EVOLVING MODULAR PROGRAMS BY EXTRACTING REUSABLE FUNCTIONS USING SIGNIFICANCE TESTING

by

ANTHONY OREN LOEPPERT

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2005

ACKNOWLEDGEMENTS

I'd like to thank my parents and wife for encouraging and supporting me to start and complete my thesis. My advisor, Dr. Huber, of course, provided much patience and advice during this project.

December 9, 2005

ABSTRACT

EVOLVING MODULAR PROGRAMS BY EXTRACTING REUSABLE FUNCTIONS USING SIGNIFICANCE TESTING

Publication No. ______

ANTHONY OREN LOEPPERT, M.S.C.S.

The University of Texas at Arlington, 2005

Supervising Professor: Manfred Huber

Genetic programming is an automatic programming method that uses biologically inspired methods to evolve programs. Genetic programming, and evolutionary methods in general, are useful for problem domains in which a method for *constructing* solutions is either not known or infeasible, but a method for *rating* solutions exists. In order to address more complex problem domains, techniques exist to extract functions (modules) automatically during a GP search. This work describes a method to identify useful automatically extracted functions from a GP search to assist subsequent GP searches within the same problem domain, using significance testing. Functions classified as beneficial augment the programmer supplied function set and accelerate the learning rate, by seeding the initial population of a subsequent GP search.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii

ABSTRACT iv

LIST OF FIGURES viii

LIST OF TABLES x

CHAPTER

1. OVERVIEW AND MOTIVATION 1

    1.1 Motivation for Automatic Programming 1

    1.2 What is Automatic Programming? 1

    1.3 It worked for Nature 2

    1.4 Evolving Programs 3

    1.5 Document Overview 3

2. INTRODUCTION TO GENETIC ALGORITHMS 5

    2.1 What is a genetic algorithm? 5

    2.2 Representation 5

    2.3 Operators 6

    2.4 Fitness Function 8

    2.5 Selection 9

    2.6 Putting all the pieces together 10

3. OVERVIEW OF GENETIC PROGRAMMING 12

    3.1 Primitive Functions 12

    3.2 Operators 13

    3.3 Scalability 13

3.3.1 Automatically Defined Functions 15

3.3.2 Module Acquisition 16

3.4 Purpose 17

4. RELATED WORK 19

4.1 Building Good Functions 19

4.2 Identifying Good Functions 20

4.2.1 Separate Evaluation 20

4.2.2 Evolutionarily Viable 21

4.2.3 Fitness of Individuals Containing the Module 21

5. METHODOLOGY AND TOOLS 22

5.1 GP System 22

5.2 Function Encapsulation 23

5.3 Function Judgment/Identification Mechanism 25

5.4 Function Dissolution and the Mutation Operator 27

5.5 Function Sharing Across Problems 28

5.6 Breeding Policy 29

5.6.1 Standard Policy 29

5.6.2 Extended Policy 29

6. PROBLEM DOMAIN 32

6.1 GP Primitives 32

6.1.1 Types 32

6.1.2 Terminals 33

6.1.3 Non-Terminals 34

6.1.4 Sample Program 34

6.2 Fitness Function 35

7. RESULTS 37

7.1 Function Identification Methods 39
7.1.1 SigTest x 39
7.1.2 EvoVia 40
7.1.3 Rand 41
7.1.4 Best 41
7.1.5 NoFunc 41
7.2 Standard Breeding Policy 41
7.3 Extended Breeding Policy 46
7.4 Standard vs. Extended 48
8. CONCLUSION 51
8.1 Observations 51
8.2 Applicable Domains 51
8.3 Future Work 53
REFERENCES 54
BIOGRAPHICAL STATEMENT 58

LIST OF FIGURES

Figure Page

2.1 A simple GA individual 6

2.2 Example GA 1 point crossover 7

2.3 Example GA 2 point crossover 8

2.4 Example GA mutation 9

2.5 A Simple GA 11

3.1 A simple GP algorithm. 13

3.2 Example GP 1 point crossover 14

3.3 An example of a GP mutation. 15

3.4 A GP Individual with two ADFs 16

3.5 Example function extraction 17

5.1 A simplification of the abstract class that all GP function nodes must subclass 23

5.2 Example function extraction 24

5.3 Example GP 1 point crossover 30

6.1 Board configurations before and after a play at the center light. 32

6.2 Illustration of coordinate system and location constants. 33

6.3 A sample Lights Off program 35

6.4 Evaluation Method for **Move** 36

7.1 Overview of the GP system with function extraction 38

7.2 Example confidence intervals 40

7.3 Overview of function identification methods 42

7.4 Standard breeding policy results for levels 1,2 and 3 44

7.5 More standard breeding policy results for levels 2 and 3 45

7.6 Summary of function id. strategies on level 3 using standard breeding policy 46

7.7 Extended breeding policy results for level 3 47

7.8 A comparison of various standard and extended policy results 50

8.1 Two very different board configurations with same fitness 52

LIST OF TABLES

Table Page

5.1 Extraction Algorithm 25

5.2 Function Rating Algorithm 28

5.3 Extended Breeding Policy 31

6.1 Types 33

6.2 Terminals 34

6.3 Non-Terminals 35

7.1 Random Run 37

7.2 Average number of functions promoted using standard breeding 43

7.3 Average number of functions promoted using extended breeding 47

CHAPTER 1

OVERVIEW AND MOTIVATION

1.1 Motivation for Automatic Programming

"Due to the rise in complexity demanded and the fact that technological advances in hardware of the last decade have far outstripped those in software technology, the software component of a computing system currently accounts for more than 70% of the total cost; and this figure is on the rise. Meanwhile it is projected that the trend in demand for ever more, and ever more complex, systems will not (and cannot) be matched with a corresponding increase in the number of competent personnel needed to produce them using traditional technology (and its expected extensions). The implication is that, unless significant break-throughs are made in the area of software technology, we may expect a serious crisis in the next decade."[1]

Though the work quoted above was published in 1977, the statement (specifics regarding the estimated percentage of software cost relative to a total product cost aside) rings true today. In fact, it's probably more relevant now, given the proliferation of embedded microprocessors in our environment and our increasing reliance on technology, as compared with almost 30 years ago.

1.2 What is Automatic Programming?

Gray[2] describes automatic programming, a "holy grail"[1] of computer science, as an interface that allows easy program description which is both compilable and can

[1]Pessimistic readers might interpret this as "fool's errand".

describe all possible programs. Gray indicates a good measure of “easy” as an effort several orders of magnitude less than what is currently needed. Using Gray’s definition, one accustomed to programming in machine code could rightly call a high-level language compiler automatic programming. However, by automatic programming, we mean something more than straightforward abstractions that have occurred in programming: flipping physical switches to machine code to assembly to the variety of high-level languages.

Attempting to rigorously define automatic programming is approximately the same task as defining artificial intelligence, but the word “automatic” implies a *collaboration* between a computer system and human program to produce programs to produce a design specific enough for conventional compilers, assemblers, etc. to take over and produce a program. Merely speeding up the process of programming should not be the only property of an automatic programming system. It should also enable the human programmer to produce products he or she could not have produced with conventional tools. Ultimately, the definition is a slippery one, but we have some idea of what an automatic programming system should be.

1.3 It worked for Nature

While there are many different methods of automatic programming, genetic programming looks toward nature for inspiration. Genetic programming applies simplified models of biological evolution to program construction. The motivation is rather simple: biological evolution has produced novel problem solvers, so perhaps we can successfully employ the process in a digital environment.

1.4 Evolving Programs

Evolving programs is challenging because the "solution" is the product of a non-deterministic simulation governed by the complex interactions of the genetic and selection operators guided by the supplied fitness function. The evolution process may produce unexpected and/or disappointing results due to subtle interactions of all the pieces composing a GA that cannot be anticipated prior to initiating the search process.

Another challenge for evolutionary programming is producing solutions that generalize. Ideally, running solution program $Prog$, evolved against problem set A, on problem set B may result in less than stellar performance, even if A and B are in the same problem domain. Evolved solutions are often specific to the environment they evolved from. This problem, however, is not specific to evolutionary programming and is often called over fitting.

Even if end solutions within a problem domain do not generalize, it could be advantageous to use some of the output, in the form of program fragments, of previous evolutionary searches to aid subsequent searches within the same problem domain. This, ideally, would speed up the evolutionary process since the search need not start from a completely random state, instead leveraging previous related searches. Identifying these beneficial program fragments is the focus of this thesis.

1.5 Document Overview

Chapter 2 introduces genetic algorithms and explains some of the necessary inputs to a GA and Chapter 3 describes the application of GA to automatic programming. Then some related work regarding beneficial function identification is discussed in Chapter 4. Chapter 5 presents the method we propose to identify program fragments that will benefit subsequent evolutionary searches. Chapters 6 describes the problem domain used in experimentation and Chapter 7 shows the results. Chapter 8 offers some closing remarks and suggestions for future work.

CHAPTER 2

INTRODUCTION TO GENETIC ALGORITHMS

2.1 What is a genetic algorithm?

A genetic algorithm is a heuristic search mechanism that applies the principles of biological evolution, such as survival of the fittest, towards problem solving. A genetic algorithm maintains a set of possible solutions (individuals) to the given problem called a population, where the initial population is typically random. A genetic algorithm search is broken into stages called generations, where applying genetic operators to members of the current population produce a new population. Individuals are ranked according to a fitness value produced by running a user supplied fitness function on each individual in the population. Individuals are sampled from the population, usually according to some probability proportional to its fitness, to reproduce and create the next generation. The GA typically runs until an ideal individual is found or a maximum generation is reached.[3]

2.2 Representation

When approaching a new problem with a genetic algorithm, a decision regarding candidate solution representation is necessary. This is simply a description of what a solution will look like. GA representations can be binary strings or made up of a more complex alphabet and have a fixed length. These are often referred to as chromosomes. Each chromosome contains a number of genes coding different traits or values. Figure 2.1 is an example representation using the binary alphabet, where there are two chromosomes for each individual with three genes on each chromosome. The decision could have been to have one chromosome with six genes instead and depending on the crossover routine, this may or may not subtly impact the GA search. As stated before, there are many choices of representation encoding [4] and most GA implementations have fixed length chromosomes, though there is interest in resizing chromosome length during runtime [5]. For the GA examples in this chapter, we will use a simple binary encoding with fixed length chromosomes.

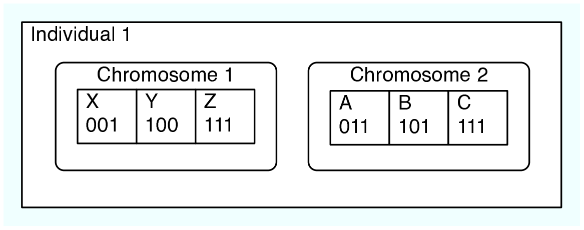


Figure 2.1. A simple layout of an individual with two chromosomes, each containing three genes.

2.3 Operators

A genetic algorithm applies crossover and mutation operators to individuals in a population to produce new individuals. Figures 2.2 and 2.3 show two simple crossover operators; one and two point crossovers.

Crossover operator choice is typically governed by the particular problem domain, or more precisely the particular representation for the specific problem, and is a dominant factor in guiding a GA search. Depending on the specifics of the representation, simple one or two point crossovers may slow down evolution by splicing two individuals together in a way that sabotages the search. The traveling salesman problem (TSP) is a common benchmark test for GA crossover methods and Katayama[6] explores some of the performance numbers for a variety crossover operators applied to the TSP. While

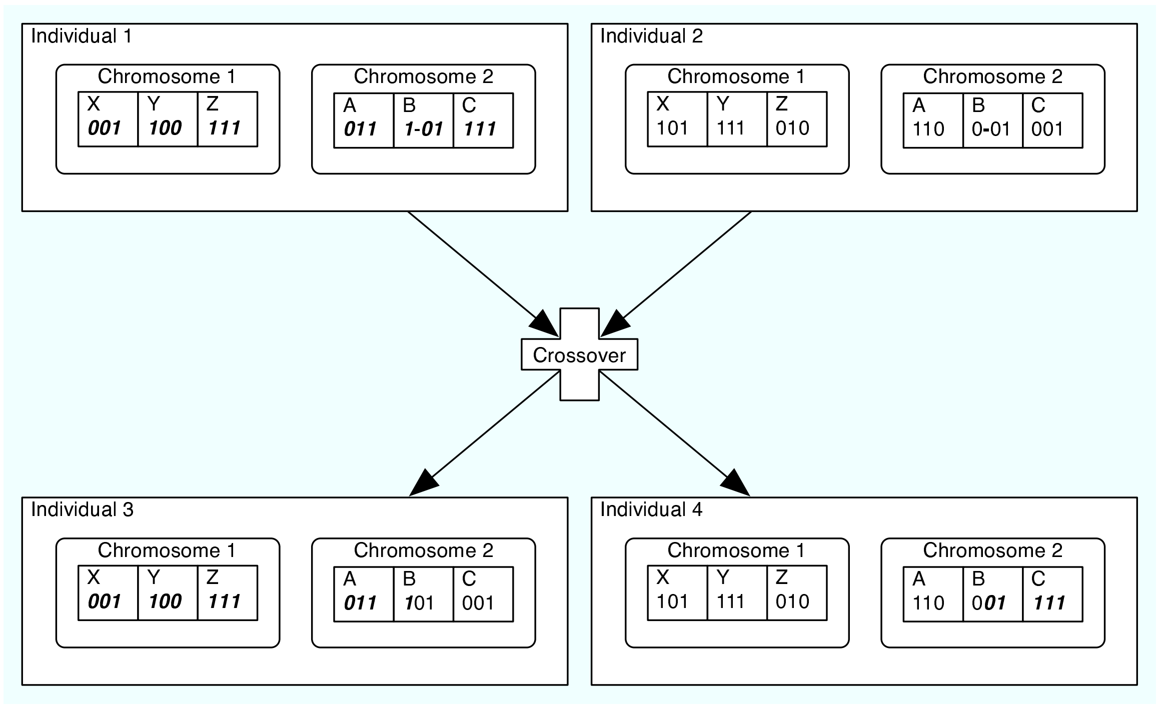


Figure 2.2. This is an example of a single point crossover of two individuals producing two offspring. The dash in the B gene of the parents marks the crossover point.

such comparisons can be useful guides, there isn't a universal crossover that will perform well in all circumstances, and many of the crossover methods are specifically tailored to graph theory. Similarly, there isn't a universal encoding scheme that will work optimally for all domains.

A mutation operation (see Figure 2.4), applied with some, usually small, probability, is also integral to a GA by allowing the search to explore search space outside the crossover-only space and also helps keep the population diverse. Novel structure discovered randomly, that benefits the search by increasing the fitness of individuals containing the mutation, will propagate through the population. Disadvantageous mutations will disappear as the GA search progresses since it will lower the fitness of individuals that contain it, thus lowering their chance of reproducing.

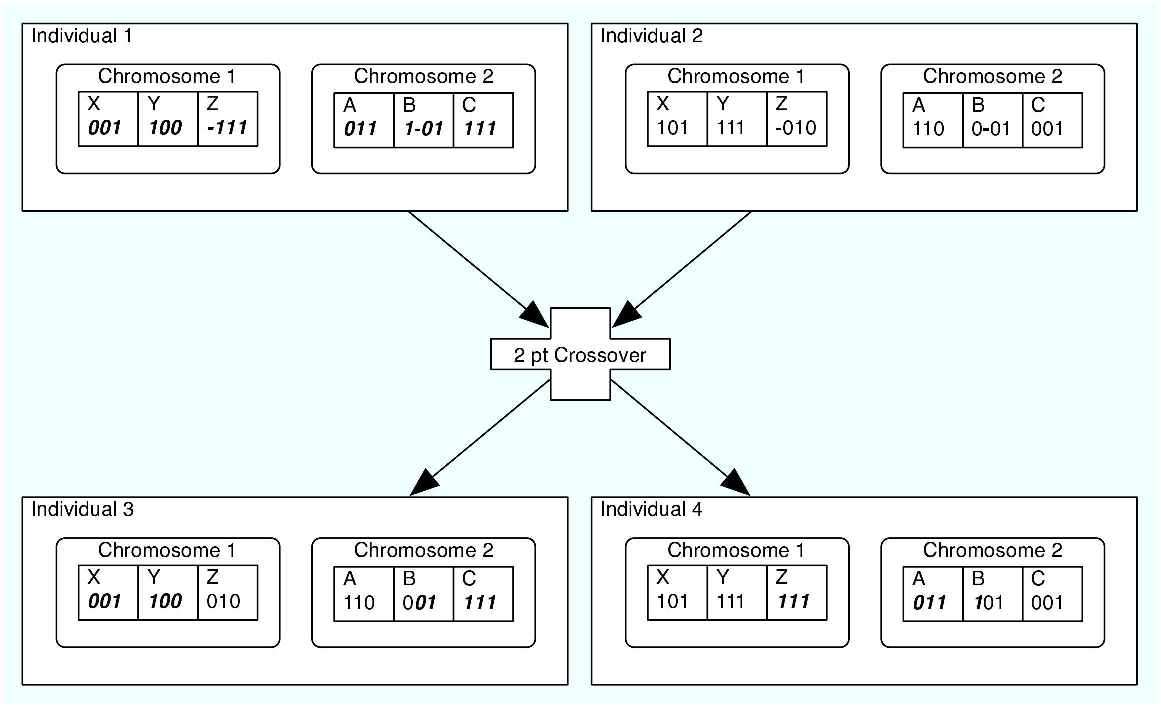


Figure 2.3. This is an example of a two point crossover of two individuals producing two offspring. The dash in the Z and B genes of the parents mark the crossover points.

The crossover and mutation operators have values p_c and p_m associated with them, respectively that represent the probabilities that an operation will be applied. Choosing effective values for p_c and p_m is likely a matter of trial and error in preliminary experimentations, but typically $p_c \gg p_m$, where $0.75 \leq p_c \leq 1.0$ and $0.0 < p_m \leq 0.1$.

2.4 Fitness Function

Genetic algorithms require a programmer supplied scoring mechanism for rating and ranking an individual's performance within the population. This scoring mechanism is specific to the problem at hand and guides the GA search. It is the heuristic that the probabilistic operations are based on.

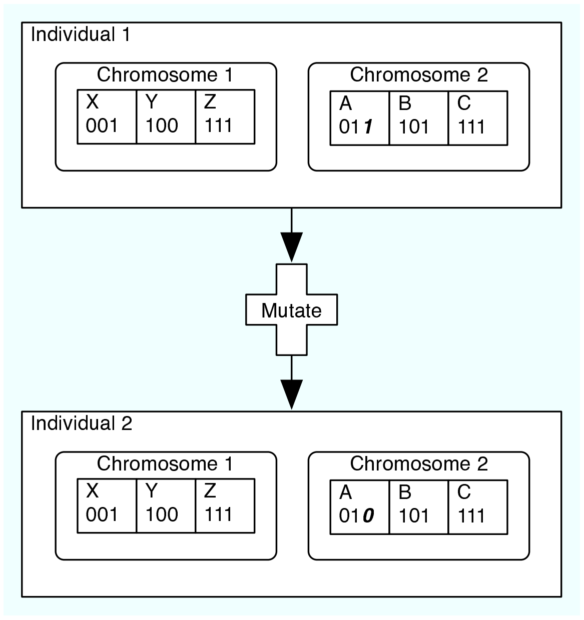


Figure 2.4. The bold italicized bit in gene A from the top individual is randomly chosen and flipped to create a new individual.

2.5 Selection

Choosing which individuals reproduce from the current generation's population is the job of the selection mechanism. A selection method choice affects a GA search in the following way, illustrated by this extreme example. Imagine two otherwise identical GA searches, one utilizing some fitness proportionate[1] selection method and the other utilizing uniform selection[2]. The fitness proportionate selection method will bias subsequent populations to resemble the best individuals, and the average fitness of the population and the fitness of the best individual will likely converge. This represents a loss of genetic diversity and the search stagnates, which is fine if an ideal individual is found, but otherwise undesirable. Conversely, the search using uniform selection would likely maintain a large diversity but improve little from the initial random state in terms of the best individual's fitness and the population's average fitness, which is also undesirable. So

[1]Individuals are selected for reproduction in proportion to their fitness.

[2]Individuals are sampled uniformly from the population for reproduction, ignoring fitness.

a selection method should, when applied to the given problem, should balance selective pressure with diversity preservation. The liberal application of the mutation operator can also reintroduce diversity.

Bäck[7] discusses the concept of *takeover times*, which is an attempt to quantify the selective pressure of a selection method. Takeover times are essentially an estimation in the number of generations it would take for an initial population to become entirely made up of copies of an ideal individual placed in the first population. The smaller a takeover time the more aggressive selection pressure generated by a given selection method.

Some common selection methods include the roulette wheel (a fitness proportionate selection scheme) and linear rank selection. Roulette wheel assigns a probability in proportion to an individual's fitness value, so the larger the raw fitness value, the higher the probability an individual will be chosen. Linear rank selection sorts the population according to fitness but the probability an individual will be sampled from the population is not directly related to its raw fitness value, but to a probability value mapped to its sorted position. A fitness proportionate selection method would allow a best individual that is relatively far from the next best's fitness to dominate the search, the rank selection tempers premature convergence by not allowing a small number of highly fit individuals from dominating the reproduction cycle[8]. Thus it is surprising that, in [7], Bäck experimentally found that linear ranking had a higher selective pressure than proportional selection, as linear ranking is designed to lower selection pressure, at least when the fitness variance is high.

2.6 Putting all the pieces together

The first step in applying a genetic algorithm to a problem starts with choosing the candidate solution representation as well as designing a fitness function that will rate candidate solutions. Once the representation is chosen the genetic operators must be designed, taking care that the crossover routine does not destroy representation syntax when combining parents to form offspring. The GA randomly initializes by creating a random population then begins the cycle of applying the genetic operators to produce new individuals from existing ones. Figure 2.5 illustrates a simple GA.

```
$Pop \leftarrow randomIndividuals(popSize)$ {Initialize the GA search}
$setFitness(Pop)$ {Run the fitness function on each individual}
**while** $Pop_{best}$ not ideal and $gen < maxGen$ **do**
  $gen \leftarrow gen + 1$
  $NextPop \leftarrow \{\}$
  $numOffspring \leftarrow 0$
  **while** $numOffspring < popSize$ **do**
    $parent_1 \leftarrow selection(Pop)$
    $parent_2 \leftarrow selection(Pop)$
    **if** $randFrom0To1() <= p_c$ **then**
      $offspring \leftarrow crossover(parent_1, parent_2)$
    **else**
      $offspring_1 \leftarrow parent_1$
      $offspring_2 \leftarrow parent_2$
    **end if**
    **if** $randFrom0To1() <= p_m$ **then**
      $offspring_1 \leftarrow mutate(offspring_1)$
    **end if**
    **if** $randFrom0To1() <= p_m$ **then**
      $offspring_2 \leftarrow mutate(offspring_2)$
    **end if**
    $NextPop \leftarrow NextPop \cup offspring$
    $numOffspring \leftarrow numOffspring + 2$
  **end while**
  $Pop \leftarrow NextPop$
  $setFitness(Pop)$
**end while**
```

Figure 2.5. A Simple GA.

CHAPTER 3

OVERVIEW OF GENETIC PROGRAMMING

Genetic programming (GP)[9] is a type of genetic algorithm. It's the application of GA to the field of automatic programming. Figure 3.1 shows the flow of a simple GP algorithm. The main differentiating feature of genetic programming versus a standard genetic algorithm is the solution representation[10]. In GP, the individuals must be executed, literally, as the individuals are *programs*. In a standard GA, individuals are typically *parameters* to a program. GP representation often a tree structure corresponding to a parse tree, rather than a bit string.

3.1 Primitive Functions

GP needs a set of terminal (no parameters) and non-terminal (with parameters) functions with which to construct individuals. These are programmer-supplied functions that will bound the limits of the GP search. Individuals will never be able to perform functionality beyond the capabilities of the primitive function set, so including many different primitive functions is beneficial. On the other hand, including functions not necessary to solve the problem at hand can retard a GP search since the search space of possible programs becomes larger. In practice, it is something of an art, or much trial and error, to choose proper function sets for efficient GP searches of particular problem domains.

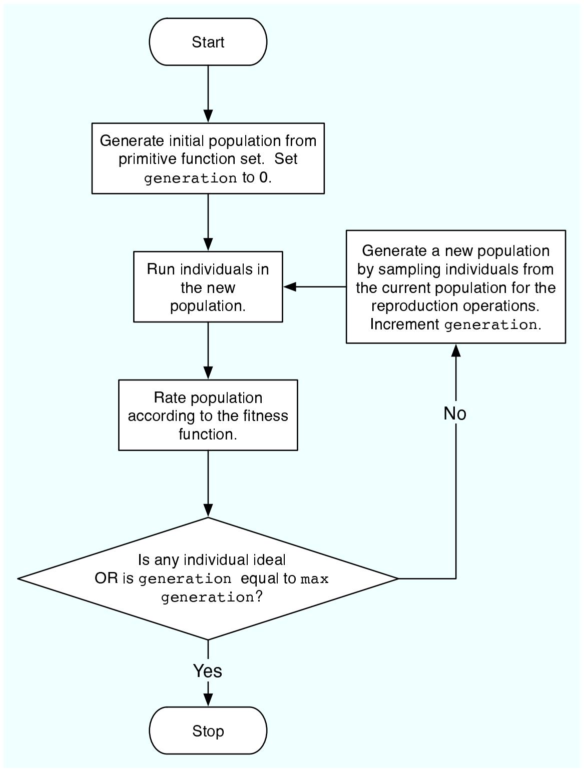

Start
Generate initial population from primitive function set. Set generation to 0.
Run individuals in the new population.
Rate population according to the fitness function.
Is any individual ideal OR is generation equal to max generation?
Yes
No
Stop
Generate a new population by sampling individuals from the current population for the reproduction operations. Increment generation.

Figure 3.1. A simple GP algorithm..

3.2 Operators

Genetic operators must correspond to the encoding choice made, thus crossover and mutation for GP are tree operations (unless a more elaborate GP representation is chosen) rather than string operations for GA. Figures 3.2 and 3.3 show examples of crossover and mutation operations on individuals of the Lights Off problem domain (see section 6).

3.3 Scalability

Standard GP doesn't provide any facilities for encapsulating programs structures. The hand written equivalent would be to write a large program that consists only of a main function. Human programmers quickly discover the usefulness of creating and

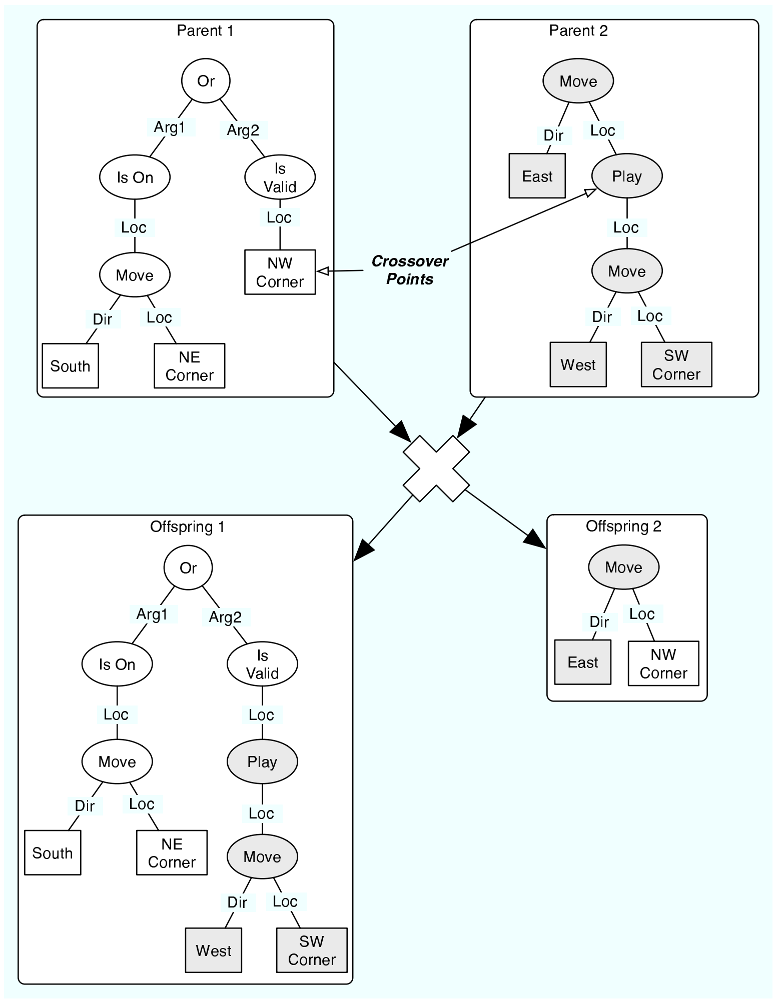


Figure 3.2. An example of a simple one point crossover between two parents resulting in two offspring..

calling functions if the goal is writing a program of non-trivial complexity. If GP is to be able to address complex problems, it too needs a method to create modules using the primitive function set. There are two common methods for evolving hierarchical programs.

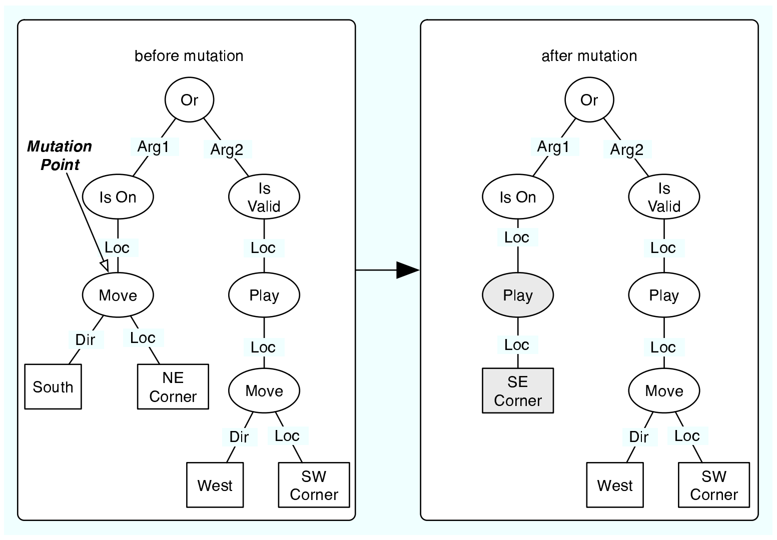


Figure 3.3. An example of a GP mutation..

3.3.1 Automatically Defined Functions

Evolving modular programs seems to be a key ingredient to successfully tackling problems of high complexity with evolutionary programming techniques. Automatically Defined Functions (ADFs) [11][12] is one method of evolving modular solutions to problems. ADFs are functions that can be called from other ADFs or the main function of an individual. The number of ADFs, their prototypes, and sets of functions visible to each ADF are decided prior to the GP search by the programmer though the function body of each ADF is evolved. Each individual within the population has the same ADFs visible to it, though each can have different bodies for each ADF. Figure 3.4 shows an example of an individual with two ADFs. This method allows knowledge to be supplied by the programmer via constraints on the structure individuals, thus constraining the GP search space. A potential draw back is the program must be partially designed by a programmer. Presumably the programmer doesn't know how to *implement* the solution to the problem since he is using an evolutionary method to *discover* the solution. Poorly chosen ADFs may inadvertently introduce constraints that impede the GP search rather than aid it.

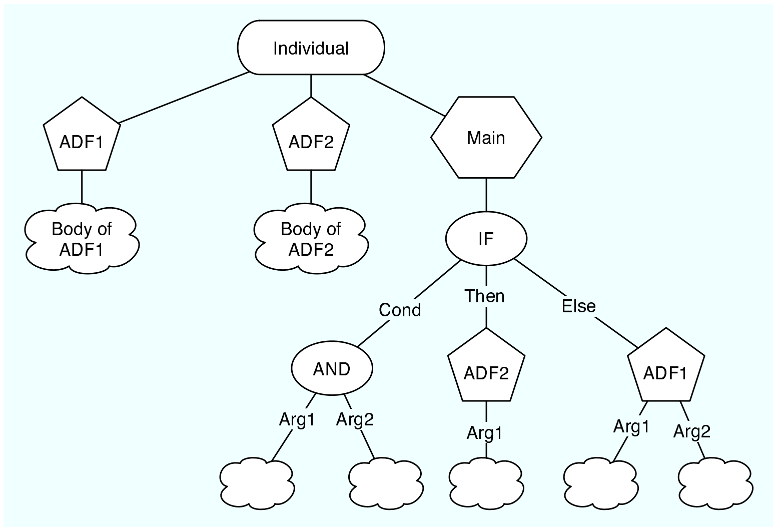


Figure 3.4. This shows an individual consisting of two ADFs and the main program. The function sets and argument lists for each ADF are declared globally, but the body of an ADF is local to an individual, and evolves along with the main program..

3.3.2 Module Acquisition

Another method for evolving modular solutions, called GLiB[13] or Module Acquisition[1], simply randomly extracts functions from the population. A random point in an individual is selected as the root of the new module and all program structure below this point, up to some maximum depth, is encapsulated[2] within the module. If there are branches beyond the maximum depth of the module, they become parameters to the module. This has the advantage that no knowledge is needed regarding end program structure, as is implicitly required by ADFs. This can be especially useful if the pro-

[1]This is also referred to as encapsulation.

[2]This is also referred to as a compression operation.

grammer his no intuitive sense of how to construct the ADF constraints for the problem domain at hand. See Figure 3.5 for an example.

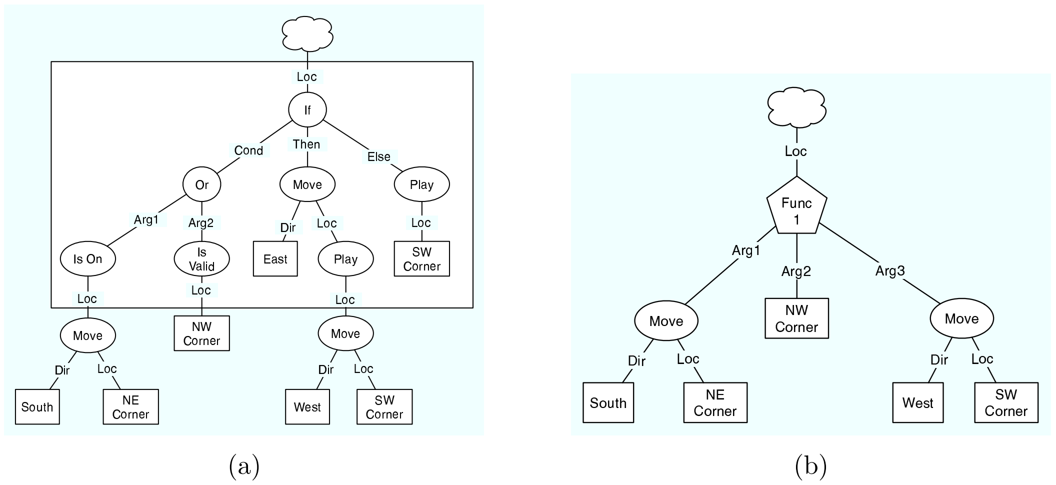


Figure 3.5. (a) shows a program fragment before a function extraction. (b) shows the result of a function extraction rooted at the If statement with a maximum depth of 3.

3.4 Purpose

Section 3.3 introduces the importance of modules for evolving hierarchical programs. The next logical step in benefiting from modularity is reuse of modules on new problems or tasks, thus capitalizing on previous computational effort to speed the current GP search. To discover which modules should be extracted from a GP search to be reused, one must rate them to sort out the most beneficial ones. Unfortunately, assigning a fitness value to a *piece* of an individual is not a trivial task. This thesis describes a method that identifies randomly selected modules[3], created using the GLiB method, that are good candidates for reuse by estimating their worth using a statistical significance

[3]Modules are also referred to as functions. The terms are used interchangeably in this thesis.

test. Functions identified as worthy for sharing are then added to the primitive function set for a subsequent GP search on a different problem within the same domain.

CHAPTER 4

RELATED WORK

Here we show different methods to assign fitness measures to functions either evolved in (ADFs) or extracted from (Module Acquisition) a GP search. The various works have different implementations details and problem domains, but we focus on how functions are built and subsequently identified as beneficial.

4.1 Building Good Functions

Creating modules from subtrees that occur frequently with a population seems like a good step. After all, the idea of functions is to use them in multiple locations. If the body of the function occurs at multiple locations, replacing the body structure with the name of the function seems like a good simplification of the program. [14] compared performance between selecting modules based on frequency of subtree occurrence and random subtree selection and showed that random selection outperformed the frequency selection strategy. This result is echoed in [15] in which a variety of module selection heuristics were tried. The [15] found that selecting modules randomly resulted in high performance consistently across problem domains. It should be noted that in this context, "selection" refers to *how* the body of modules are created, not identifying which functions, that have already been created, are useful. It also should be pointed out that selecting random subtrees from individuals in the population is different from creating random subtrees from a set of functions. The dynamics of a GP search have no effect on the later.

We end the discussion of selection heuristics here as the literature suggests that it is difficult to do better than random selection, both in terms of overhead and performance, without sacrificing problem domain independence. Random selection is use in all the experiments conducted within this work.

4.2 Identifying Good Functions

The previous section discussed ways of *creating* functions that benefit a GP search. In this section, we look at ways of identifying *individual* beneficial functions, *after* they have been created. While the tasks are similar, the previous section focused on producing functions that, in aggregate, increase performance by increasing the learning rate or overall fitness or both. This section looks at ways to single out exceptional functions from the set of created functions. This is important since, as stated in Section 3.1, introducing non-beneficial functions into the primitive functions set will negatively impact a GP search since each function expands the search space.

4.2.1 Separate Evaluation

In [16], a module is evaluated as if it were an individual in the population using the standard fitness function. Thus, functions that solve more of the overall problem are better functions. This is desirable because it requires no extra effort from the programmer regarding GP setup. There is a potential pitfall to rating functions using the fitness functions for individuals. Such a rating scheme disadvantages modules that are context sensitive (i.e. they make little or no sense outside the context in which they are called) and make little or no progress towards the overall solution as measured by the fitness function but, for example, serve as a useful condition branch in an **if** statement. It is possible the rating system would rank a large class of otherwise beneficial functions as poor performers.

4.2.2 Evolutionarily Viable

The frequency of a module within the population can be used to indicate its usefulness. [13] calls a function with sufficient[1] occurrences *evolutionarily viable*, meaning it proliferated throughout the population. This suggests that it positively contributes to the solution.

4.2.3 Fitness of Individuals Containing the Module

[17] experimented with adding ADFs from the best individual from a GP search to the function set of a subsequent GP search. Applying this methodology sped up subsequent GP searches, measured in terms of the number of individuals evaluated, within the even n-parity function[2] problem domain. Using ADFs from 2-parity solutions in a GP search for the even 6-parity problem decreased the needed number of evaluations from 1,500,000 to 700,000. Using ADFs from the more complicated 3-parity problem decreased needed evaluations from 1,500,000 to 400,000.

[18] is similar to [17] in that it extracts functions from the fittest individuals but it performs this operation during a GP search every time a new best individual is found. Functions are rated by summing the fitnesses of each individual containing it and dividing it by the total number of calls to the function.

[1]What indicates sufficiency is subjective.

[2]The even n-parity problem attempts to construct a function that will output true if an even number of n inputs are true.

CHAPTER 5

METHODOLOGY AND TOOLS

5.1 GP System

The focus of this thesis is to explore a domain independent method for identifying exceptionally performing functions from all functions extracted in order to add them to the primitive function set and speed subsequent GP searches on similar problems in the same domain. We chose the GLiB method of function extraction to build our method upon as it does not have the extra hierarchy requirement of the programmer.

The GP environment used to perform the experiments is a custom Java program loosely modeled on [19]. Each individual in the population has its own tree structure of function nodes that represent the individual's program structure, where node children represent arguments to a particular node. Each function node has a return type[20] as well as a description of its required arguments and an evaluate method that will, as needed, evaluate the node's children to perform the specific task of the particular node. If a function node needs information not provided by its children or by its own instance variables, references to the problem environment it is executing in and the individual it resides in are provided as parameters. Figure 5.1 shows the basic skeleton of every function node in an individual's program tree.

Encapsulating a subtree within an individual and replacing it with a stub has no affect on the fitness of an individual. An individual, pre and post encapsulation performs identically since the post encapsulation individual simply has a stub representing the subtree.

```
public abstract class GPFunction{
    public int          retType;
    public int[]        childTypes;
    public GPFunction[] children;
    public abstract Object eval(GPProblem problem,
                                GPIndividual individual);
}
```

Figure 5.1. A simplification of the abstract class that all GP function nodes must subclass.

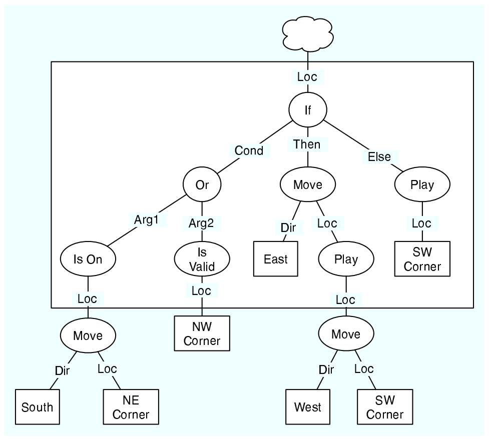
The main effect of encapsulation protects the function's body from genetic operators and in turn reduces the number of crossover points within an individual. In Section 5.2, we describe the function extraction[1] operation. The extraction process randomly selects a subtree from a copy of the original and replaces it with a newly created function. The descendants of both the original and the copy (which contains the new function) are tracked, up to a specified maximum number of generations. While a function is being tracked, the difference between the average fitnesses of the original's living descendants and the copy's living descendants is recorded each generation and a statistical significance test is applied to these differences. Functions passing the test criteria (i.e. descendants of the copy that contained the new function are on average, with $x\%$ certainty, performing y better than the descendants of the original individual) are promoted to the primitive function set and shared with subsequent GP searches.

5.2 Function Encapsulation

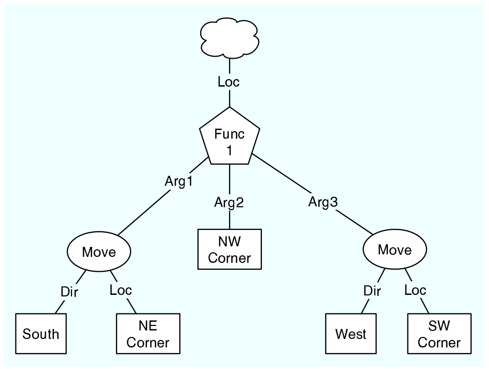
As stated in Section 4.1, functions are selected randomly from individuals in the population. Individuals chosen for function extraction are uniformly sampled from the population, rather than in proportion to fitness. This thesis uses the same extraction procedure as [13] in which functions were created from an individual's genotype by picking

[1]The term encapsulation will also be used.

a random location and encapsulating, up to a maximum depth, nodes beneath it. Any nodes beyond the maximum depth became arguments to the function and thus defined the new function's prototype. Before each function extraction attempt a random maximum depth is chosen uniformly between 2 and 10. See Figure 5.2 (a reproduction of Figure 3.5) for an example. Functions were further constrained to have at least one parameter.



(a)



(b)

Figure 5.2. (a) shows a program fragment before a function extraction. (b) shows the result of a function extraction rooted at the If statement with a maximum depth of 3.

When an individual is selected for a random function extraction, a copy is made of that individual and the function extraction is performed on the copy. The original is referred to as *unmodified* and the copy of the original with the function stub substituted for the function body is called *modified*. A reference to both the unmodified and modified individuals is kept in the new function for use in judging the usefulness of the function. Table 5.1 contains a more detailed description of the process.

Table 5.1. Extraction Algorithm

1. Set `UNMODIFIED_IND` to a sample from the population.
2. Create `MODIFIED_IND` by copying `UNMODIFIED_IND`.
3. Create `EXTRACTED_FUNC` by extracting a random function from `MODIFIED_IND`.
4. Replace structure in `MODIFIED_IND` used to create `EXTRACTED_FUNC` with a stub pointing to `EXTRACTED_FUNC`.
5. Set `EXTRACTED_FUNC.UNMODIFIED` reference to `UNMODIFIED_IND`.
6. Set `EXTRACTED_FUNC.MODIFIED` reference to `MODIFIED_IND`.
7. Add `EXTRACTED_FUNC` to the extracted function repository.
8. Add both `UNMODIFIED_IND` and `MODIFIED_IND` to the next generation's population.

5.3 Function Judgment/Identification Mechanism

Given the increased search space overhead for each addition to the function set, we can't simply share all functions extracted. If we expect to benefit from sharing extracted functions, a method is needed to identify the beneficial functions from the total set of extracted functions. Here we propose a scheme for estimating *how much* better or worse off the GP search is after extracting a function using significance testing (SigTest).

For each function under consideration, the fitnesses of the descendants of the unmodified and modified individuals in the current population are compared each generation until a worthiness decision on the function is made. This comparison each generation is called a sample. The sample for the current generation is the average fitness of the descendants of the unmodified individual minus the average fitness of the descendants of the modified individual. Equation 5.1 shows how a sample is calculated for function id. fit_{unmod}^{id} and fit_{mod}^{id} are lists of fitness values for individuals that are living descendants (i.e. contained within the current population) of the id.`UNMODIFIED` and id.`MODIFIED` individuals, respectively. *Length* returns the number of elements in a list, and the bracket operation indexes into a given list. g is the number of generations after id was created, thus X_g^{id} is the performance sample for function id for the gth generation after id was created.

$$\begin{aligned} X_g^{id} &= \frac{\sum_{j=1}^{m} fit_{unmod}^{id}[j]}{m} - \frac{\sum_{k=1}^{n} fit_{mod}^{id}[k]}{n} \\ m &= \text{Length}(fit_{unmod}^{id}) \\ n &= \text{Length}(fit_{mod}^{id}) \end{aligned} \tag{5.1}$$

The reason we look at the difference in performance of the descendants of the unmodified (without) and modified (with) for each extracted function is because we are attempting to single out exceptional functions for sharing. We are inferring the difference in performance of the unmodified and modified individuals should be credited to the extracted function since they are identical otherwise. Thus, if the performance difference favors the modified individual's descendants, we assign the reason for the performance difference to the function. While this is an unproven assumption, it seems more straightforward to compare performance differentials of initially identical individuals, save the function extraction, to tease out the worth of a function rather than assigning worth to functions based on the fitness of individuals containing said function, or simply because a function occurs frequently.

One sample is not much information judge a function with, so multiple samples are taken. After two samples are obtained[2], a t-test is performed to analyze the samples obtained in Equation 5.1. Functions that *pass* the t-distribution significance test in Equation (5.2), where X is the function's sample array and $\alpha = 0.95$, are deemed *good* and are added to the primitive function set which allows them to be introduced into

[2]There is one sample per generation for each function currently under consideration by the judgement mechanism.

the population via mutation as well as propagated by reproduction. Functions that *fail* Equation (5.2), with $\alpha = 0.6$, are deemed unworthy and are dissolved[3].

$$\overline{X^{id}}(n) - t_{n-1,\alpha}\sqrt{\frac{S^2(n)}{n}} \geq DecencyThreshold \tag{5.2}$$

where $\overline{X^{id}}(n)$ is the mean of the performance differences between the function and non-function individuals over the last n generations, and $S^2(n)$ is the corresponding variance.

$$\overline{X^{id}}(n) = \frac{\sum_{g=1}^{n} X_g^{id}}{n} \tag{5.3}$$

$$S^2(n) = \frac{\sum_{g=1}^{n} [X_g^{id} - \overline{X^{id}}(n)]^2}{n-1} \tag{5.4}$$

It should be noted that the assumed fitness value range is from zero to infinity, where zero is the best fitness value. The values for $t_{n-1,\alpha}$ and Equations (5.3) and (5.4) can be found in [21]. Equation 5.2 simply states that if the inequality holds true, the confidence interval of the mean of all the samples taken for function *id* is better than the specified value, *DecencyThreshold*. The larger *DecencyThreshold* is, the higher the expectations of performance for promotion are. Table 5.2 shows the judgement algorithm in more detail.

5.4 Function Dissolution and the Mutation Operator

Functions are dissolved randomly (i.e. replacing the function stub with the actual subtree representing its body) via the mutation operator. If a site selected at random within an individual is an encapsulated function, it will be dissolved. If not, the mutation operation works normally (i.e. replacing the selected site with a randomly constructed subtree). Because functions insulate the program structure that they represent from

[3]The function stub is replaced with the associated body in individuals containing the function. This is the reverse of the extraction process.

Table 5.2. Function Rating Algorithm

1. Set `UNMOD_DESCENDANTS` to the list of descendants of `FUNC.UNMODIFIED` within the current population. This value was set during the function's creation.
2. Set `MOD_DESCENDANTS` to the list of descendants of `FUNC.MODIFIED` within the current population.
3. Remove individuals from `MOD_DESCENDANTS` that either don't contain `FUNC` or didn't execute `FUNC`.
4. Sort both `UNMOD_DESCENDANTS` and `MOD_DESCENDANTS` according to fitness, from best to worst.
5. Set `AVG_UNMOD` and `AVG_MOD` to the average fitness of the individuals in `UNMOD_DESCENDANTS` and `MOD_DESCENDANTS` respectively.
6. Set `SAMPLE` to `AVG_UNMOD` minus `AVG_MOD`. Samples greater than zero indicate `AVG_MOD` is performing better than `AVG_UNMOD` since lower values are better, according to the fitness value range assumptions.
7. Add `SAMPLE` to `FUNC.SAMPLES`.
8. Calculate the `SAMPLE_MEAN` and t-distribution confidence intervals `CONF_INT_95%` and `CONF_INT_60%` of `FUNC.SAMPLES`.
9. If `SAMPLE_MEAN` minus `CONF_INT_95%` *is* greater than or equal to *DecencyThreshold* then `FUNC` is deemed *good* and the rating algorithm is done, otherwise proceed to the next step. *DecencyThreshold* is a runtime parameter.
10. If `SAMPLE_MEAN` minus `CONF_INT_60%` *is not* greater than or equal to *DecencyThreshold* then `FUNC` is deemed *bad*, otherwise no decision is made, and more samples are needed.

the standard genetic operators, the encapsulated subtree forming the function is frozen. While this is desirable (it is, after all, the point of function encapsulation), allowing too much of the genetic code to be frozen can drive the GP search into a local optimum[13][22]. Random function dissolution is a counter balance to this effect.

5.5 Function Sharing Across Problems

To measure how well the rating system selects functions that generalize to other problems, *good* functions from previous GP searches are added to the primitive function set before the initial random population is constructed for a subsequent GP search on a different problem set.

5.6 Breeding Policy

Experimentation with the breeding policy was performed to attempt to speed learning by placing constraints on offspring that increase diversity between successive generations. Enforcing diversity among the population mitigates the search getting caught in local optima. Diversity metrics found in [23], [24] and [25] focus on analysis of the genotype (i.e. the structure of the individual). In contrast, this research considers only the results of individuals. If two individuals in a population produce different results for a given problem, it is inferred that their structure is different, though no metric of *how much* different is estimated.

In the case of the Lights Off problem domain (see section 6), the result of an individual is the end configuration of the game board after its program tree has been executed. This end configuration is mapped to an integer (each light represents a binary digit of an integer) that represents the result.

5.6.1 Standard Policy

The standard breeding policy does not consider diversity at all. Two offspring are generated by each breeding procedure. Two parents are sampled probabilistically, according to fitness, from the current population. A single point crossover (see Figure 5.3, a copy of Figure 3.2, for an example) is performed to form two offspring individuals. Then a mutation is possibly performed on each offspring, determined by the mutation rate. The offspring are then executed on the problem set and enter the next generation's population.

5.6.2 Extended Policy

Offspring are generated and executed in the same manner as the standard policy. After execution, the GP system analyzes the results of an offspring and if it does not

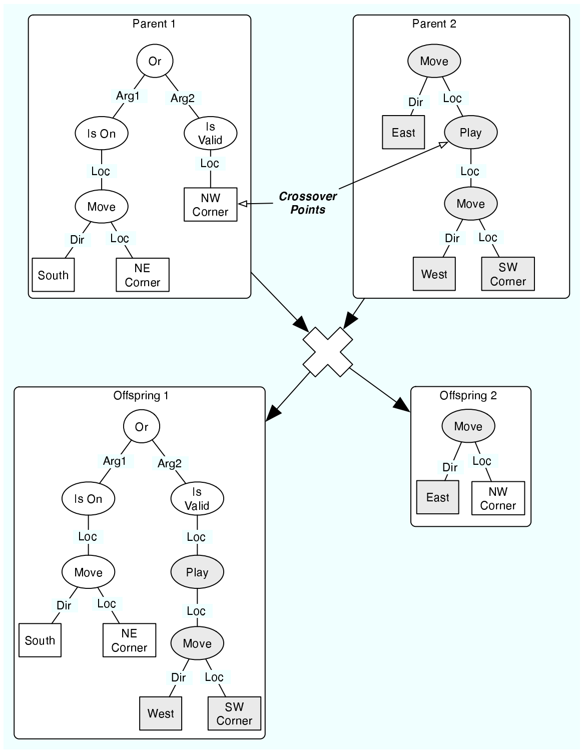


Figure 5.3. An example of a simple one point crossover between two parents resulting in two offspring..

have sufficient result diversity with respect to the current population it is discarded. Sufficient result diversity is defined as a minimum number of different results. If, for instance, there are 2 results produced by each individual, and a diversity requirement of 1 is imposed, then any offspring not producing at least one new result, with respect to the current population's result set, will not enter the next generation's population and is thrown away. For more detail, see Table 5.3.

Table 5.3. Extended Breeding Policy

1. Select `PARENT1` and `PARENT2` from the population.
2. Generate `OFFSPRING` from `PARENT1` and `PARENT2`.
3. Execute `OFFSPRING` on the problem set.
4. Set `NUM_UNIQUE_RESULTS` to 0.
5. For each `PROBLEM` / `RESULT` pair in `OFFSPRING`:
    - If `RESULT` is not in the current generation's result set for `PROBLEM`, then increment `NUM_UNIQUE_RESULTS`.
6. If `NUM_UNIQUE_RESULTS` is less than `DIVERSITY_REQUIREMENT` then throw `OFFSPRING` away. Otherwise, add `OFFSPRING` to the next generation's population and add a reference to `OFFSPRING` to `PARENT1.OFFSPRING_SET` and `PARENT2.OFFSPRING_SET`.

CHAPTER 6

PROBLEM DOMAIN

The problem domain chosen for experimentation is the game known as "Lights Off" or "Lights Out". The board consists of a 5 x 5 matrix of "lights" that have binary states, either on or off. Pressing a light will toggle not only that light, but also the light's four cardinal neighbors. On corners and edges, there is no wrap around. See figure 6.1 for an example play. The goal of the game is to extinguish all the lights.

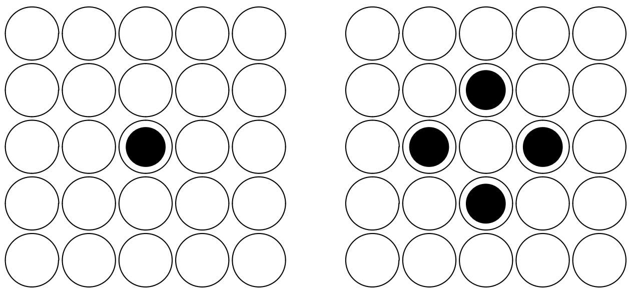


Figure 6.1. Board configurations before and after a play at the center light..

6.1 GP Primitives

6.1.1 Types

Three types have been defined for Lights Off. Table 6.1 shows each type and gives a short description of its purpose. The primitive types correspond to legal argument and return types. Any structure altering operation must respect these constraints to ensure post-operation program structure is legal. Every function within the GP system contains a data structure indicating what its return type is, as well as the types of any parameters it takes.

Table 6.1. Types

type	*description*
boolean	An object representing the binary values true or false.
direction	An ordered pair object representing cardinal directions. These values are added to **location**s to move from one **location** to another. For example, **North** is (0, -1) and **South** is (0, 1).
location	An ordered pair object representing a light on the board matrix. The coordinate system used to specify lights on the board has the origin at the top left button and positive y values growing down the page. Increasing x values grow from left to right on the page. This coordinate system will be used unless otherwise indicated.

6.1.2 Terminals

Terminals (see Table 6.2) are any functions that do not take any parameters, and typically refer to constant values, such as a particular board **location** or **direction**. Figure 6.2 displays the corner board **location**s.

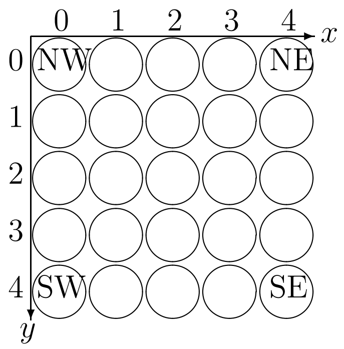


Figure 6.2. Illustration of coordinate system and location constants..

Table 6.2. Terminals

return type	*names*
boolean	**False**
boolean	**True**
direction	**East**
direction	**North**
direction	**South**
direction	**West**
location	**CornerNE**
location	**CornerNW**
location	**CornerSE**
location	**CornerSW**

6.1.3 Non-Terminals

Table 6.3 shows the data types and primitive functions used for this domain. **And** and **Or** short circuit, as in C and many other programming languages. For example, if the first argument of an **And** function node evaluates to **False** then the second argument is not evaluated. Depending on the result of the condition branch of an **If**, either the then-branch or the else-branch is evaluated and returned. **IsOn** returns **True** if its argument branch returns a board location that is in the on state. **IsValid** returns **True** if its argument branch returns a meaningful board location. **Move** returns a **location** displaced by a **direction**. **PlayAt** toggles a neighborhood on the board centered at **location** and then returns that same **location**.

6.1.4 Sample Program

Programs within the GP system are represented by tree structures. Parameters to functions are contained within an array of references to children nodes. A program such as **PlayAt(Move(If(IsOn(CornerNE), North, South), Move(North, CornerSE)))** would have the structure shown in Figure 6.3. The effect of the program is to make a

Table 6.3. Non-Terminals

return type	*name, argument types*
boolean	**And(boolean Arg1, boolean Arg2)**
boolean	**Not(boolean Arg1)**
boolean	**Or(boolean Arg1, boolean Arg2)**
boolean	**IsOn(location Loc)**
boolean	**IsValid(location Loc)**
location	**Move(direction Dir, location Loc)**
location	**PlayAt(location Loc)**
boolean	**If(boolean Cond, boolean Then, boolean Else)**
direction	**If(boolean Cond, direction Then, direction Else)**
location	**If(boolean Cond, location Then, location Else)**

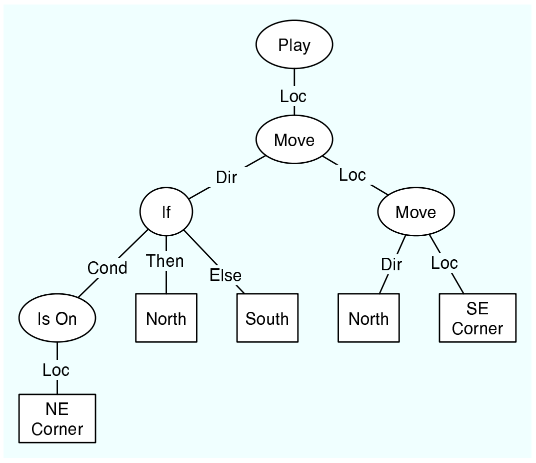


Figure 6.3. A sample Lights Off program.

play at either (4,2) or southeast corner (4,4), depending on whether the northeast corner of the board is on or not. Figure 6.4 lists the function evaluation method for **Move** .

6.2 Fitness Function

The fitness of an individual is the total number of lights still in the on state after the individual has been executed. Thus, zero is the ideal fitness, indicating the initial board configuration has been solved. The GP search will terminate before the maximum

```
public Object eval(GPProblem problem,
                   GPIndividual individual)
{
    Point dir, loc;

    dir = (Point)children[DIR].eval(problem,
                                    individual);
    loc = (Point)children[LOC].eval(problem,
                                    individual);
    loc.translate(dir.x, dir.y);
    return loc;
}
```

Figure 6.4. Evaluation Method for **Move**.

number of generations has been reached if an ideal solution to the problem configuration set is found, otherwise it will continue running until the specified maximum generation.

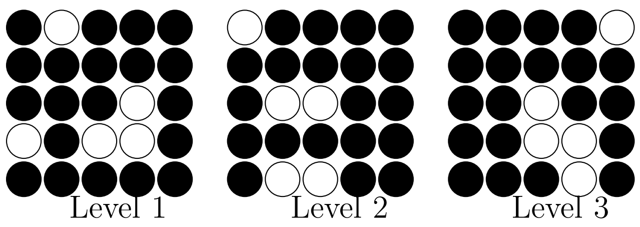
CHAPTER 7

RESULTS

A series of experiments were performed to compare the rates of learning without function extraction and with our GP function identification and sharing approach. A level is a single starting light configuration for the Lights Off puzzle. Successively presenting the GP system with 3 levels (all separate GP searches) is considered a run. Functions were extracted during levels 1 and 2 but not 3. Level 3 had access to the original primitive function set (see Tables 6.2 and 6.3) as well as functions promoted in levels 1 and 2. Level 2 had access to the original function set and functions promoted during the level 1 GP search. See Figure 7.1 for an overview of the GP system augmented with function extraction.

For each function identification scheme tested, it is applied to 100 runs of 3 random levels, each with a 20 of 25 lights on. No two random levels are identical. Each function identification scheme tested is presented with the same random levels. Table 7.1 is an example of one random run.

Table 7.1. Random Run



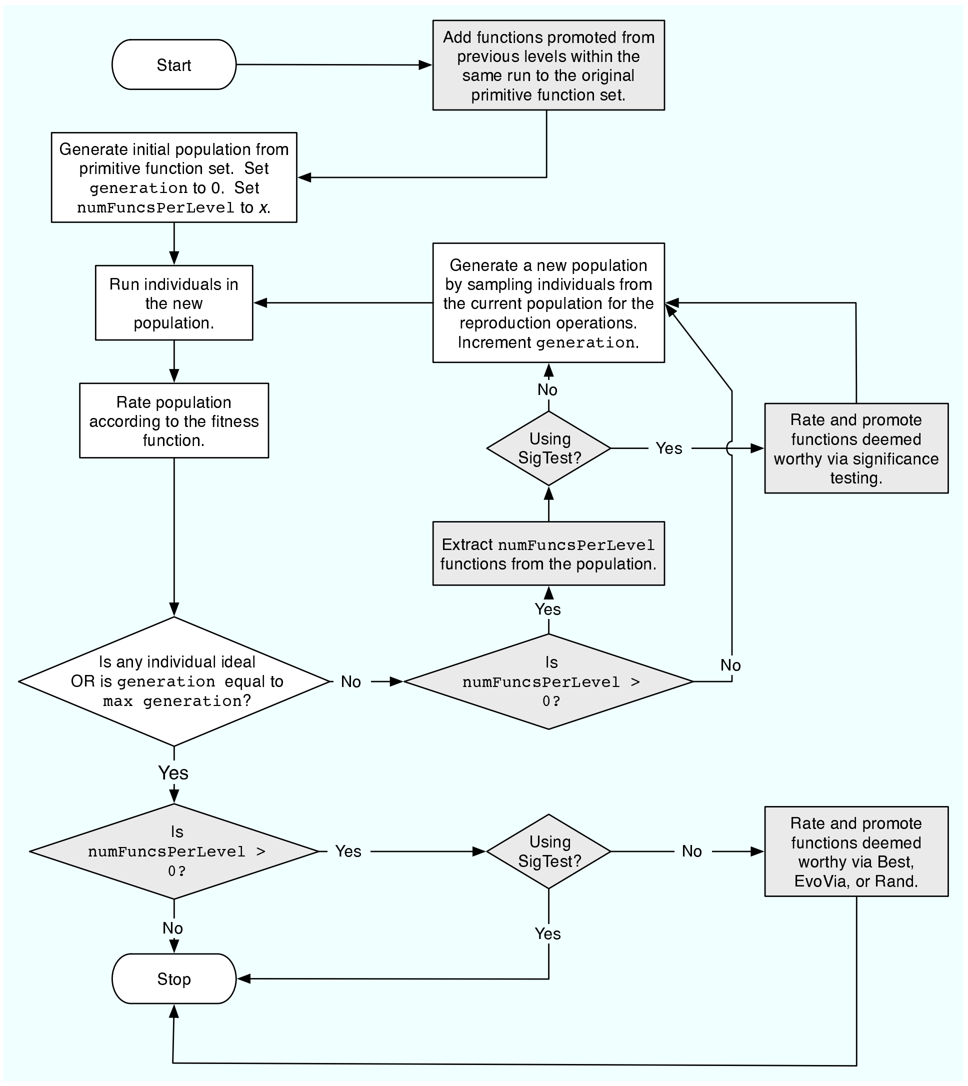

Start
Add functions promoted from previous levels within the same run to the original primitive function set.
Generate initial population from primitive function set. Set generation to 0. Set numFuncsPerLevel to x.
Run individuals in the new population.
Generate a new population by sampling individuals from the current population for the reproduction operations. Increment generation.
Rate population according to the fitness function.
No
Using SigTest?
Yes
Rate and promote functions deemed worthy via significance testing.
Extract numFuncsPerLevel functions from the population.
Yes
Is any individual ideal OR is generation equal to max generation?
No
Is numFuncsPerLevel > 0?
No
Yes
Is numFuncsPerLevel > 0?
Yes
Using SigTest?
No
Rate and promote functions deemed worthy via Best, EvoVia, or Rand.
No
Yes
Stop

Figure 7.1. Overview of the GP system with function extraction. Elements shaded gray are steps added to the original GP system in Figure 3.1.

In the cases where the extended breeding policy was used (see Section 5.6.2), `DIVERSITY_REQUIREMENT` was 1. In all cases (except for the control experiment with no functions at all), the function extraction routine attempted to create 32 functions per generation. The result plots represent averages for each level, across all runs[1], with 95%

[1]Occasionally, the GP system would run out of memory on a level within a run. In this case, the entire run was thrown away, thus the data may only reflect 98 or 99 runs.

t-distribution confidence intervals shown every 15 generations. The top 5% of the population is automatically carried over (copied) to the next generation and the population size is 512. Population sampling is linearly proportional to the individual's fitness value (roulette wheel) . The mutation rate was fixed at 20%.

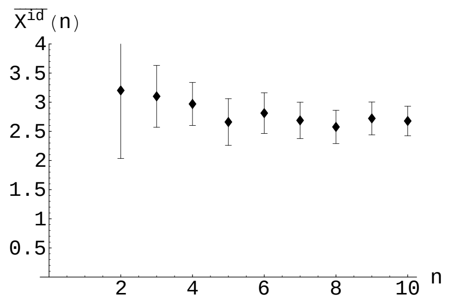
7.1 Function Identification Methods

Three methods for identifying exceptionally performing functions were compared against the significance testing method (SigTest x). The following is a summary of each method. The EvoVia, Best, and Rand methods all identify and promote functions at the end a GP search. SigTest x rates and promotes each generation. In all cases, once functions are committed to the primitive function set, they are not removed until the run (i.e. all 3 levels) is over. Also, for all methods, adding functions to the primitive function set behaves like standard set operations, so there are no duplicates within the function set (i.e. multiple promotions of the same function has no effect).

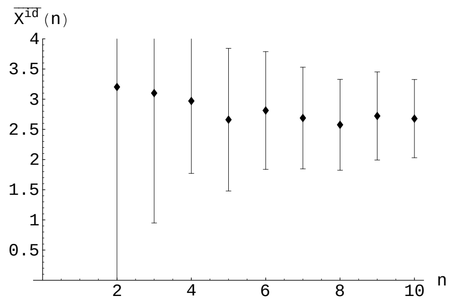
7.1.1 SigTest x

SigTest x is the algorithm described in Section 5.3, where x is the value chosen for *DecencyThreshold*. As *DecencyThreshold* increases, a higher differential performance between the average fitness of the descendants with (modified) and without (unmodified) a particular function. For example, if the series of numbers {2.35424, 4.05059, 2.89817, 2.57818, 1.42205, 3.57517, 1.93977, 1.78932, 3.88968, 2.28426}[2] represents the difference of the average descendant fitness of the unmodified and modified subsequent generations after the function *id* was created (see Equation 5.1), then the 60% and 95% confidence intervals look like Figure 7.2. Continuing with the example, consider

[2]Note that each generation after the creation of *id* one sample value is added to the list, thus this is the samples list for *id* at $n = 10$.



(a)



(b)

Figure 7.2. (a)60% and (b)95% confidence interval plots for the sample data in Section 7.1.1.

$DecencyThreshold = 2.0$ (SigTest 2). A function's rating begins two generations after its creation since statistical tests require at least two samples. Since at $n = 2$, the lower half of the 95% confidence interval is not above 2.0, *id* is not deemed worthy of promotion at this point. The lower part of the 60% confidence interval is (just barely) above 2.0 so *id* is allowed to continue. This same rating outcome for *id* occurs each generation after $n = 2$ until $n = 10$. During this generation, the 95% confidence interval of the mean of the sample differences creeps above 2.0 and at this point, the decision to promote *id* is made and rating *id* stops as well. If the lower end of the 60% error bar had gone below 2.0, then *id* would have been deemed unhelpful (to the degree specified by $DecencyThreshold$) and dissolved in all the individuals containing it.

7.1.2 EvoVia

EvoVia promotes the functions by looking at the last population of a GP search and creating a list, $Func_{freq}$, of extracted functions sorted from most frequently to least frequently occurring. Then, the top 1% of the functions in $Func_{freq}$ are promoted. Functions promoted in previous levels that are designated for promotion in the current level have no effect, though count towards the 1% requirement. See Figure 7.3(a).

7.1.3 Rand

Rand (see Figure 7.3(b)) is similar to EvoVia in that the number of functions to promote is calculated as 1% of the total number of created functions. Rand chooses the functions to promote at random, sampling uniformly from the entire set of extracted functions, regardless of whether functions are contained within the last population or not.

7.1.4 Best

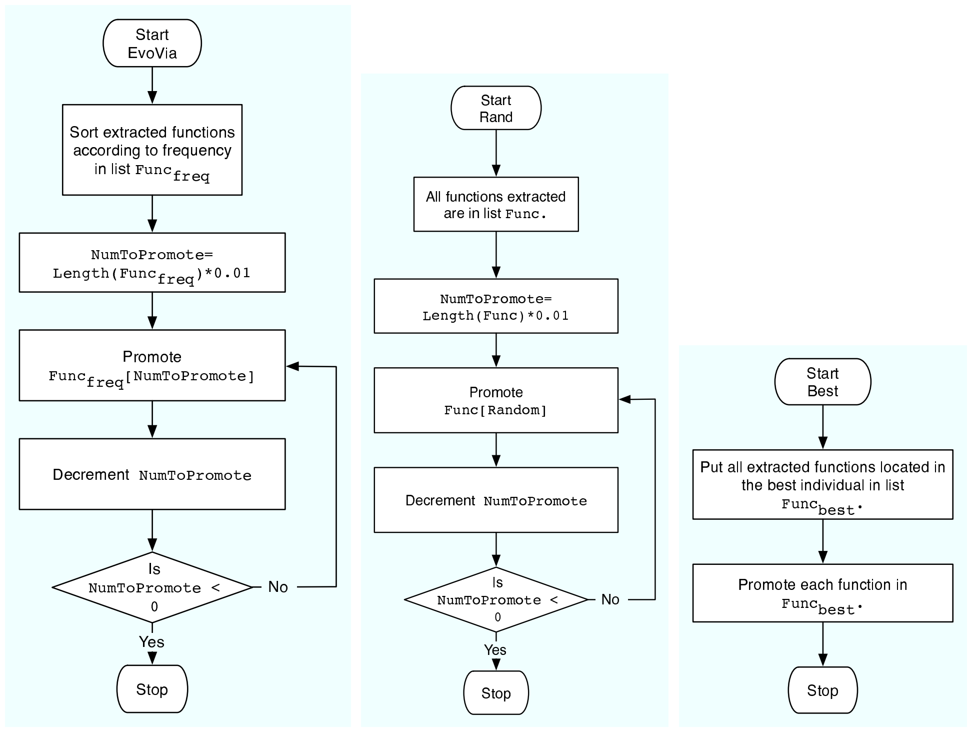
Best is the simplest function identification scheme tested. All the extracted functions residing in the best individual from the last generation's population are promoted to the primitive function set.

7.1.5 NoFunc

NoFunc is the control experiment that doesn't extract, and thus doesn't share, any functions on any level of a run. This facilitates comparison of the different functions extraction/identification methods with standard GP.

7.2 Standard Breeding Policy

Figure 7.4(a) shows the mean best performance plotted against the mean number of individuals evaluated each generation for NoFunc, EvoVia, Rand, and SigTest 5. All strategies perform statistically the same. This shows that the Lights Off domain resists improvement by extracting functions within a GP search, since none of the function strategies performed significantly better than NoFunc. While only EvoVia, Rand, and SigTest 5 strategies are shown against NoFunc, Best performed virtually identically in level 1. This is an indicator that the Lights Off domain is not particularly conducive to random module extraction, as enabling module extraction doesn't increase the learning


Start EvoVia
Sort extracted functions according to frequency in list Func_freq
NumToPromote= Length(Func_freq)*0.01
Promote Func_freq[NumToPromote]
Decrement NumToPromote
Is NumToPromote < 0
No
Yes
Stop
Start Rand
All functions extracted are in list Func.
NumToPromote= Length(Func)*0.01
Promote Func[Random]
Decrement NumToPromote
Is NumToPromote < 0
No
Yes
Stop
Start Best
Put all extracted functions located in the best individual in list Func_best.
Promote each function in Func_best.
Stop

(a) (b) (c)

Figure 7.3. Overview of various function identification methods: (a)EvoVia, (b)Rand, and (c)Best.

rate. Analyzing levels 2 & 3 will give insight on how domain can benefit from seeding the initial population with program fragments evolved from earlier GP searches.

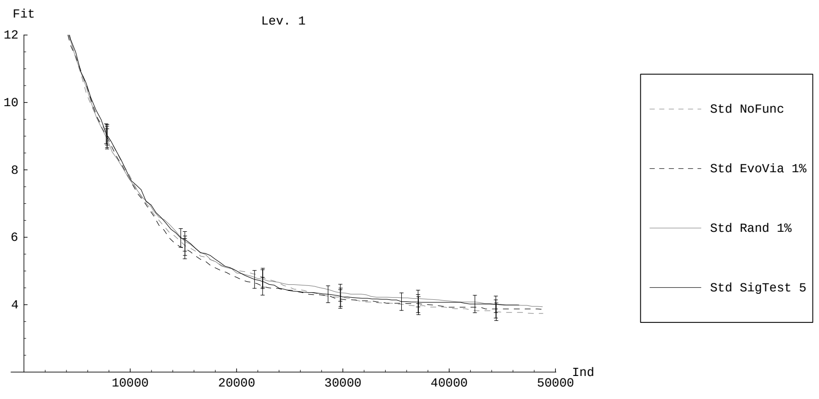
Level 2 (see Figure 7.4(b)) shows the effect of sharing functions promoted from level 1. EvoVia, while seeding the population at the best starting fitness, shows the poorest performance. There appears to be a long term learning cost associated with using promoted functions from a previous GP search in the primitive function set, even as it gives better initial performance.

For both levels 2 and 3, SigTest 5 distinguishes itself above the other function judgement strategies, and performs almost as good as NoFunc asymptotically. Even though the domain shows no favoritism towards extracting functions, the SigTest strategy mitigates the penalty for sharing extracted functions, while still providing initial performance gains. By decreasing the *DecencyThreshold* to 3, from 5, Figure 7.5(b) shows that the SigTest strategy can achieve initial fitnesses comparable to the other function identification strategies, while still maintaining the asymptotic performance of SigTest 5.

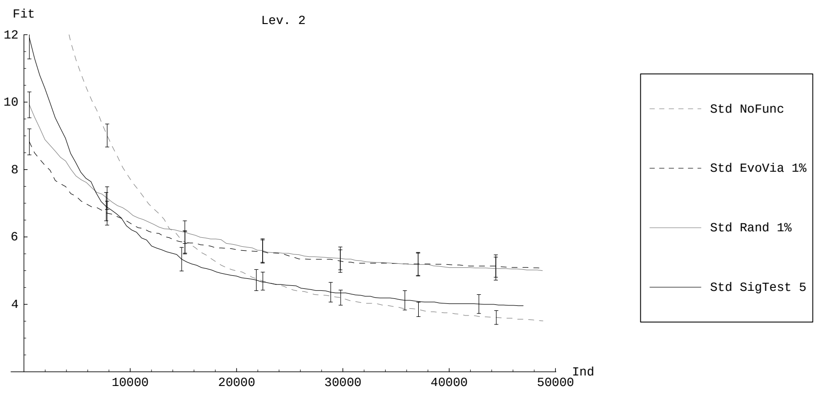
Looking at Table 7.2, one can see that there is a large difference between EvoVia and SigTest 5 in the average number of functions promoted each run. SigTest 5 promotes about one fourth as many functions each run. Lowering *DecencyThreshold* to 3 lowers the performance differential expectations (i.e. the SigTest strategy become less picky and promotes more functions). SigTest 3 promoted on average 40.47 per run, which is more than EvoVia. If the performance difference between SigTest x and EvoVia is simply due to the number of functions promoted in a run, then we should see SigTest 3 perform poorer than EvoVia. Figure 7.5 shows this is not the sole explanation, since SigTest3 also outperforms EvoVia. It is interesting to see that both EvoVia and SigTest3 start at essentially the same initial fitness in levels 2 and 3, but quickly diverge. SigTest 3 does not suffer from an early learning rate plateau as severely as EvoVia. Also worthy of note is by level 3, SigTest 3 and SigTest 5 perform so similarly, despite the stark difference in the average number of functions promoted during a run.

Table 7.2. Average number of functions promoted per run using the standard breeding policy over 100 generations

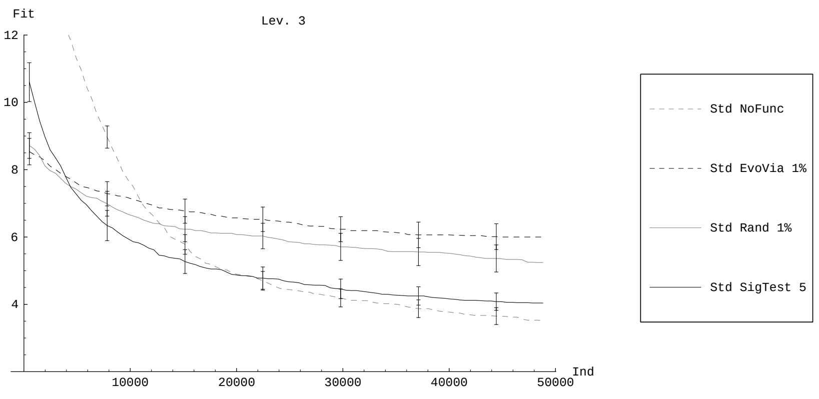
Best	6.44
EvoVia	34.81
Rand	45.14
SigTest 3	40.47
SigTest 5	8.63



(a)

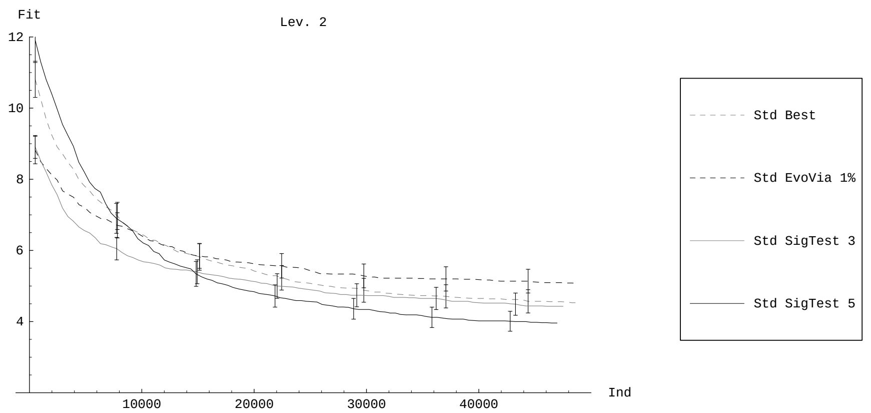


(b)

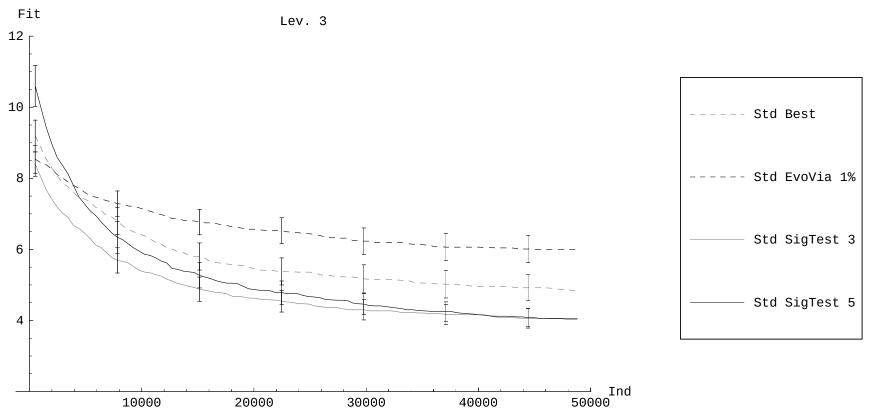


(c)

Figure 7.4. (a), (b), and (c) correspond to levels 1, 2, and 3, respectively. The horizontal axis represents the mean number of individuals evaluated for the level over all runs. The vertical axis represents the mean fitness of the best individuals over all (100) runs for the particular level. Search duration was 100 generations. The vertical bars are 95% confidence intervals, shown every 15 generations. The standard breeding policy was used.



(a)



(b)

Figure 7.5. (a)Level 2 and (b)Level 3. Mean fitness, over 100 runs, of the best individual each generation. The vertical bars are 95% confidence intervals, shown every 15 generations. The standard breeding policy was used.

Figure 7.6 shows results for the four function identification methods on level 3. SigTest 5 significantly outperforms the other methods in terms of number of individuals evaluated, under the standard breeding policy. It should be noted, as stated previously, that NoFunc surpassed them all by the end of level's GP search.

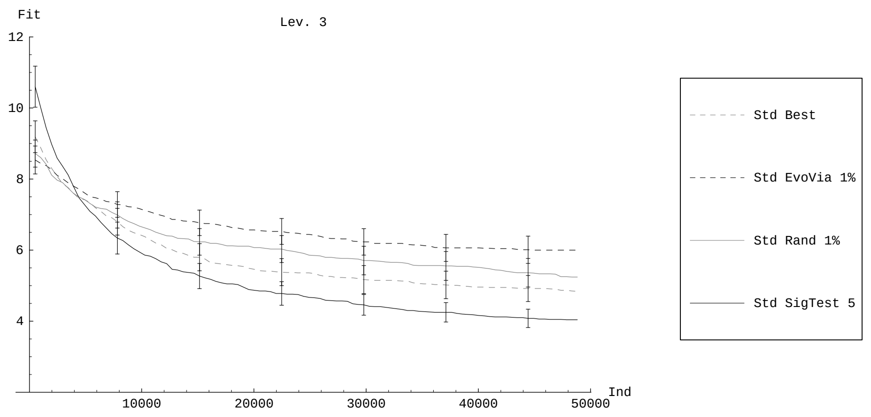


Figure 7.6. A quick summary of the four function identification strategies on level 3, all using the standard breeding policy.

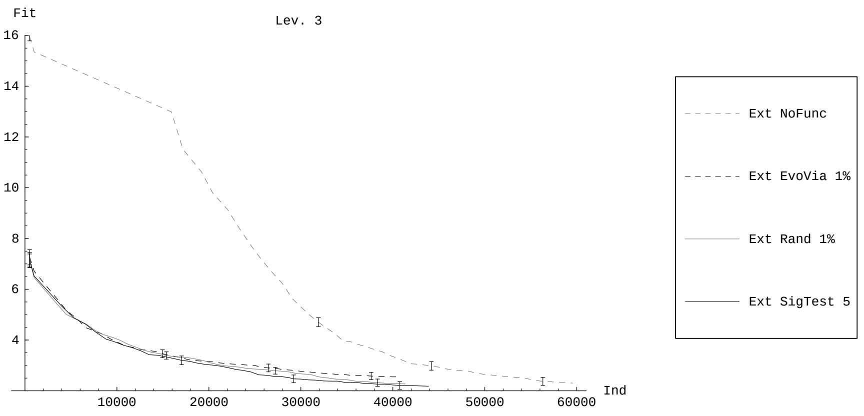
7.3 Extended Breeding Policy

Running the same experiments as in the previous section (7.2) with the extended breeding policy produced some unexpected results. Remember the extended breeding policy throws out offspring that don't meet the diversity requirements, thus only sufficiently diverse offspring survive for a chance at reproducing for the next generation. In this case, the diversity requirement is that the offspring produce a board configuration not already produced in the new population. Only the level 3 plots are shown in Figure 7.7 because level 2 and level 3 performance were virtually identical. Level 1 is not shown, because as was the case with the standard breeding policy, all the function identification strategies performed identically with NoFunc. Because offspring not meeting the diversity requirements are thrown away, the number of individuals evaluated per generation can be much higher for the extended breeding policy than the standard one. Table 7.3 shows the average number of functions promoted each run.

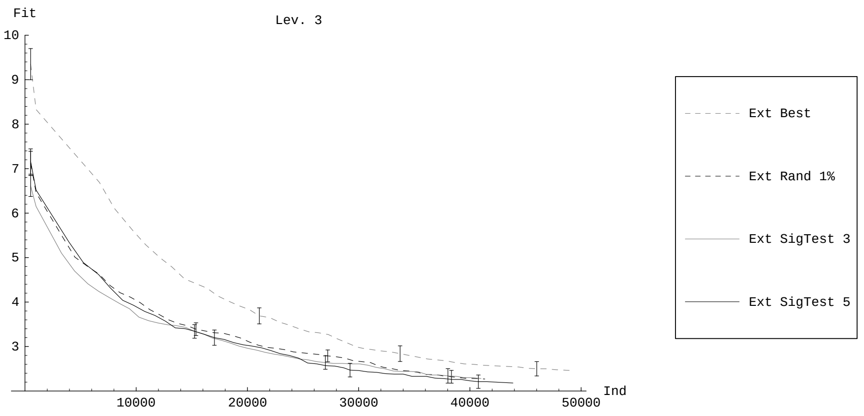
As we see in Figure 7.7(a), NoFunc doesn't surpass the function identification methods. NoFunc matches the performance of the function identification methods, but using roughly 50% more individuals. The extended breeding policy homogenized the

Table 7.3. Average number of functions promoted per run using the extended breeding policy over 50 generations

Best	5.43
EvoVia	16.29
Rand	26.22
SigTest 3	47.99
SigTest 5	26.06



(a)



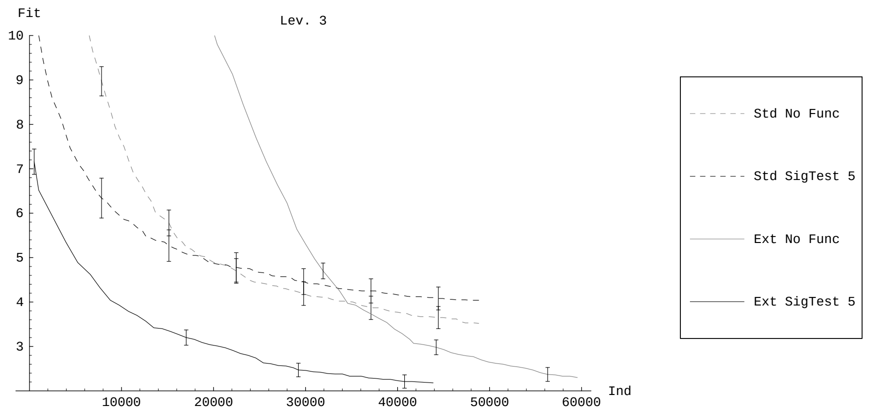
(b)

Figure 7.7. Both (a) and (b) are level 3, comparing different strategies. Mean fitness, over 100 runs, of the best individual each generation. The vertical bars are 95% confidence intervals, shown every 15 generations. Search duration was 50 generations. The extended breeding policy was used.

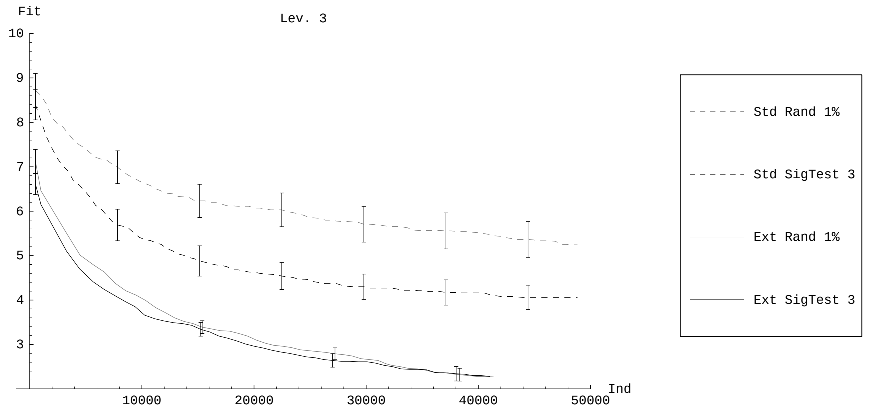
average best fitness performance of all the function identification strategies, except Best. The most unexpected result was Rand performing so well. It is difficult to justify spending the overhead needed to run the significance testing or any other heuristic for identifying exceptional functions when promoting functions at random does just as well. It seems the extended breeding policy plays a strong role in function extractions such that the method of choosing functions to promote from the set of extracted functions is largely irrelevant. The more discriminating breeding policy produced a bias towards populations with a diversity of structure that is inferred from the fact that each member of the population produces a different end configuration. The Rand strategy performing as well SigTest and EvoVia suggests that the probability a module will be beneficial to a GP search is roughly uniform across strategies and that each strategy selects similar traits.

7.4 Standard vs. Extended

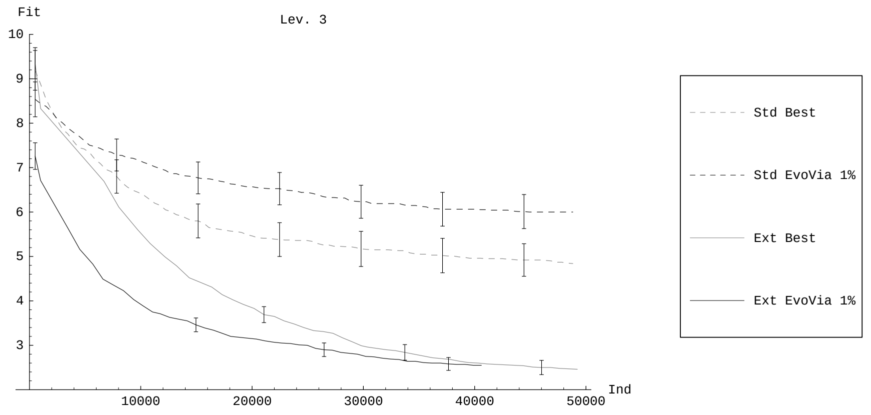
A head to head comparison of selected standard and extended breeding policy results is shown Figure 7.8. It shows the extended breeding policy to be much more efficient terms of best individual per individual evaluated. Figure 7.8(b) also illustrates the collapse in performance difference between SigTest 3 and Rand going from the standard to extended breeding policy. Given that the extended breeding policy produces better fitnesses on a per individual evaluated basis, why would one not employ the extended breeding policy and randomly pick modules to promote in all GP searches? One reason is it may not be practical or possible in all problem domains to have a diversity metric that implements result equality checks. Without an equality checking mechanism the extended breeding policy cannot be used. Wasting CPU time on offspring that end up thrown away is another consequence of the extended breeding policy. It may be that a breeding policy with less strict diversity requirements, such as specifying a probability that an offspring not meeting the diversity requirements is *not* thrown away, may perform as well as the extended policy with less CPU waste.



(a)



(b)



(c)

Figure 7.8. A comparison of various standard and extended policy results. (a), (b), and (c) all represent results for level 3.

CHAPTER 8

CONCLUSION

8.1 Observations

Sharing extracted functions across GP searches seems to be beneficial to the learning of subsequent problems within the same domain, although the benefit decreases as generations pass, to the point where the no-function strategy catches up with function sharing strategies in terms of fitness of the best individual.

Using a more discriminating breeding policy than the standard one renders the different function identification strategies virtually indistinguishable from each other. If the evaluation of individuals is a substantial fraction of total simulation run time, imposing diversity constraints on the breeding policy coupled with function sharing could realize a large benefit over no function sharing.

While the rate of learning, in terms of fitness, increased with function sharing, the end fitness stayed approximately the same. Implementing a more sophisticated fitness function for the tested problem domain might address this. One possible modification to the fitness function of Lights Off domain is giving a slight preference to individuals that produce board configurations with the remaining lights clustered together. Figure 8.1 illustrates this.

8.2 Applicable Domains

The Lights Off domain proved to be a poor choice for module extraction, shown by the fact that NoFunc and all the function strategies performed the same for the first level. Lights Off plays have side effects that move out in all directions, potentially

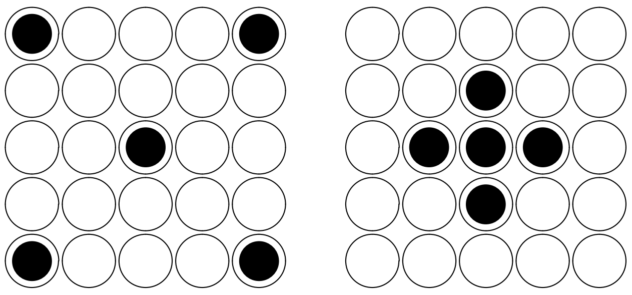


Figure 8.1. Both board configurations have a fitness of 5, though the configuration on the right is one play away from turning off all the lights. An individual producing the right configuration would clearly be outperforming an individual that produces the left configuration, but the current fitness function doesn't differentiate between the two..

impacting many neighborhoods. It is possible that proper context checks[1] to ensure module applicability are too complex to evolve within the current experimental set up. If the condition statements can never be sophisticated enough to capture the nuances of Lights Off, no function will ever have anything but apparently random chance of being labeled "beneficial", no matter which strategy is applied, thus explaining why Rand performs so well. Functions should only be called under certain circumstances, and if the program structures that properly describe these circumstances are rare to evolve, then whether a function should be called is (a seemingly) random event, due to some dependent variables not being visible (to the condition for a module call).

Domains that would probably perform better with modules would have operations or events that can be decided with very little context needed. The Tower Of Hanoi puzzle, a block stacking problem, is an example of a domain where there is known regularity and modularity to solutions, so it is likely that it can benefit from introducing module extraction towards this problem domain. Evolving analog circuits also seems like a promising domain for exploiting modules since many circuits exhibit regularity. For

[1]Here we mean a context check to be a series of condition clauses evaluated to determine whether to call a module.

instance, filters evolved by Koza et al.[26] display regularity in the form of identical or very similar sub-circuits linked together in series.

A separate application for the seeding of GP searches with modules from previous GP searches is in self-healing software. Imagine that a robot controlled by GP evolved software suffers a malfunction. The robot could drastically reduce the needed CPU time to evolve a new controller (on board) customized to the new hardware situation by sharing previously extracted modules with the current GP search. If there are real-time constraints, this decreased evolution time could be critical to adapting fast enough to the environment. Figure 7.7(a) shows this significant CPU savings in terms of number of individuals evaluated.

8.3 Future Work

Future work on this topic might include a method to identify and remove functions that were initially deemed *good* but are actually hindering GP search progress, as well as implementing more problem domains to test with. A known modular domain, like the Tower of Hanoi, should be tested to get a better idea of how module extraction can benefit a GP search, and which function identification strategy is best.

Another interesting point worth looking is after the initial random population of level that has benefited from shared functions is created, to go through the entire population dissolving every function in each individual. This might help mitigate the early learning plateau penalty sharing functions imposes while still benefiting from the initial advantage in fitness.

REFERENCES

[1] G. R. Ruth, "Automatic programming: Automating the software system development process," in *ACM '77: Proceedings of the 1977 annual conference.* New York, NY, USA: ACM Press, 1977, pp. 174–180.

[2] J. Gray, "What next?: A dozen information-technology research goals," *J. ACM*, vol. 50, no. 1, pp. 41–57, 2003.

[3] D. Whitley, "A genetic algorithm tutorial," Sept. 06 1993. [Online]. Available: http://citeseer.ist.psu.edu/29471.html; http://bioinfo.cpgei.cefetpr.br/mirrors/encore/GA/papers/tutor93.ps.gz

[4] ——, "An overview of evolutionary algorithms: practical issues and common pitfalls," *Information and Software Technology*, vol. 43, no. 14, pp. 817–831, 2001. [Online]. Available: citeseer.ist.psu.edu/whitley01overview.html

[5] H. Stringer and A. S. Wu, "Behavior of finite population variable length genetic algorithms under random selection," in *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation.* New York, NY, USA: ACM Press, 2005, pp. 1249–1255.

[6] K. Katayama, H. Hirabayashi, and H. Narihisa, "Performance analysis for crossover operators of genetic algorithm," *Transactions of the Institute of Electronics, Information and Communication Engineers*, vol. J81D-I, no. 6, pp. 639–650, 1998.

[7] T. Bäck, "Selective pressure in evolutionary algorithms: A characterization of selection mechanisms," in *Proc. of the First IEEE Conf. on Evolutionary Computation.* Piscataway, NJ: IEEE Press, 1994, pp. 57–62.

[8] M. Mitchell, *An Introduction to Genetic Algorithms.* Cambridge, MA: MIT Press, 1996.

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

[10] L. Spector, "What is the difference betweeen ga and gp," 19 Nov. 2005. [Online]. Available: http://groups.yahoo.com/group/genetic_programming/message/3680

[11] J. R. Koza, "Scalable learning in genetic programming using automatic function definition," in *Advances in Genetic Programming*, K. E. Kinnear, Jr., Ed. Cambridge, MA, USA: MIT Press, 1994, ch. 6, pp. 99–117.

[12] ——, *Genetic Programming II: Automatic Discovery of Reusable Programs.* Cambridge Massachusetts: MIT Press, May 1994.

[13] J. B. Pollack and P. J. Angeline, "Coevolving high-level representations," Apr. 02 1997. [Online]. Available: http://citeseer.ist.psu.edu/125247.html; http://www.demo.cs.brandeis.edu/papers/alife3.ps.gz

[14] S. C. Roberts, D. Howard, and J. R. Koza, "Evolving modules in genetic programming by subtree encapsulation," in *Genetic Programming, Proceedings of EuroGP'2001*, ser. LNCS, J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, Eds., vol. 2038. Lake Como, Italy: Springer-Verlag, 18-20 Apr. 2001, pp. 160–175.

[15] A. Dessi, A. Giani, and A. Starita, "An analysis of automatic subroutine discovery in genetic programming," in *Proceedings of the Genetic and Evolutionary Computation Conference*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., vol. 2. Orlando, Florida, USA: Morgan Kaufmann, 13-17 July 1999, pp. 996–1001. [Online]. Available: http://www.cs.bham.ac.uk/ wbl/biblio/gecco1999/GP-432.pdf

[16] M. Ahluwalia and L. Bull, "Co-evolving functions in genetic programming: Dynamic ADF creation using GLiB," *Lecture Notes in Computer Science*, vol. 1447, pp. 809–818, 1998.

[17] O. Brock, "Evolving reusable subroutines for genetic programming," in *Artificial Life at Stanford 1994*, J. R. Koza, Ed. Stanford, California, 94305-3079 USA: Stanford Bookstore, June 1994, pp. 11–19. [Online]. Available: http://robotics.stanford.edu/users/oli/PAPERS/a-life.ps

[18] N. Hondo, H. Iba, and Y. Kakazu, "Sharing and refinement for reusable subroutines of genetic programming," in *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*, vol. 1, Nagoya, Japan, 20-22 May 1996, pp. 565–570.

[19] S. Luke, ECJ 11: A Java evolutionary computation library. http://cs.gmu.edu/∼eclab/projects/ecj/, 2004.

[20] D. J. Montana, "Strongly typed genetic programming," *Evolutionary Computation*, vol. 3, no. 2, pp. 199–230, 1995.

[21] A. M. Law and D. M. Kelton, *Simulation Modeling and Analysis.* McGraw-Hill Higher Education, 1999.

[22] E. K. Burke, S. Gustafson, G. Kendall, and N. Krasnogor, "Is increased diversity in genetic programming beneficial? an analysis of the effects on performance," in *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, R. Sarker, R. Reynolds, H. Abbass, K. C. Tan, B. McKay, D. Essam, and T. Gedeon, Eds. Canberra: IEEE Press, 8-12 Dec. 2003, pp. 1398–1405.

[23] A. Ekárt and S. Z. Németh, "Maintaining the diversity of genetic programs," in *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, ser. LNCS, J. A. Foster, E. Lutton, J. Miller, C. Ryan, and A. G. B. Tettamanzi, Eds., vol. 2278. Kinsale, Ireland: Springer-Verlag, 3-5 Apr. 2002, pp. 162–171. [Online]. Available: http://www.sztaki.hu/∼ekart/eurgp2.ps

[24] R. E. Keller and W. Banzhaf, “Explicit maintenance of genetic diversity on genospaces,” Feb. 13 1997. [Online]. Available: http://citeseer.ist.psu.edu/32228.html; http://www.cs.mun.ca/∼banzhaf/papers/genDiv.ps.gz

[25] N. F. McPhee and N. J. Hopper, “Analysis of genetic diversity through population history,” in *Proceedings of the Genetic and Evolutionary Computation Conference*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., vol. 2. Orlando, Florida, USA: Morgan Kaufmann, 13-17 July 1999, pp. 1112–1120. [Online]. Available: http://www.cs.bham.ac.uk/∼wbl/biblio/gecco1999/GP-421.pdf

[26] J. R. Koza, F. H. Bennett, III, D. Andre, M. A. Keane, and F. Dunlap, “Automated synthesis of analog electrical circuits by means of genetic programming,” *IEEE Trans. on Evolutionary Computation*, vol. 1, no. 2, pp. 109–128, 1997.

BIOGRAPHICAL STATEMENT

Anthony Loeppert received a B.S. in Computer Sciences from the University of Texas at Austin in 1999. He is married and has one son. He now lives in San Diego, CA and works for Motorola Inc.