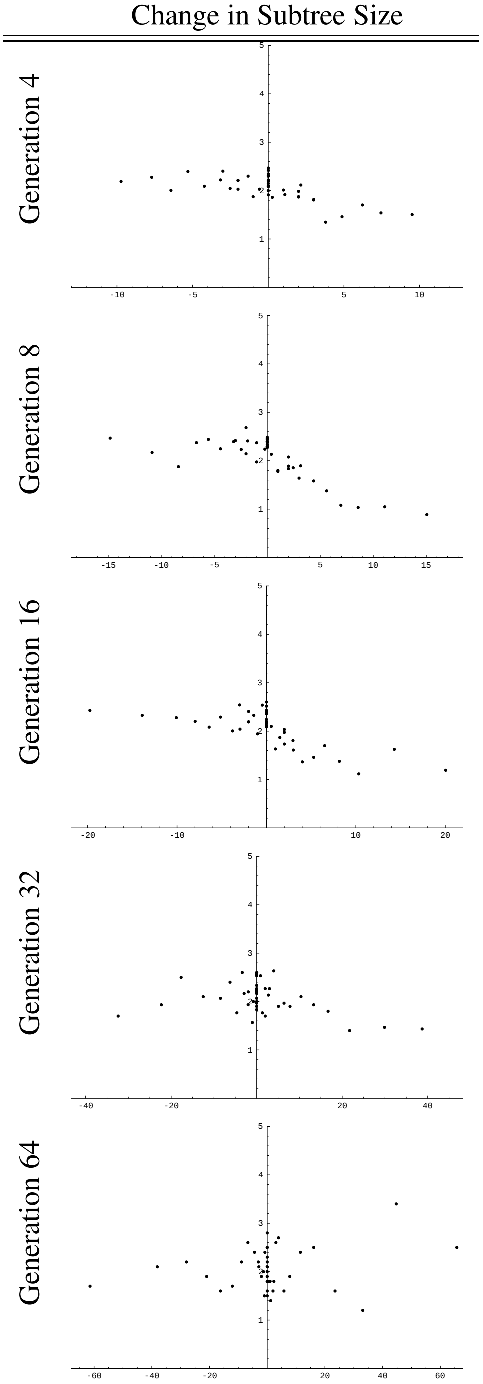



Figure C.31: Relationship of Change in Size to Child Survivability, 6-Bit Multiplexer Domain

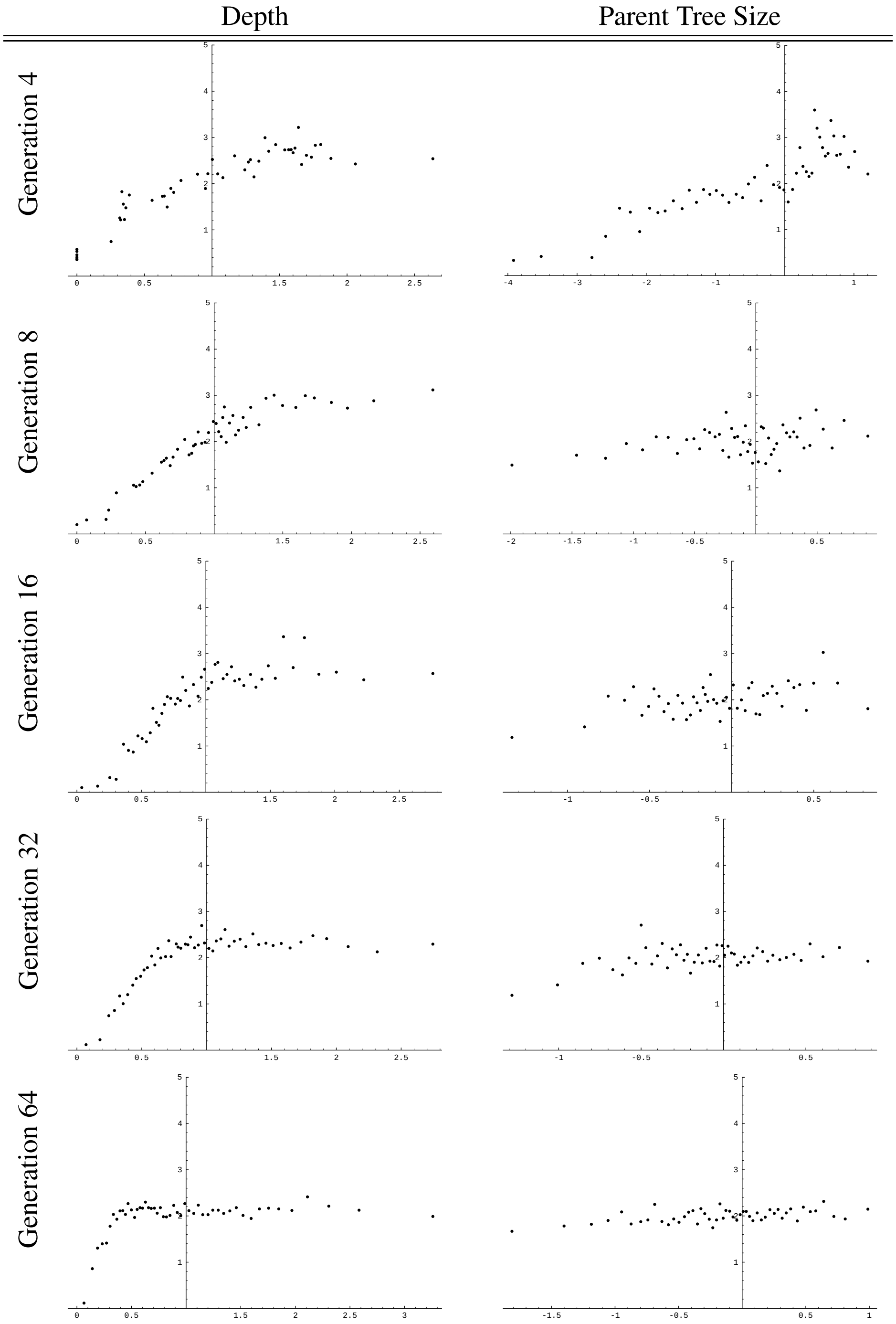



Figure C.32: Relationship of Depth and Parent Tree Size to Child Survivability, 6-Bit Multiplexer Domain With Inviable Code Restricted

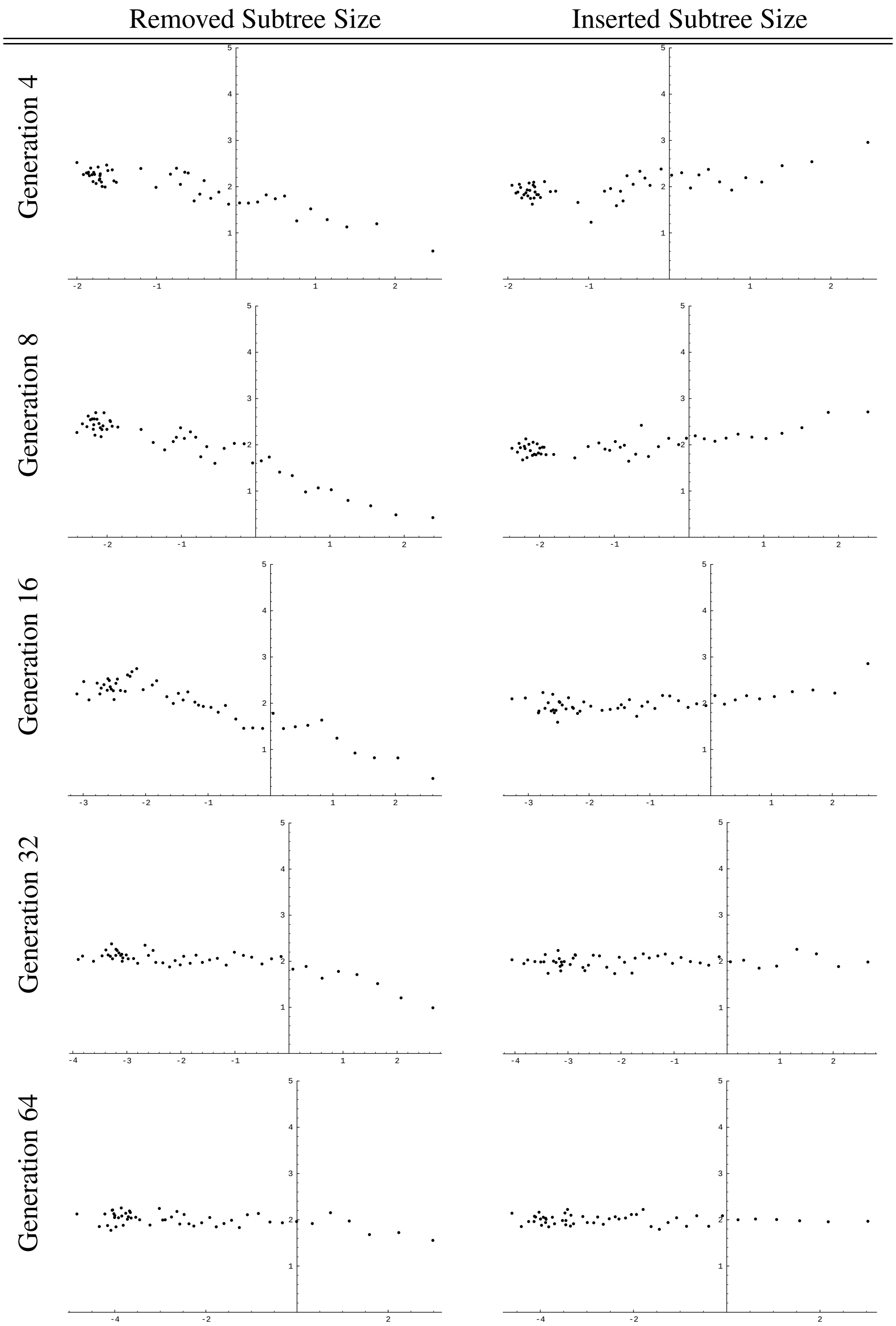



Figure C.33: Relationship of Removed and Inserted Subtree Size to Child Survivability, 6-Bit Multiplexer Domain With Inviable Code Restricted

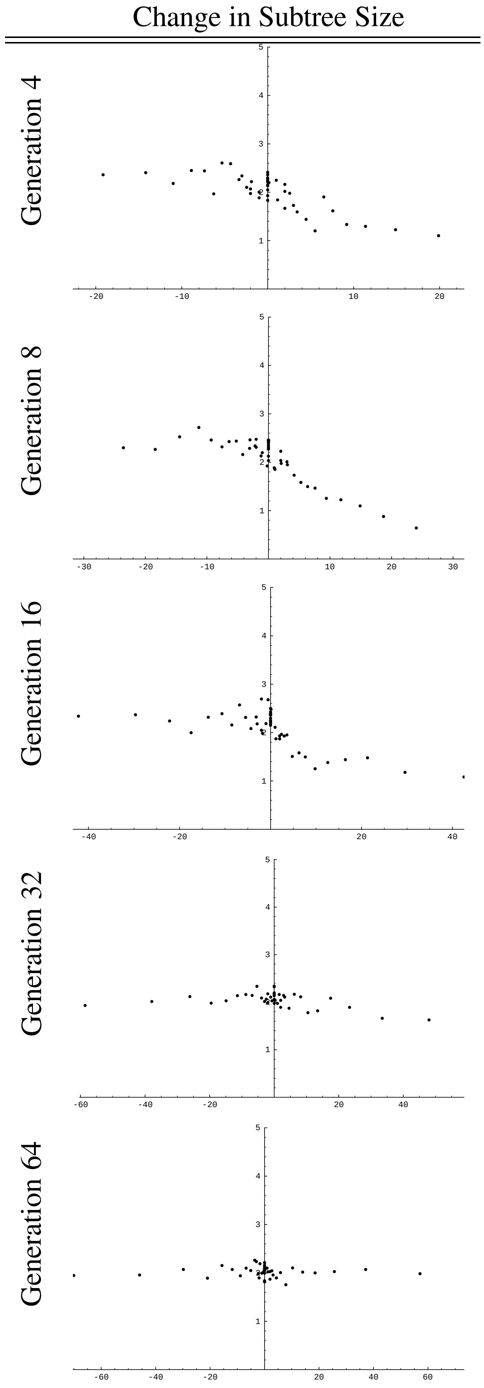



Figure C.34: Relationship of Change in Size to Child Survivability, 6-Bit Multiplexer Domain With Inviable Code Restricted

**Multiple Regression of Child Survivability**
**by Depth, Parent Size, Removed Size, Inserted Size**

| Gen | Depth | Parent | Removed | Inserted | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|---|---|---|
| 4 | 0.1556 | 0.1438 | -0.1481 | -0.0671 | 0.5673 | 2.5249 | 4.7766 |
| 8 | 0.4381 | -0.1031 | -0.3130 | -0.1895 | 0.6953 | 2.5954 | 4.3241 |
| 16 | 0.5521 | *-0.0016 | -0.1379 | -0.0856 | 0.2393 | 2.3894 | 4.1211 |
| 32 | 0.4953 | -0.2495 | -0.1024 | -0.0413 | 0.5091 | 2.2728 | 3.9419 |
| 64 | 0.4940 | -0.2505 | -0.0458 | -0.0102 | 0.4548 | 2.1539 | 3.6412 |

**Multiple Regression of Child Survivability**
**by Depth, Log(Parent Size), Log(Removed Size), Log(Inserted Size)**

| Gen | Depth | Parent | Removed | Inserted | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|---|---|---|
| 4 | *-0.0267 | 0.2467 | -0.3306 | -0.1730 | 0.5209 | 2.5304 | 4.6923 |
| 8 | 0.3593 | *0.0249 | -0.3715 | -0.3140 | -0.2328 | 2.4805 | 4.1844 |
| 16 | 0.5047 | 0.1144 | -0.1811 | -0.1596 | -0.2547 | 2.3718 | 4.0341 |
| 32 | 0.4603 | -0.2027 | -0.1151 | -0.0894 | -0.1884 | 2.2398 | 3.9076 |
| 64 | 0.4637 | -0.1719 | -0.0497 | -0.0385 | *-0.0240 | 2.1594 | 3.6451 |

**Regression of Removed Size**
**by Depth**

| Gen | Depth | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|
| 4 | -0.4196 | 0.4401 | 0.6013 | 0.3615 |
| 8 | -0.5767 | 0.5275 | 0.4975 | 0.2475 |
| 16 | -0.6683 | 0.6285 | 0.3994 | 0.1596 |
| 32 | -0.5983 | 0.5530 | 0.4602 | 0.2118 |
| 64 | -0.8792 | 0.6350 | 0.3985 | 0.1588 |

**Regression of Parent Size**
**by Depth**

| Gen | Depth | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|
| 4 | 0.5598 | 1.4181 | 1.2474 | 1.5562 |
| 8 | 0.4724 | 1.5763 | 1.2953 | 1.6778 |
| 16 | 0.3715 | 1.6683 | 1.6326 | 2.6655 |
| 32 | 0.4470 | 1.5983 | 2.2798 | 5.1979 |
| 64 | 0.3650 | 1.8792 | 2.9464 | 8.6818 |

**Legend.** Asterisks indicate statistically insignificant results (with an alpha value greater than 0.05). *Prescaled Dev/DF* indicates the deviance of the model, divided by the degrees of freedom, prior to scaling (at which time it approaches 1.0). *Intercept* and *Scale* are constants in the model function.

Table C.1: Exploratory Regression Analysis of the Symbolic Regression Domain

**Regression of Child Survivability by Depth**

| Gen. | Depth | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | 0.2881 | 0.3781 | 2.5087 | 4.8236 |
| 8 | 0.4497 | 0.1807 | 2.4417 | 4.5604 |
| 16 | 0.5882 | 0.0126 | 2.3938 | 4.2020 |
| 32 | 0.3988 | 0.2512 | 2.2904 | 4.0427 |
| 64 | 0.4130 | 0.2455 | 2.1666 | 3.6854 |

**Regression of Child Survivability by Parent Size**

| Gen. | Parent | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | 0.1964 | 0.4864 | 2.5256 | 4.8236 |
| 8 | 0.1902 | 0.4960 | 2.5049 | 4.5604 |
| 16 | 0.4466 | 0.2222 | 2.4453 | 4.2020 |
| 32 | 0.1297 | 0.5608 | 2.3134 | 4.0427 |
| 64 | 0.1081 | 0.5838 | 2.1907 | 3.6854 |

**Regression of Child Survivability by Removed Size**

| Gen. | Removed | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | -0.1562 | 0.8333 | 2.5491 | 4.8236 |
| 8 | -0.4359 | 1.0207 | 2.6725 | 4.5604 |
| 16 | -0.2166 | *0.8623 | 2.4556 | 4.2020 |
| 32 | -0.1467 | 0.8018 | 2.3111 | 4.0427 |
| 64 | -0.0711 | 0.7480 | 2.1785 | 3.6854 |

**Regression of Child Survivability by Inserted Size**

| Gen. | Inserted | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | -0.0679 | 0.7587 | 2.5607 | 4.8236 |
| 8 | -0.1923 | 0.8587 | 2.5326 | 4.5604 |
| 16 | -0.0818 | 0.7669 | 2.4443 | 4.2020 |
| 32 | -0.0433 | 0.7321 | 2.3024 | 4.0427 |
| 64 | -0.0104 | 0.7031 | 2.1874 | 3.6854 |

**Regression of Child Survivability by Removed Size, Inserted Size**

| Gen. | Removed | Inserted | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|---|
| 4 | -0.1566 | -0.0688 | 0.8991 | 2.5608 | 4.8236 |
| 8 | -0.4343 | -0.1902 | 1.1835 | 2.7469 | 4.5604 |
| 16 | -0.2163 | -0.0813 | 0.9354 | 2.4665 | 4.2020 |
| 32 | -0.1462 | -0.0423 | 0.8396 | 2.3089 | 4.0427 |
| 64 | -0.0711 | -0.0103 | 0.7579 | 2.1791 | 3.6854 |

**Legend.** Asterisks indicate statistically insignificant results (with an alpha value greater than 0.05). *Prescaled Dev/DF* indicates the deviance of the model, divided by the degrees of freedom, prior to scaling (at which time it approaches 1.0). *Intercept* and *Scale* are constants in the model function.

Table C.2: More Exploratory Regression Analysis of the Symbolic Regression Domain

**Multiple Regression of Child Survivability**
**by Depth, Parent Size, Removed Size, Inserted Size**

| Gen | Depth | Parent | Removed | Inserted | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|---|---|---|
| 4 | 0.1852 | 0.1104 | -0.1153 | -0.0304 | 0.5038 | 2.4793 | 4.7501 |
| 8 | 0.3745 | -0.1022 | -0.2140 | -0.1825 | 0.7016 | 2.6143 | 4.5322 |
| 16 | 0.5354 | -0.1691 | -0.2463 | -0.1884 | 0.5757 | 2.5345 | 4.2825 |
| 32 | 0.5448 | -0.2539 | -0.1559 | -0.0504 | 0.4959 | 2.4962 | 4.4755 |
| 64 | 0.5411 | -0.3333 | -0.1243 | -0.0266 | 0.5431 | 3.6327 | 4.5287 |

**Multiple Regression of Child Survivability**
**by Depth, Log(Parent Size), Log(Removed Size), Log(Inserted Size)**

| Gen | Depth | Parent | Removed | Inserted | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|---|---|---|
| 4 | 0.0432 | 0.2126 | -0.2688 | -0.1269 | 0.4752 | 2.4923 | 4.6827 |
| 8 | 0.2859 | *0.0326 | -0.3346 | -0.3024 | -0.1183 | 2.5732 | 4.3687 |
| 16 | 0.4931 | -0.0801 | -0.2429 | -0.2409 | -0.4383 | 2.4546 | 4.1566 |
| 32 | 0.4713 | -0.1536 | -0.1192 | -0.0824 | -0.2128 | 2.4493 | 4.4731 |
| 64 | 0.3858 | -0.1967 | -0.0836 | -0.0593 | -0.0720 | 2.4329 | 4.5663 |

**Regression of Removed Size**
**by Depth**

| Gen | Depth | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|
| 4 | -0.3678 | 0.4869 | 0.6396 | 0.4091 |
| 8 | -0.4633 | 0.5559 | 0.5659 | 0.3203 |
| 16 | -0.5221 | 0.6192 | 0.4438 | 0.1970 |
| 32 | -0.6351 | 0.5593 | 0.4396 | 0.1933 |
| 64 | -0.8571 | 0.5276 | 0.4626 | 0.2140 |

**Regression of Parent Size**
**by Depth**

| Gen | Depth | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|
| 4 | 0.5128 | 1.3668 | 1.2846 | 1.6503 |
| 8 | 0.4441 | 1.4632 | 1.3062 | 1.7063 |
| 16 | 0.3808 | 1.5221 | 1.4837 | 2.2015 |
| 32 | 0.4407 | 1.6351 | 2.0503 | 4.2041 |
| 64 | 0.4724 | 1.8571 | 2.7484 | 7.5544 |

**Legend.** Asterisks indicate statistically insignificant results (with an alpha value greater than 0.05). *Prescaled Dev/DF* indicates the deviance of the model, divided by the degrees of freedom, prior to scaling (at which time it approaches 1.0). *Intercept* and *Scale* are constants in the model function.

Table C.3: Exploratory Regression Analysis of the Symbolic Regression Domain with Inviable Code Restricted

**Regression of Child Survivability by Depth**

| Gen. | Depth | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | 0.2779 | 0.3839 | 2.4750 | 4.7757 |
| 8 | 0.3649 | 0.2809 | 2.5158 | 4.7164 |
| 16 | 0.5130 | 0.0955 | 2.5094 | 4.5311 |
| 32 | 0.4187 | 0.2272 | 2.4887 | 4.6262 |
| 64 | 0.3167 | 0.3523 | 2.4650 | 4.6847 |

**Regression of Child Survivability by Parent Size**

| Gen. | Parent | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | 0.1845 | 0.4980 | 2.5090 | 4.7757 |
| 8 | 0.1057 | 0.5850 | 2.5552 | 4.7164 |
| 16 | 0.2617 | 0.4215 | 2.5606 | 4.5311 |
| 32 | 0.1652 | 0.5238 | 2.5113 | 4.6262 |
| 64 | 0.0387 | 0.6542 | 2.4830 | 4.6847 |

**Regression of Child Survivability by Removed Size**

| Gen. | Removed | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | -0.1310 | 0.8113 | 2.5291 | 4.7757 |
| 8 | -0.3078 | 0.9397 | 2.6339 | 4.7164 |
| 16 | -0.3496 | 0.9487 | 2.5758 | 4.5311 |
| 32 | -0.2108 | 0.8423 | 2.5156 | 4.6262 |
| 64 | -0.1648 | 0.8008 | 2.6458 | 4.6847 |

**Regression of Child Survivability by Inserted Size**

| Gen. | Inserted | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | -0.0319 | 0.7244 | 2.5420 | 4.7757 |
| 8 | -0.1796 | 0.8489 | 2.5681 | 4.7164 |
| 16 | -0.1929 | 0.8524 | 2.5514 | 4.5311 |
| 32 | -0.0524 | 0.7403 | 2.5074 | 4.6262 |
| 64 | -0.0266 | 0.7172 | 2.4850 | 4.6847 |

**Regression of Child Survivability by Removed Size, Inserted Size**

| Gen. | Removed | Inserted | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|---|
| 4 | -0.1311 | -0.0322 | 0.8427 | 2.5337 | 4.7757 |
| 8 | -0.3088 | -0.1815 | 1.0979 | 2.6863 | 4.7164 |
| 16 | -0.3464 | -0.1885 | 1.1019 | 2.6161 | 4.5311 |
| 32 | -0.2106 | -0.0517 | 0.8888 | 2.5254 | 4.6262 |
| 64 | -0.1645 | -0.0260 | 0.8243 | 2.6453 | 4.6847 |

**Legend.** Asterisks indicate statistically insignificant results (with an alpha value greater than 0.05). *Prescaled Dev/DF* indicates the deviance of the model, divided by the degrees of freedom, prior to scaling (at which time it approaches 1.0). *Intercept* and *Scale* are constants in the model function.

Table C.4: More Exploratory Regression Analysis of the Symbolic Regression Domain with Inviable Code Restricted

**Multiple Regression of Child Survivability by Depth, Parent Size, Removed Size, Inserted Size**

| Gen | Depth | Parent | Removed | Inserted | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|---|---|---|
| 4 | 0.5861 | 0.1121 | -0.0398 | 0.0246 | -0.0920 | 2.5291 | 4.5488 |
| 8 | 0.4123 | -0.1078 | -0.1912 | 0.0186 | 0.4810 | 2.6392 | 4.7157 |
| 16 | 0.5441 | -0.2494 | -0.1076 | 0.0230 | 0.4093 | 2.5156 | 4.5584 |
| 32 | 0.6007 | -0.3066 | -0.0558 | 0.0080 | 0.3795 | 2.3085 | 4.0362 |
| 64 | 0.4074 | -0.2146 | -0.0399 | 0.0045 | 0.4992 | 1.7202 | 2.5757 |

**Multiple Regression of Child Survivability by Depth, Log(Parent Size), Log(Removed Size), Log(Inserted Size)**

| Gen | Depth | Parent | Removed | Inserted | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|---|---|---|
| 4 | 0.3234 | 0.2303 | -0.1362 | 0.0655 | 0.3482 | 2.5313 | 4.5120 |
| 8 | 0.3871 | -0.0521 | -0.2092 | 0.0391 | 0.1029 | 2.5338 | 4.7306 |
| 16 | 0.5069 | -0.1820 | -0.1497 | 0.0494 | 0.0052 | 2.5002 | 4.5631 |
| 32 | 0.5804 | -0.2442 | -0.0749 | 0.0129 | *-0.0456 | 2.3121 | 4.0489 |
| 64 | 0.4049 | -0.2110 | -0.0549 | 0.0096 | *0.1672 | 1.7253 | 2.5829 |

**Regression of Removed Size by Depth**

| Gen | Depth | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|
| 4 | -0.9333 | 0.1310 | 0.6794 | 0.4616 |
| 8 | -1.9229 | 0.6544 | 0.4121 | 0.1698 |
| 16 | -1.3266 | 0.6905 | 0.3522 | 0.1241 |
| 32 | -1.2333 | 0.7647 | 0.3140 | 0.0986 |
| 64 | -1.0214 | 0.7370 | 0.3406 | 0.1160 |

**Regression of Parent Size by Depth**

| Gen | Depth | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|
| 4 | 0.8690 | 1.9333 | 1.9655 | 3.8634 |
| 8 | 0.3456 | 2.9228 | 1.8325 | 3.3583 |
| 16 | 0.3095 | 2.3266 | 2.2633 | 5.1227 |
| 32 | 0.2353 | 2.2333 | 2.6967 | 7.2726 |
| 64 | 0.2630 | 2.0214 | 3.3274 | 11.0727 |

**Legend.** Asterisks indicate statistically insignificant results (with an alpha value greater than 0.05). *Prescaled Dev/DF* indicates the deviance of the model, divided by the degrees of freedom, prior to scaling (at which time it approaches 1.0). *Intercept* and *Scale* are constants in the model function.

Table C.5: Exploratory Regression Analysis of the 11-Multiplexer Domain

**Regression of Child Survivability by Depth**

| Gen. | Depth | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | 0.7118 | -0.1147 | 2.5323 | 4.5681 |
| 8 | 0.5315 | 0.1307 | 2.5396 | 4.8046 |
| 16 | 0.5128 | 0.1400 | 2.5095 | 4.6313 |
| 32 | 0.5430 | 0.1016 | 2.3137 | 4.0798 |
| 64 | 0.3651 | 0.3029 | 1.7311 | 2.6056 |

**Regression of Child Survivability by Parent Size**

| Gen. | Parent | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | 0.3317 | 0.3172 | 2.5400 | 4.5681 |
| 8 | *0.0427 | 0.6502 | 2.5471 | 4.8046 |
| 16 | 0.0978 | 0.5947 | 2.5107 | 4.6313 |
| 32 | 0.1269 | 0.5653 | 2.3316 | 4.0798 |
| 64 | 0.0813 | 0.6114 | 1.7373 | 2.6056 |

**Regression of Child Survivability by Removed Size**

| Gen. | Removed | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | -0.1180 | 0.7919 | 2.5423 | 4.5681 |
| 8 | -0.2623 | 0.8801 | 2.9196 | 4.8046 |
| 16 | -0.1638 | 0.8123 | 2.5472 | 4.6313 |
| 32 | -0.0943 | 0.7644 | 2.3266 | 4.0798 |
| 64 | -0.0568 | 0.7372 | 1.7279 | 2.6056 |

**Regression of Child Survivability by Inserted Size**

| Gen. | Inserted | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | 0.0245 | 0.6672 | 2.5414 | 4.5681 |
| 8 | 0.0186 | 0.6738 | 2.5457 | 4.8046 |
| 16 | 0.0227 | 0.6689 | 2.5075 | 4.6313 |
| 32 | 0.0085 | 0.6844 | 2.3298 | 4.0798 |
| 64 | *0.0038 | 0.6892 | 1.7346 | 2.6056 |

**Regression of Child Survivability by Removed Size, Inserted Size**

| Gen. | Removed | Inserted | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|---|
| 4 | -0.1181 | 0.0247 | 0.7658 | 2.5410 | 4.5681 |
| 8 | -0.2623 | 0.0186 | 0.8607 | 2.9226 | 4.8046 |
| 16 | -0.1636 | 0.0225 | 0.7882 | 2.5404 | 4.6313 |
| 32 | -0.0944 | 0.0088 | 0.7553 | 2.3260 | 4.0798 |
| 64 | -0.0569 | *0.0043 | 0.7328 | 1.7279 | 2.6056 |

**Legend.** Asterisks indicate statistically insignificant results (with an alpha value greater than 0.05). *Prescaled Dev/DF* indicates the deviance of the model, divided by the degrees of freedom, prior to scaling (at which time it approaches 1.0). *Intercept* and *Scale* are constants in the model function.

Table C.6: More Exploratory Regression Analysis of the 11-Multiplexer Domain

**Multiple Regression of Child Survivability**
**by Depth, Parent Size, Removed Size, Inserted Size**

| Gen | Depth | Parent | Removed | Inserted | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|---|---|---|
| 4 | 0.6142 | 0.1166 | -0.0278 | 0.0229 | -0.1389 | 2.5311 | 4.5308 |
| 8 | 0.5249 | -0.1686 | -0.1463 | 0.0242 | 0.3794 | 2.5263 | 4.6687 |
| 16 | 0.4855 | *-0.0550 | -0.1137 | 0.0161 | 0.2904 | 2.4919 | 4.4912 |
| 32 | 0.5279 | -0.2069 | -0.0876 | 0.0113 | 0.3813 | 2.3063 | 4.0891 |
| 64 | 0.3863 | -0.1271 | -0.0262 | 0.0048 | 0.4279 | 1.5313 | 2.2049 |

**Multiple Regression of Child Survivability**
**by Depth, Log(Parent Size), Log(Removed Size), Log(Inserted Size)**

| Gen | Depth | Parent | Removed | Inserted | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|---|---|---|
| 4 | 0.3581 | 0.2324 | -0.1158 | 0.0552 | 0.3156 | 2.5323 | 4.4940 |
| 8 | 0.4694 | -0.0828 | -0.1688 | 0.0535 | 0.0514 | *2.5199 | 4.6876 |
| 16 | 0.4638 | *0.0085 | -0.1364 | 0.0296 | 0.0574 | *2.4609 | 4.5007 |
| 32 | 0.5182 | -0.1822 | -0.1082 | 0.0236 | -0.0058 | *2.2999 | 4.0986 |
| 64 | 0.3775 | -0.0902 | -0.0405 | 0.0093 | 0.2307 | 1.5350 | 2.2102 |

**Regression of Removed Size**
**by Depth**

| Gen | Depth | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|
| 4 | -0.9358 | 0.1403 | 0.6802 | 0.4627 |
| 8 | -1.8267 | 0.5608 | 0.4471 | 0.1999 |
| 16 | -1.5359 | 0.7067 | 0.3537 | 0.1251 |
| 32 | -1.3886 | 0.7328 | 0.3229 | 0.1043 |
| 64 | -1.0578 | 0.7406 | 0.3402 | 0.1157 |

**Regression of Parent Size**
**by Depth**

| Gen | Depth | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|
| 4 | 0.8596 | 1.9346 | 1.9473 | 3.7922 |
| 8 | 0.4392 | 2.8268 | 1.9132 | 3.6607 |
| 16 | 0.2933 | 2.5359 | 2.2467 | 5.0479 |
| 32 | 0.2672 | 2.3886 | 2.7406 | 7.5116 |
| 64 | 0.2594 | 2.0578 | 3.5214 | 12.4015 |

**Legend.** Asterisks indicate statistically insignificant results (with an alpha value greater than 0.05). *Prescaled Dev/DF* indicates the deviance of the model, divided by the degrees of freedom, prior to scaling (at which time it approaches 1.0). *Intercept* and *Scale* are constants in the model function.

Table C.7: Exploratory Regression Analysis of the 11-Multiplexer Domain with Inviable Code Restricted

**Regression of Child Survivability by Depth**

| Gen. | Depth | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | 0.7320 | -0.1393 | 2.5329 | 4.5492 |
| 8 | 0.5507 | 0.1044 | 2.5300 | 4.7481 |
| 16 | 0.5408 | 0.1109 | 2.4707 | 4.5432 |
| 32 | 0.5110 | 0.1421 | 2.3116 | 4.1436 |
| 64 | 0.3663 | 0.3049 | 1.5336 | 2.2196 |

**Regression of Child Survivability by Parent Size**

| Gen. | Parent | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | 0.3416 | 0.3051 | 2.5394 | 4.5492 |
| 8 | 0.0641 | 0.6285 | 2.5426 | 4.7481 |
| 16 | 0.2156 | 0.4740 | 2.4790 | 4.5432 |
| 32 | 0.1520 | 0.5397 | 2.3234 | 4.1436 |
| 64 | 0.1137 | 0.5786 | 1.5427 | 2.2196 |

**Regression of Child Survivability by Removed Size**

| Gen. | Removed | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | -0.1050 | 0.7821 | 2.5550 | 4.5492 |
| 8 | -0.2377 | 0.8631 | 2.5900 | 4.7481 |
| 16 | -0.1674 | 0.8137 | 2.5761 | 4.5432 |
| 32 | -0.1254 | 0.7818 | 2.3253 | 4.1436 |
| 64 | -0.0390 | 0.7249 | 1.5400 | 2.2196 |

**Regression of Child Survivability by Inserted Size**

| Gen. | Inserted | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | 0.0221 | 0.6698 | 2.5467 | 4.5492 |
| 8 | 0.0238 | 0.6681 | 2.5401 | 4.7481 |
| 16 | 0.0155 | 0.6769 | 2.4773 | 4.5432 |
| 32 | 0.0114 | 0.6813 | 2.3184 | 4.1436 |
| 64 | 0.0050 | 0.6880 | 1.5424 | 2.2196 |

**Regression of Child Survivability by Removed Size, Inserted Size**

| Gen. | Removed | Inserted | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|---|
| 4 | -0.1055 | 0.0228 | 0.7586 | 2.5522 | 4.5492 |
| 8 | -0.2377 | 0.0238 | 0.8380 | 2.5831 | 4.7481 |
| 16 | -0.1675 | 0.0160 | 0.7971 | 2.5750 | 4.5432 |
| 32 | -0.1254 | 0.0115 | 0.7698 | 2.3245 | 4.1436 |
| 64 | -0.0390 | 0.0048 | 0.7200 | 1.5399 | 2.2196 |

**Legend.** Asterisks indicate statistically insignificant results (with an alpha value greater than 0.05). *Prescaled Dev/DF* indicates the deviance of the model, divided by the degrees of freedom, prior to scaling (at which time it approaches 1.0). *Intercept* and *Scale* are constants in the model function.

Table C.8: More Exploratory Regression Analysis of the 11-Multiplexer Domain with Inviable Code Restricted

**Multiple Regression of Child Survivability**
**by Depth, Parent Size, Removed Size, Inserted Size**

| Gen | Depth | Parent | Removed | Inserted | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|---|---|---|
| 4 | 0.5116 | 0.1859 | -0.0523 | 0.0155 | -0.0573 | *2.5139 | 4.5375 |
| 8 | 0.4647 | -0.0611 | -0.2028 | 0.0174 | 0.3778 | 2.5124 | 4.5441 |
| 16 | 0.5388 | -0.2001 | -0.1094 | 0.0201 | 0.3732 | 2.2129 | 3.7820 |
| 32 | 0.3184 | *0.0054 | -0.0741 | *-0.0062 | 0.4066 | 1.1964 | 1.6350 |
| 64 | 0.1812 | *-0.0404 | *-0.0239 | *-0.0069 | 0.5729 | 1.1427 | 1.4647 |

**Multiple Regression of Child Survivability**
**by Depth, Log(Parent Size), Log(Removed Size), Log(Inserted Size)**

| Gen | Depth | Parent | Removed | Inserted | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|---|---|---|
| 4 | 0.2670 | 0.2778 | -0.1408 | 0.0370 | 0.3954 | 2.5195 | 4.4990 |
| 8 | 0.4367 | *-0.0046 | -0.2166 | 0.0415 | 0.0349 | *2.4849 | 4.5585 |
| 16 | 0.5270 | -0.2066 | -0.1421 | 0.0341 | -0.0182 | *2.2104 | 3.7851 |
| 32 | 0.3295 | *0.0067 | -0.0647 | *-0.0094 | 0.2120 | 1.1997 | 1.6559 |
| 64 | 0.1982 | *-0.0652 | *-0.0199 | *-0.0191 | 0.3860 | 1.1435 | 1.4699 |

**Regression of Removed Size**
**by Depth**

| Gen | Depth | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|
| 4 | -1.1544 | 0.1919 | 0.6789 | 0.4609 |
| 8 | -2.0979 | 0.7001 | 0.4269 | 0.1822 |
| 16 | -1.5757 | 0.7283 | 0.3136 | 0.0984 |
| 32 | -1.0197 | 0.6739 | 0.4188 | 0.1757 |
| 64 | -0.8889 | 0.5181 | 0.4527 | 0.2058 |

**Regression of Parent Size**
**by Depth**

| Gen | Depth | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|
| 4 | 0.8081 | 2.1546 | 1.9051 | 3.6297 |
| 8 | 0.2999 | 3.0979 | 1.9453 | 3.7844 |
| 16 | 0.2717 | 2.5757 | 2.2518 | 5.0715 |
| 32 | 0.3261 | 2.0197 | 2.8418 | 8.0866 |
| 64 | 0.4819 | 1.8889 | 3.6929 | 13.6913 |

**Legend.** Asterisks indicate statistically insignificant results (with an alpha value greater than 0.05). *Prescaled Dev/DF* indicates the deviance of the model, divided by the degrees of freedom, prior to scaling (at which time it approaches 1.0). *Intercept* and *Scale* are constants in the model function.

Table C.9: Exploratory Regression Analysis of the 11-Multiplexer Domain

**Regression of Child Survivability by Depth**

| Gen. | Depth | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | 0.7070 | -0.0930 | 2.5206 | 4.5744 |
| 8 | 0.6073 | *0.0460 | 2.4928 | 4.6319 |
| 16 | 0.5420 | 0.1116 | 2.2220 | 3.8440 |
| 32 | 0.3566 | 0.3108 | 1.2027 | 1.6701 |
| 64 | 0.1781 | 0.5088 | 1.1410 | 1.4685 |

**Regression of Child Survivability by Parent Size**

| Gen. | Parent | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | 0.3478 | 0.3020 | 2.5127 | 4.5744 |
| 8 | 0.0805 | 0.6119 | 2.5005 | 4.6319 |
| 16 | 0.1431 | 0.5488 | 2.2333 | 3.8440 |
| 32 | 0.1890 | 0.5001 | 1.2181 | 1.6701 |
| 64 | *0.0709 | 0.6215 | 1.1400 | 1.4685 |

**Regression of Child Survivability by Removed Size**

| Gen. | Removed | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | -0.1222 | 0.7955 | 2.5341 | 4.5744 |
| 8 | -0.2859 | 0.8870 | 2.5813 | 4.6319 |
| 16 | -0.1664 | 0.8137 | 2.2360 | 3.8440 |
| 32 | -0.0931 | 0.7621 | 1.2114 | 1.6701 |
| 64 | -0.0300 | 0.7182 | 1.1386 | 1.4685 |

**Regression of Child Survivability by Inserted Size**

| Gen. | Inserted | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | 0.0149 | 0.6780 | 2.5292 | 4.5744 |
| 8 | 0.0166 | 0.6759 | 2.4974 | 4.6319 |
| 16 | 0.0187 | 0.6733 | 2.2287 | 3.8440 |
| 32 | *-0.0082 | 0.7011 | 1.2200 | 1.6701 |
| 64 | *-0.0066 | 0.6994 | 1.1378 | 1.4685 |

**Regression of Child Survivability by Removed Size, Inserted Size**

| Gen. | Removed | Inserted | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|---|
| 4 | -0.1224 | 0.0154 | 0.7798 | 2.5329 | 4.5744 |
| 8 | -0.2860 | 0.0168 | 0.8696 | 2.5796 | 4.6319 |
| 16 | -0.1667 | 0.0194 | 0.7933 | 2.2341 | 3.8440 |
| 32 | -0.0930 | *-0.0073 | 0.7691 | 1.2113 | 1.6701 |
| 64 | -0.0304 | *-0.0076 | 0.7257 | 1.1400 | 1.4685 |

**Legend.** Asterisks indicate statistically insignificant results (with an alpha value greater than 0.05). *Prescaled Dev/DF* indicates the deviance of the model, divided by the degrees of freedom, prior to scaling (at which time it approaches 1.0). *Intercept* and *Scale* are constants in the model function.

Table C.10: More Exploratory Regression Analysis of the t-Multiplexer Domain Restricted

**Multiple Regression of Child Survivability**
**by Depth, Parent Size, Removed Size, Inserted Size**

| Gen | Depth | Parent | Removed | Inserted | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|---|---|---|
| 4 | 0.3015 | 0.2096 | -0.0987 | 0.0402 | 0.1621 | 2.5107 | 4.5705 |
| 8 | 0.4823 | -0.1285 | -0.1643 | 0.0340 | 0.3750 | 2.5061 | 4.4502 |
| 16 | 0.4299 | -0.1032 | -0.1315 | 0.0250 | 0.3973 | 2.1435 | 3.6314 |
| 32 | 0.3006 | -0.1207 | -0.0388 | 0.0022 | *0.5274 | 1.2459 | 1.7070 |
| 64 | 0.1119 | *-0.0141 | -0.0096 | -0.0004 | *0.6017 | 1.0760 | 1.3338 |

**Multiple Regression of Child Survivability**
**by Depth, Log(Parent Size), Log(Removed Size), Log(Inserted Size)**

| Gen | Depth | Parent | Removed | Inserted | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|---|---|---|
| 4 | *0.0002 | 0.3357 | -0.2323 | 0.0819 | 0.6526 | 2.5026 | 4.5083 |
| 8 | 0.4533 | -0.0761 | -0.1677 | 0.0624 | 0.0469 | *2.4468 | 4.4621 |
| 16 | 0.4051 | *-0.0514 | -0.1427 | 0.0411 | 0.0894 | 2.1368 | 3.6399 |
| 32 | 0.2912 | -0.0902 | -0.0438 | 0.0046 | *0.3039 | 1.2469 | 1.7163 |
| 64 | 0.1069 | *-0.0004 | -0.0168 | -0.0023 | *0.5364 | 1.0757 | 1.3341 |

**Regression of Removed Size**
**by Depth**

| Gen | Depth | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|
| 4 | -1.0538 | 0.3131 | 0.7042 | 0.4960 |
| 8 | -1.5924 | 0.6657 | 0.4458 | 0.1988 |
| 16 | -1.3793 | 0.6800 | 0.3625 | 0.1315 |
| 32 | -1.0265 | 0.6501 | 0.3980 | 0.1584 |
| 64 | -0.6048 | 0.5823 | 0.4631 | 0.2145 |

**Regression of Parent Size**
**by Depth**

| Gen | Depth | Intercept | Scale | Prescaled Dev/DF |
|---|---|---|---|---|
| 4 | 0.6869 | 2.0538 | 1.8672 | 3.4866 |
| 8 | 0.3343 | 2.5915 | 1.8532 | 3.4345 |
| 16 | 0.3200 | 2.3793 | 2.2845 | 5.2196 |
| 32 | 0.3499 | 2.0265 | 2.9996 | 8.9995 |
| 64 | 0.4177 | 1.6048 | 3.7426 | 14.0102 |

**Legend.** Asterisks indicate statistically insignificant results (with an alpha value greater than 0.05). *Prescaled Dev/DF* indicates the deviance of the model, divided by the degrees of freedom, prior to scaling (at which time it approaches 1.0). *Intercept* and *Scale* are constants in the model function.

Table C.11: Exploratory Regression Analysis of the 6-Multiplexer Domain with Inviable Code Restricted

**Regression of Child Survivability by Depth**

| Gen. | Depth | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | 0.5242 | 0.1109 | 2.5261 | 4.6397 |
| 8 | 0.5500 | 0.0960 | 2.4639 | 4.5416 |
| 16 | 0.4716 | 0.1848 | 2.1523 | 3.7110 |
| 32 | 0.2795 | 0.3992 | 1.2497 | 1.7288 |
| 64 | 0.1107 | 0.5794 | 1.0757 | 1.3357 |

**Regression of Child Survivability by Parent Size**

| Gen. | Parent | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | 0.3027 | 0.3566 | 2.5167 | 4.6397 |
| 8 | 0.0787 | 0.6138 | 2.4773 | 4.5416 |
| 16 | 0.1638 | 0.5272 | 2.1557 | 3.7110 |
| 32 | 0.0655 | 0.6272 | 1.2510 | 1.7288 |
| 64 | 0.0579 | 0.6347 | 1.0755 | 1.3357 |

**Regression of Child Survivability by Removed Size**

| Gen. | Removed | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | -0.1406 | 0.8073 | 2.5371 | 4.6397 |
| 8 | -0.2555 | 0.8708 | 2.6376 | 4.5416 |
| 16 | -0.1799 | 0.8178 | 2.1570 | 3.7110 |
| 32 | -0.0546 | 0.7373 | 1.2483 | 1.7288 |
| 64 | -0.0122 | 0.7044 | 1.0758 | 1.3357 |

**Regression of Child Survivability by Inserted Size**

| Gen. | Inserted | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|
| 4 | 0.0400 | 0.6496 | 2.5209 | 4.6397 |
| 8 | 0.0362 | 0.6540 | 2.4727 | 4.5416 |
| 16 | 0.0250 | 0.6662 | 2.1479 | 3.7110 |
| 32 | *0.0030 | 0.6901 | 1.2492 | 1.7288 |
| 64 | *-0.0004 | 0.6935 | 1.0754 | 1.3357 |

**Regression of Child Survivability by Removed Size, Inserted Size**

| Gen. | Removed | Inserted | Intercept | Scale | Prescaled Dev/DoF |
|---|---|---|---|---|---|
| 4 | -0.1400 | 0.0392 | 0.7642 | 2.5322 | 4.6397 |
| 8 | -0.2550 | 0.0351 | 0.8324 | 2.6221 | 4.5416 |
| 16 | -0.1799 | 0.0252 | 0.7906 | 2.1506 | 3.7110 |
| 32 | -0.0545 | *0.0025 | 0.7348 | 1.2484 | 1.7288 |
| 64 | -0.0122 | *-0.0004 | 0.7048 | 1.0758 | 1.3357 |

**Legend.** Asterisks indicate statistically insignificant results (with an alpha value greater than 0.05). *Prescaled Dev/DF* indicates the deviance of the model, divided by the degrees of freedom, prior to scaling (at which time it approaches 1.0). *Intercept* and *Scale* are constants in the model function.

Table C.12: More Exploratory Regression Analysis of the 6-Multiplexer Domain with Inviable Code Restricted

# Acknowledgements

During my research I've had several mentors to whom I am grateful. First and foremost, as my graduate advisor, Jim Hendler saw fit to give me wide berth to pursue many research areas. The work in this thesis, as well as my work in knowledge representation and in biological modelling, are the direct result of his guidance. I cannot thank him enough. Lee Spector got me working in this area and has been a constant source of help, creative inspiration, and critical analysis. I also owe him a considerable debt of gratitude. Jim Reggia's open door has been abused by me more than a few times, as has been his wit and ability to see the essence of the matter. And Bill Gasarch's frankness has kept me on the straight and narrow.

Thanks to Chau-Wen Tseng, Stephen Mount, and Dana Nau for serving on my committee, to Chuck Delwiche for his help in understanding biological nuances, and to Chip Denman who was exceptionally patient in helping me work through the statistical intricacies of multiple poisson regression. Thanks also to Bill Langdon, Peter Angeline, and Matt Evett.

While I was a student at Maryland, I worked with a number of friends who helped me as I plugged away, notably Brian Kettler, Reiko Tsuneto, Kutluhan Erol, Bill Andersen, Bob Kohout, Jeff Heflin, Jaime Montemayor, Houman Alborzi, Brian Sullivan, and Dave Rager. Edna Walker kept me from falling apart due to my own idiocy, and Nancy Lindley bashed open doors whenever I or other graduate students couldn't get through them. In addition to commiseration, Brian Postow went through the entire dissertation with a fine-tooth comb. Leslie Kelling, Sam Sun, and Laura Drake offered much friendship and support through stressful times. And thanks to Glen Henshaw and the General Motors Truck Division, who helped me out in more scrapes than I can count, and who has been there for me for years on end.

The RoboCup project could not have been done without the help of my team of undergraduates, notably Charles Hohn, Jonathan Farris, and Gary Jackson, and to the supercomputer staff who handed us extra time under the table. Much of the RoboCup work and language induction work is also due to efforts by Minoru Asada at Osaka University and Hiroaki Kitano at Sony CSL. Kitano-san also was kind enough to let me stay in Japan at Sony for five months while stewing over parts of this research, and Asada-san gave me the chance to meet researchers throughout Japan. Thanks also to Kim Binstead at Sony for her ideas and friendship.

Last but far from least, I'm forever grateful to my mom and dad, who have been guides for all my life, and particularly during the preparation of the thesis. Thanks also to the help from my brothers Ryan and Tyler, and to my sister Nicole.

# Bibliography


[ Aho, Sethi, and Ullman 1988] Aho, A. V., Sethi, R. and Ullman, J. D. 1988. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

[ Andre and Teller 1996] Andre, D. and Teller, A. 1996. A study in program response and the negative effects of introns in genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B. and Riolo, R. L., eds., *Genetic Programming 1996: Proceedings of the First Annual Conference*, 12–20. Stanford University, CA, USA: MIT Press.

[ Andre, Bennett III, and Koza 1996] Andre, D., Bennett III, F. H. and Koza, J. R. 1996. Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem. In Koza, J. R., Goldberg, D. E., Fogel, D. B. and Riolo, R. L., eds., *Genetic Programming 1996: Proceedings of the First Annual Conference*, 3–11. Stanford University, CA, USA: MIT Press.

[ Andre 1995] Andre, D. 1995. The evolution of agents that build mental models and create simple plans using genetic programming. In Eshelman, L., ed., *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, 248–255. Pittsburgh, PA, USA: Morgan Kaufmann.

[ Angeline and Pollack 1993] Angeline, P. J. and Pollack, J. B. 1993. Competitive environments evolve better solutions for complex tasks. In Forrest, S., ed., *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, 264–270. University of Illinois at Urbana-Champaign: Morgan Kaufmann.

[ Angeline, Saunders, and Pollack 1994] Angeline, P. J., Saunders, G. M. and Pollack, J. B. 1994. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks* 54–65.

[ Angeline 1994] Angeline, P. J. 1994. Genetic programming and emergent intelligence. In Kinnear, Jr., K. E., ed., *Advances in Genetic Programming*. MIT Press. 75–98.

[ Angeline 1996] Angeline, P. J. 1996. Two self-adaptive crossover operators for genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., eds., *Advances in Genetic Programming 2*. Cambridge, MA, USA: MIT Press. 89–110.

[ Angeline 1997] Angeline, P. J. 1997. Subtree crossover: Building block engine or macromutation? In Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H. and Riolo, R. L., eds., *Genetic Programming 1997: Proceedings of the Second Annual Conference*, 9–17. Stanford University, CA, USA: Morgan Kaufmann.

[ Angeline 1998] Angeline, P. J. 1998. Subtree crossover causes bloat. In Koza, J. R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M. H., Goldberg, D. E., Iba, H. and Riolo, R., eds., *Genetic Programming 1998: Proceedings of the Third Annual Conference*, 745–752. University of Wisconsin, Madison, Wisconsin, USA: Morgan Kaufmann.

[ Banzhaf *et al.* 1998] Banzhaf, W., Nordin, P., Keller, R. E. and Francone, F. D. 1998. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann.

[ Blickle and Thiele 1994] Blickle, T. and Thiele, L. 1994. Genetic programming and redundancy. In Hopf, J., ed., *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)*, 33–38. Saarbrücken,Germany: Max-Planck-Institut für Informatik.

[ Blickle 1996] Blickle, T. 1996. *Theory of Evolutionary Algorithms and Application to System Synthesis*. Ph.D. Dissertation, Swiss Federal Institute of Technology, Zurich.

[ Boers *et al.* 1993] Boers, E. J. W., Kuiper, H., M., B. L. and Sprinkhuizen-Kuyper, I. G. 1993. Designing modular artificial networks. In Wijshoff, H. A., ed., *Proceedings of Computing Science in The Netherlands*, 154–159. Amsterdam: SION, Stichting Mathematisch Centrum.

[ Bohm and Geyer-Schulz 1996] Bohm, W. and Geyer-Schulz, A. 1996. Exact uniform initialization for genetic programming. In Belew, R. K. and Vose, M., eds., *Foundations of Genetic Algorithms IV*, 379–407. University of San Diego, CA, USA: Morgan Kaufmann.

[ Brave 1996] Brave, S. 1996. Evolving deterministic finite automata using cellular encoding. In Koza, J. R., Goldberg, D. E., Fogel, D. B. and Riolo, R. L., eds., *Genetic Programming 1996: Proceedings of the First Annual Conference*, 39–44. Stanford University, CA, USA: MIT Press.

[ Brown 1992] Brown, T. A. 1992. *Genetics: A Molecular Approach*. Chapman and Hall.

[ Burke *et al.* 1998] Burke, D., Dejong, K., Grefenstette, J., Ramsey, C. and Wu, A. 1998. Putting more genetics into genetic algorithms. *Evolutionary Computation* 6(4).

[ Chellapilla 1997] Chellapilla, K. 1997. Evolving computer programs without subtree crossover. *IEEE Transactions on Evolutionary Computation* 1(3):209–216.

[ Chong and Langdon 1999] Chong, F. S. and Langdon, W. B. 1999. Java based distributed genetic programming on the internet. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M. and Smith, R. E., eds., *Proceedings of the Genetic and Evolutionary Computation Conference*, Volume 2, 1229. Orlando, Florida, USA: Morgan Kaufmann.

[ Chong 1998] Chong, F. S. 1998. A java based distributed approach to genetic programming on the internet. Master's thesis, Computer Science, University of Birmingham.

[ Cohoon *et al.* 1987] Cohoon, J. P., Hegde, S. U., Martin, W. N. and Richards, D. S. 1987. Punctuated equilibria: a parallel genetic algorithm. In Grefenstette, J. J., ed., *Proceedings of the Second International Conference on Genetic Algorithms*, 148–154. Erlbaum.

[ Collins and Jefferson 1991] Collins, R. and Jefferson, D. 1991. An artificial neural network representation for artificial organisms. In Schwefel, H. P. and Mäanner, R., eds., *Parallel Problem Solving from Nature*, 269–263. Berlin: Springer-Verlag.

[ Cramer 1985] Cramer, N. L. 1985. A representation for the adaptive generation of simple sequential programs. In Grefenstette, J. J., ed., *Proceedings of an International Conference on Genetic Algorithms and the Applications*, 183–187.

[ Daida *et al.* 1997] Daida, J., Ross, S., McClain, J., Ampy, D. and Holczer, M. 1997. Challenges with verification, repeatability, and meaningful comparisons in genetic programming. In Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H. and Riolo, R. L., eds., *Genetic Programming 1997: Proceedings of the Second Annual Conference*, 64–69. Stanford University, CA, USA: Morgan Kaufmann.

[ DeJong 1989] DeJong, K. 1989. Using genetic algorithms to learn task programs: the pitt approach. *Machine Learning* 2(2-3).

[ Delwiche 2000] Delwiche, C. 2000. Discussion with the Author, July 2000.

[ Fogel, Owens, and Walsh 1966] Fogel, L. J., Owens, A. J. and Walsh, M. J. 1966. *Artificial Intelligence through Simulated Evolution*. New York: John Wiley.

[ Fonseca and Fleming 1993] Fonseca, C. M. and Fleming, P. J. 1993. Genetic algorithms for multiobjective optimization: formulation, discussion, and generalization. In Forrest, S., ed., *Proceedings of the Fifth International Conference on Genetic Algorithms*, 416–423. Morgan Kaufmann.

[ Fourman 1985] Fourman, M. P. 1985. Compaction of symbolic layout using genetic algorithms. In Grefenstette, J. J., ed., *Genetic Algorithms and Their Uses: Proceedings of the First International Conference on Genetic Algorithms*, 141–153. Erlbaum.

[ Francone *et al.* 1999] Francone, F. D., Conrads, M., Banzhaf, W. and Nordin, P. 1999. Homologous crossover in genetic programming. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M. and Smith, R. E., eds., *Proceedings of the Genetic and Evolutionary Computation Conference*, Volume 2, 1021–1026. Orlando, Florida, USA: Morgan Kaufmann.

[ Fullmer and Miikkulainen 1991] Fullmer, B. and Miikkulainen, R. 1991. Using marker-based genetic encoding of neural networks to evolve finite-state behavior. In *Proceedings of the First European Conference on Artificial Life (ECAL-91), Paris*.

[ Gathercole and Ross 1996] Gathercole, C. and Ross, P. 1996. An adverse interaction between crossover and restricted tree depth in genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B. and Riolo, R. L., eds., *Genetic Programming 1996: Proceedings of the First Annual Conference*, 291–296. Stanford University, CA, USA: MIT Press.

[ Geyer-Schulz 1995] Geyer-Schulz, A. 1995. *Fuzzy Rule-Based Expert Systems and Genetic Machine Learning*, Volume 3 of *Studies in Fuzziness*. Heidelberg: Physica-Verlag.

[ Goldberg *et al.* 1993] Goldberg, D. E., Deb, K., Kargupta, H. and Harik, G. 1993. Rapid, accuate optimization of difficult problems using fast messy genetic algorithms. In *Proceedings of the Fifth Annual International Conference of Genetic Algorithms (ICGA-93)*, 56–64.

[ Goldberg 1989] Goldberg, D. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading: Addison-Wesley.

[ Grefenstette and Baker 1989] Grefenstette, J. J. and Baker, J. E. 1989. How genetic algorithms work: a critical look at implicit parallelism. In Schaffer, J. D., ed., *Proceedings of the Third International Conference on Genetic Algorithms*, 20–27. Morgan Kaufmann.

[ Gruau 1992] Gruau, F. 1992. Genetic synthesis of boolean neural networks with a cell rewriting developmental process. In Schaffer, J. D. and Whitley, D., eds., *Proceedings of the Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN92)*, 55–74. The IEEE Computer Society Press.

[ Hajela and Lin 1992] Hajela, P. and Lin, C.-Y. 1992. Genetic search strategies in multicriterion optimal design. *Structural Optimization* 4:99–107.

[ Handley 1994] Handley, S. 1994. On the use of a directed acyclic graph to represent a population of computer programs. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, 154–159. Orlando, Florida, USA: IEEE Press.

[ Haynes *et al.* 1995] Haynes, T., Sen, S., Schoenefeld, D. and Wainwright, R. 1995. Evolving a team. In Siegel, E. V. and Koza, J. R., eds., *Working Notes for the AAAI Symposium on Genetic Programming*, 23–30. MIT, Cambridge, MA, USA: AAAI.

[ Haynes, Schoenefeld, and Wainwright 1996] Haynes, T. D., Schoenefeld, D. A. and Wainwright, R. L. 1996. Type inheritance in strongly typed genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., eds., *Advances in Genetic Programming 2*. Cambridge, MA, USA: MIT Press. 359–376.

[ Haynes 1997] Haynes, T. 1997. Phenotypical building blocks for genetic programming. In Back, T., ed., *Genetic Algorithms: Proceedings of the Seventh International Conference*, 26–33. Michigan State University, East Lansing, MI, USA: Morgan Kaufmann.

[ Haynes 1998] Haynes, T. 1998. Collective adaptation: The exchange of coding segments. *Evolutionary Computation* 6(4):311–338.

[ Hinterding, Gielewski, and Peachey 1995] Hinterding, R., Gielewski, H. and Peachey, T. 1995. The nature of mutation in genetic algorithms. In Eshelman, L. J., ed., *Proceedings of the Sixth International Conference on Genetic Algorithms*, 65–72. Morgan Kaufmann.

[ Hohn 1997] Hohn, C. 1997. Evolving predictive functions from observed data for simulated robots. Senior honor's thesis, Department of Computer Science, University of Maryland.

[ Holland 1975] Holland, J. 1975. *Adaption in Natural and Artificial Systems*. University of Michigan Press.

[ Howley 1996] Howley, B. 1996. Genetic programming of near-minimum-time spacecraft attitude maneuvers. In Koza, J. R., Goldberg, D. E., Fogel, D. B. and Riolo, R. L., eds., *Genetic Programming 1996: Proceedings of the First Annual Conference*, 98–106. Stanford University, CA, USA: MIT Press.

[ Iba and de Garis 1996] Iba, H. and de Garis, H. 1996. Extending genetic programming with recombinative guidance. In Angeline, P. J. and Kinnear, Jr., K. E., eds., *Advances in Genetic Programming 2*. Cambridge, MA, USA: MIT Press. 69–88.

[ Iba and Terao 2000] Iba, H. and Terao, M. 2000. Controlling effective introns for multi-agent learning by genetic programming. In Whitley, D., Goldberg, D., Cantú-Paz, E., Spector, L., Parmee, I. and Beyer, H.-G., eds., *GECCO-2000: Proceedings of the Genetic and Evolutionary Computat9ion Conference*. San Fransisco: Morgan Kaufmann. 419–426.

[ Iba, de Garis, and Sato 1994] Iba, H., de Garis, H. and Sato, T. 1994. Genetic programming using a minimum description length principle. In Kinnear, Jr., K. E., ed., *Advances in Genetic Programming*. MIT Press. 265–284.

[ Iba 1996a] Iba, H. 1996a. Emergent cooperation for multiple agents using genetic programming. In Koza, J. R., ed., *Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996*, 66–74. Stanford University, CA, USA: Stanford Bookstore.

[ Iba 1996b] Iba, H. 1996b. Random tree generation for genetic programming. In Voigt, H.-M., Ebeling, W., Rechenberg, I. and Schwefel, H.-P., eds., *Parallel Problem Solving from Nature IV, Proceedings of the International Conference on Evolutionary Computation*, Volume 1141 of *LNCS*, 144–153. Berlin, Germany: Springer Verlag.

[ Igel and Chellapilla 1999] Igel, C. and Chellapilla, K. 1999. Investigating the influence of depth and degree of genotypic change on fitness in genetic programming. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M. and Smith, R. E., eds., *Proceedings of the Genetic and Evolutionary Computation Conference*, Volume 2, 1061–1068. Orlando, Florida, USA: Morgan Kaufmann.

[ Itsuki 1995] Itsuki, N. 1995. Soccer server: a simulator for robocup. In *Proceedings of the 1995 Artificial Intelligence Symposium, Japanese Society for Artificial Intelligence, Special Session on RoboCup*.

[ Jakobi 1995] Jakobi, N. 1995. Harnessing morphogenesis. In *International Conference on Information Processing in Cells and Tissues*.

[ Jones and Joines 1999] Jones, E. A. and Joines, W. T. 1999. Genetic design of electronic circuits. In Brave, S. and Wu, A. S., eds., *Late Breaking Papers at the 1999 Genetic and Evolutionary Computation Conference*, 125–133.

[ Kargupta 1996] Kargupta, H. 1996. The gene expression messy genetic algorithm. In *Proceedings of the IEEE International Conference on Evolutionary Computation*.

[ Keijzer 1996] Keijzer, M. 1996. Efficiently representing populations in genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., eds., *Advances in Genetic Programming 2*. Cambridge, MA, USA: MIT Press. 259–278.

[ Kitano *et al.* 1995] Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I. and Osawa, E. 1995. Robocup: The robot world cup initiative. In *Proceedings of the Workshop on Entertainment and AI/Alife, International Joint Conferences in Artificial Intelligence*.

[ Kitano 1990] Kitano, H. 1990. Designing neural networks using a genetic algorithm with a graph generation system. *Complex Systems* 4:461–476.

[ Koza and Rice 1991] Koza, J. R. and Rice, J. P. 1991. Genetic generation of both the weights and architecture for a neural network. In *International Joint Conference on Neural Networks, IJCNN-91*, Volume 2, 397–404. Washington State Convention and Trade Center, Seattle, WA, USA: IEEE Computer Society Press.

[ Koza *et al.* 1997a] Koza, J. R., Bennett III, F. H., Keane, M. A. and Andre, D. 1997a. Evolution of a time-optimal fly-to controller circuit using genetic programming. In Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H. and Riolo, R. L., eds., *Genetic Programming 1997: Proceedings of the Second Annual Conference*, 207–212. Stanford University, CA, USA: Morgan Kaufmann.

[ Koza *et al.* 1997b] Koza, J. R., Bennett III, F. H., Hutchings, J. L., Bade, S. L., Keane, M. A. and Andre, D. 1997b. Evolving sorting networks using genetic programming and rapidly reconfigurable field-programmable gate arrays. In Higuchi, T., ed., *Workshop on Evolvable Systems. International Joint Conference on Artificial Intelligence*, 27–32.

[ Koza *et al.* 1997c] Koza, J. R., Bennett III, F. H., Lohn, J., Dunlap, F., Keane, M. A. and Andre, D. 1997c. Automated synthesis of computational circuits using genetic programming. In *Proceedings of the 1997 IEEE International Conference on Evolutionary Computation*, 447–452. Indianapolis: IEEE Press.

[ Koza *et al.* 1999] Koza, J. R., III, F. H. B., Andre, D. and Keane, M. A. 1999. *Genetic Programming III: Darwinian Invention and Problem Solving*. San Fransisco: Morgan Kaufmann.

[ Koza *et al.* 2000] Koza, J. R., Keane, M. A., Yu, J. and Mydlowec, W. 2000. Automatic synthesis of electrical circuits containing a free variable using genetic programming. In Whitley, D., Golberg, D., Cantù-Paz, E., Spector, L., Parmee, I. and Beyer, H.-G., eds., *Proceedings of the Genetic and Evolutionary Computation Conference*, 477–484. Las Vegas: Morgan Kaufmann.

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

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

[ Langdon and Poli 1997a] Langdon, W. B. and Poli, R. 1997a. An analysis of the MAX problem in genetic programming. In Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H. and Riolo, R. L., eds., *Genetic Programming 1997: Proceedings of the Second Annual Conference*, 222–230. Stanford University, CA, USA: Morgan Kaufmann.

[ Langdon and Poli 1997b] Langdon, W. B. and Poli, R. 1997b. Fitness causes bloat. In Chawdhry, P. K., Roy, R. and Pant, R. K., eds., *Soft Computing in Engineering Design and Manufacturing*, 13–22. Springer-Verlag London.

[ Langdon and Poli 1997c] Langdon, W. B. and Poli, R. 1997c. Fitness causes bloat: Mutation. In Koza, J., ed., *Late Breaking Papers at the GP-97 Conference*, 132–140. Stanford, CA, USA: Stanford Bookstore.

[ Langdon and Poli 1998] Langdon, W. B. and Poli, R. 1998. Genetic programming bloat with dynamic fitness. In Banzhaf, W., Poli, R., Schoenauer, M. and Fogarty, T. C., eds., *Proceedings of the First European Workshop on Genetic Programming*, Volume 1391 of *LNCS*, 96–112. Paris: Springer-Verlag.

[ Langdon *et al.* 1999] Langdon, W. B., Soule, T., Poli, R. and Foster, J. A. 1999. The evolution of size and shape. In Spector, L., Langdon, W. B., O'Reilly, U.-M. and Angeline, P. J., eds., *Advances in Genetic Programming 3*. Cambridge, MA, USA: MIT Press. 163–190.

[ Langdon 1998] Langdon, W. B. 1998. The evolution of size in variable length representations. In *1998 IEEE International Conference on Evolutionary Computation*, 633–638. Anchorage, Alaska, USA: IEEE Press.

[ Langdon 1999] Langdon, W. B. 1999. Size fair and homologous tree genetic programming crossovers. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M. and Smith, R. E., eds., *Proceedings of the Genetic and Evolutionary Computation Conference*, Volume 2, 1092–1097. Orlando, Florida, USA: Morgan Kaufmann.

[ Langdon 2000] Langdon, W. B. 2000. Quadratic bloat in genetic programming. In Whitley, D., Goldberg, D., Cantú-Paz, E., Spector, L., Parmee, I. and Beyer, H.-G., eds., *GECCO-2000: Proceedings of the Genetic and Evolutionary Computat9ion Conference*, 451–458. San Fransisco: Morgan Kaufmann.

[ Lanzi 1999a] Lanzi, P. L. 1999a. Extending the representation of classifier conditions part i: from binary to messy coding. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M. and Smith, R. E., eds., *Proceedings of the Genetic and Evolutionary Computation Conference*, Volume 1, 337–344. Orlando, Florida, USA: Morgan Kaufmann.

[ Lanzi 1999b] Lanzi, P. L. 1999b. Extending the representation of classifier conditions part ii: from messy coding to s-expressions. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M. and Smith, R. E., eds., *Proceedings of the Genetic and Evolutionary Computation Conference*, Volume 1, 345–352. Orlando, Florida, USA: Morgan Kaufmann.

[ Lindgren *et al.* 1992] Lindgren, K., Nilsson, A., Nordahl, M. G. and Råde, I. 1992. Regular language inference using evolving neural networks. In Schaffer, J. D. and Whitley, D., eds., *Proceedings of the Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN92)*, 55–74. Los alamitos: The IEEE Computer Society Press.

[ Luke and Spector 1996] Luke, S. and Spector, L. 1996. Evolving teamwork and coordination with genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B. and Riolo, R. L., eds., *Genetic Programming 1996: Proceedings of the First Annual Conference*, 150–156. Stanford University, CA, USA: MIT Press.

[ Luke, Hamahashi, and Kitano 1999] Luke, S., Hamahashi, S. and Kitano, H. 1999. "Genetic" programming. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M. and Smith, R. E., eds., *Proceedings of the Genetic and Evolutionary Computation Conference*, Volume 2, 1098–1105. Orlando, Florida, USA: Morgan Kaufmann.

[ Luke 2000] Luke, S. 2000. ECJ: A Java-based evolutionary computation and genetic programming system. Available at http://www.cs.umd.edu/projects/plus/ecj/.

[ Lukschandl *et al.* 2000] Lukschandl, E., Borgvall, H., Nohle, L., Nordahl, M. and Nordin, P. 2000. Distributed java bytecode genetic programming. In Poli, R., Banzhaf, W., Langdon, W. B., Miller, J. F., Nordin, P. and Fogarty, T. C., eds., *Genetic Programming, Proceedings of EuroGP'2000*, Volume 1802 of *LNCS*, 316–325. Edinburgh: Springer-Verlag.

[ Matsumoto and Nishimura 1998] Matsumoto, M. and Nishimura, T. 1998. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation* 8(1):3–30.

[ McPhee and Miller 1995] McPhee, N. F. and Miller, J. D. 1995. Accurate replication in genetic programming. In Eshelman, L., ed., *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, 303–309. Pittsburgh, PA, USA: Morgan Kaufmann.

[ Mitchell 1996] Mitchell, M. 1996. *An Introduction to Genetic Algorithms*. MIT Press.

[ Montana 1995] Montana, D. J. 1995. Strongly typed genetic programming. *Evolutionary Computation* 3(2):199–230.

[ Nachbar 2000] Nachbar, R. B. 2000. Molecular evolution: Automated manipulation of hierarchical chemical topology and its application to average molecular structures. *Genetic Programming And Evolvable Machines* 1(1/2):57–94.

[ Nordin and Banzhaf 1995] Nordin, P. and Banzhaf, W. 1995. Complexity compression and evolution. In Eshelman, L., ed., *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, 310–317. Pittsburgh, PA, USA: Morgan Kaufmann.

[ Nordin, Francone, and Banzhaf 1995] Nordin, P., Francone, F. and Banzhaf, W. 1995. Explicitly defined introns and destructive crossover in genetic programming. In Rosca, J. P., ed., *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, 6–22.

[ Nordin, Francone, and Banzhaf 1996] Nordin, P., Francone, F. and Banzhaf, W. 1996. Explicitly defined introns and destructive crossover in genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., eds., *Advances in Genetic Programming 2*. Cambridge, MA, USA: MIT Press. 111–134.

[ Nordin 1997] Nordin, P. 1997. *Evolutionary Program Induction of Binary Machine Code and its Applications*. Ph.D. Dissertation, der Universitat Dortmund am Fachereich Informatik.

[ Ohno 1970] Ohno, S. 1970. *Evolution by Gene Duplication*. New York: Springer-Verlag.

[ O'Reilly and Oppacher 1995] O'Reilly, U.-M. and Oppacher, F. 1995. The troubling aspects of a building block hypothesis for genetic programming. In Whitley, L. D. and Vose, M. D., eds., *Foundations of Genetic Algorithms 3*, 73–88. Estes Park, Colorado, USA: Morgan Kaufmann.

[ O'Reilly 1995] O'Reilly, U.-M. 1995. *An Analysis of Genetic Programming*. Ph.D. Dissertation, Carleton University, Ottawa-Carleton Institute for Computer Science, Ottawa, Ontario, Canada.

[ Palmer and Delwiche 1996] Palmer, J. D. and Delwiche, C. F. 1996. Second-hand chloroplasts and the case of the disappearing nucleus. *Proceedings of the National Academy of Sciences* 93:7432–7435.

[ Poli and Langdon 1997] Poli, R. and Langdon, W. B. 1997. A new schema theory for genetic programming with one-point crossover and point mutation. In Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H. and Riolo, R. L., eds., *Genetic Programming 1997: Proceedings of the Second Annual Conference*, 278–285. Stanford University, CA, USA: Morgan Kaufmann.

[ Poli, Page, and Langdon 1999] Poli, R., Page, J. and Langdon, W. B. 1999. Smooth uniform crossover, sub-machine code GP and demes: A recipe for solving high-order boolean parity problems. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M. and Smith, R. E., eds., *Proceedings of the Genetic and Evolutionary Computation Conference*, Volume 2, 1162–1169. Orlando, Florida, USA: Morgan Kaufmann.

[ Poli 2000] Poli, R. 2000. Exact schema theorem and effective fitness for gp with one-point crossover. In Whitley, D., Golberg, D., Cantù-Paz, E., Spector, L., Parmee, I. and Beyer, H.-G., eds., *Proceedings of the Genetic and Evolutionary Computation Conference*, 469–476. Las Vegas: Morgan Kaufmann.

[ Price 1970] Price, G. R. 1970. Selection and covariance. *Nature* 227:520–521.

[ Qureshi 1997] Qureshi, A. 1997. The gpsys genetic programming system. Available at http://www.cs.ucl.ac.uk/staff/A.Qureshi/gpsys.html.

[ Radcliffe and George 1993] Radcliffe, N. and George, F. 1993. A study in set recombination. In *Proceedings of the Fifth Annual International Conference of Genetic Algorithms (ICGA-93)*, 22–30.

[ Raik and Durnota 1994] Raik, S. and Durnota, B. 1994. The evolution of sporting strategies. In Stonier, R. J. and Yu, X. H., eds., *Complex Systems: Mechanisms of Adaption*, 85–92. IOS Press.

[ Rechenberg 1973] Rechenberg, I. 1973. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen evolution ("Evolution strategy: the optimization of technical systems according to the principles of biological evolution").* Stuttgart: Frommann–Holzboog Verlag.

[ Rosca and Ballard 1995] Rosca, J. and Ballard, D. H. 1995. Causality in genetic programming. In Eshelman, L., ed., *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, 256–263. Pittsburgh, PA, USA: Morgan Kaufmann.

[ Rosca and Ballard 1996] Rosca, J. P. and Ballard, D. H. 1996. Discovery of subroutines in genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., eds., *Advances in Genetic Programming 2*. Cambridge, MA, USA: MIT Press. 177–202.

[ Rosca 1996] Rosca, J. 1996. Generality versus size in genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B. and Riolo, R. L., eds., *Genetic Programming 1996: Proceedings of the First Annual Conference*, 381–387. Stanford University, CA, USA: MIT Press.

[ Rosca 1997] Rosca, J. P. 1997. Analysis of complexity drift in genetic programming. In Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H. and Riolo, R. L., eds., *Genetic Programming 1997: Proceedings of the Second Annual Conference*, 286–294. Stanford University, CA, USA: Morgan Kaufmann.

[ Schwefel 1977] Schwefel, H.-P. 1977. *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie ("Numeric Optimization of Computer Models by Means of an Evolution Strategy"), Interdisciplinary System Research*, Volume 26. Bassel: Birkhauser.

[ Shaffer and Eshelman 1991] Shaffer, J. D. and Eshelman, L. J. 1991. On crossover as an evolutionarily viable strategy. In Belew, R. K. and Booker, L. B., eds., *Proceedings of the Fourth International Conference on Genetic Algorithms*, 61–68. Morgan Kaufmann.

[ Smith and Harries 1998] Smith, P. W. H. and Harries, K. 1998. Code growth, explicitly defined introns, and alternative selection schemes. *Evolutionary Computation* 6(4):339–360.

[ Smith 1980] Smith, S. F. 1980. *A Learning System Based on Genetic Adaptive Algorithms*. Ph.D. Dissertation, Computer Science Department, University of Pittsburgh.

[ Soule and Foster 1998] Soule, T. and Foster, J. A. 1998. Removal bias: a new cause of code growth in tree based evolutionary programming. In *1998 IEEE International Conference on Evolutionary Computation*, 781–186. Anchorage, Alaska, USA: IEEE Press.

[ Soule, Foster, and Dickinson 1996a] Soule, T., Foster, J. A. and Dickinson, J. 1996a. Code growth in genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B. and Riolo, R. L., eds., *Genetic Programming 1996: Proceedings of the First Annual Conference*, 215–223. Stanford University, CA, USA: MIT Press.

[ Soule, Foster, and Dickinson 1996b] Soule, T., Foster, J. A. and Dickinson, J. 1996b. Using genetic programming to approximate maximum clique. In Koza, J. R., Goldberg, D. E., Fogel, D. B. and Riolo, R. L., eds., *Genetic Programming 1996: Proceedings of the First Annual Conference*, 400–405. Stanford University, CA, USA: MIT Press.

[ Spears 1993] Spears, W. M. 1993. Crossover or mutation? In Whitley, L. D., ed., *Foundations of Genetic Algorithms 2*. Morgan Kaufmann.

[ Spector and Luke 1996] Spector, L. and Luke, S. 1996. Cultural transmission of information in genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B. and Riolo, R. L., eds., *Genetic Programming 1996: Proceedings of the First Annual Conference*, 209–214. Stanford University, CA, USA: MIT Press.

[ Spector *et al.* 1999] Spector, L., Barnum, H., Bernstein, H. J. and Swamy, N. 1999. Quantum computing applications of genetic programming. In Spector, L., Langdon, W. B., O'Reilly, U.-M. and Angeline, P. J., eds., *Advances in Genetic Programming 3*. Cambridge, MA, USA: MIT Press. 135–160.

[ Spector 1996] Spector, L. 1996. Simultaneous evolution of programs and their control structures. In Angeline, P. J. and Kinnear, Jr., K. E., eds., *Advances in Genetic Programming 2*. Cambridge, MA, USA: MIT Press. 137–154.

[ Stoffel and Spector 1996] Stoffel, K. and Spector, L. 1996. High-performance, parallel, stack-based genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B. and Riolo, R. L., eds., *Genetic Programming 1996: Proceedings of the First Annual Conference*, 224–229. Stanford University, CA, USA: MIT Press.

[ Strachan 1992] Strachan, T. 1992. *The Human Genome*. Oxford: BIOS.

[ Tackett 1994] Tackett, W. A. 1994. *Recombination, Selection, and the Genetic Construction of Computer Programs*. Ph.D. Dissertation, University of Southern California, Department of Electrical Engineering Systems.

[ Tate and Smith 1993] Tate, D. M. and Smith, A. E. 1993. Expected allele coverage and the role of mutation in genetic algorithms. In Forrest, S., ed., *Proceedings of the Fifth International Conference on Genetic Algorithms*, 31–37. Morgan Kaufmann.

[ Teller 1994a] Teller, A. 1994a. The evolution of mental models. In Kinnear, Jr., K. E., ed., *Advances in Genetic Programming*. MIT Press. 199–219.

[ Teller 1994b] Teller, A. 1994b. Turing completeness in the language of genetic programming with indexed memory. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, Volume 1, 136–141. Orlando, Florida, USA: IEEE Press.

[ Teller 1996] Teller, A. 1996. Evolving programmers: The co-evolution of intelligent recombination operators. In Angeline, P. J. and Kinnear, Jr., K. E., eds., *Advances in Genetic Programming 2*. Cambridge, MA, USA: MIT Press. 45–68.

[ Teller 1998] Teller, A. 1998. *Algorithm Evolution with Internal Reinforcement for Signal Understanding*. Ph.D. Dissertation, School of Computer Science, Carnegie Mellon University.

[ Thompson 1968] Thompson, K. 1968. Regular expression search algorithm. *Communications of the ACM* 11(6):419–422.

[ Tomita 1982] Tomita, M. 1982. Dynamic construction of finite automata from examples using hill-climbing. In *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*, 105–108.

[ Whigham 1996] Whigham, P. A. 1996. Search bias, language bias, and genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B. and Riolo, R. L., eds., *Genetic Programming 1996: Proceedings of the First Annual Conference*, 230–237. Stanford University, CA, USA: MIT Press.

[ Whitley 1989] Whitley, D. 1989. The GENITOR algorithm and selection pressure: why ranked-based allocation of reproductive trials is best. In Schaffer, J. D., ed., *Proceedings of the Third International Conference on Genetic Algorithms*, 239–255. Morgan Kaufmann.

[ Wilson and Macleod 1993] Wilson, P. B. and Macleod, M. D. 1993. Low implementation cost IIR digital filter design using genetic algorithms. In *IEE/IEEE Workshop on Natural Algorithms in Signal Processing*, Volume 4, 1–8.

[ Wright 1964] Wright, S. 1964. Stochastic processes in evolution. In Gurland, J., ed., *Stochastic Models in Medicine and Biology*, 199–241. University of Wisconsin Press.

[ Wu and Lindsay 1995] Wu, A. and Lindsay, R. 1995. Empirical studies of the genetic algorithm with non-coding segments. *Evolutionary Computation* 3(2).

[ Zomorodian 1994] Zomorodian, A. 1994. Context-free language induction by evolution of deterministic push-down automata using genetic programming. In Koza, J. R., ed., *Genetic Algorithms at Stanford 1994*. Stanford, California, 94305-3079 USA: Stanford Bookstore. 184–193.

[ Zongker and Punch 1996] Zongker, D. and Punch, B. 1996. lilgp 1.01 user's manual. Technical report, Michigan State University, USA.