Genetic Programming: Theory, Implementation, and the Evolution of Unconstrained Solutions

Alan Robinson

Division III Thesis	Committee:	Lee Spector
Hampshire College		Jaime Davila
May 2001		Mark Feinstein

Contents

Part I: Background

Part II: PushGP

Part I

Background

1 Introduction

1.1 Background – automatic programming

Programming software is a difficult task, requiring careful attention to detail and a full understanding of each problem to be solved. Even when designed under the best of conditions, programs are often brittle; they can only successfully deal with situations for which they were explicitly designed. Not only are programs unable to adapt to new situations on their own, but programmers often find it very hard to modify a program to deal with situations not originally considered in the initial design. Particularly in the field of artificial intelligence this leads to difficulties because the problems being solved are often poorly understood at the outset and change significantly with situation and context.

The field of automatic programming focuses on making the computer take over a larger part of program construction. The ultimate goal is that the programmer only needs to rigorously state *what* an algorithm must do, not *how* it does it, and the automatic programming algorithm figures out the implementation. Depending on the speed of the programming algorithm, the work could be done offline (initiated by the programmer when a new algorithm is desired) and saved for later reuse, or generated on the fly for solving new problems as they show up by a program interacting with an end user or the world. Currently, no automatic programming system is fast enough to generate programs that solve interesting problems on the fly. As computers get faster this will become more possible, but already automatic programming systems are able to solve interesting, difficult problems offline, given enough time and computing resources.

There are several major contemporary methods for automatic programming, the most prominent two being Artificial Neural Networks (ANN)/Parallel Distributed Processing(PDP), and Genetic Programming (GP). While both systems have their strengths, in this project I have chosen to focus exclusively upon genetic programming.

Briefly, genetic programming is the iterative application of the Darwinian principles of adaptation by natural selection and survival of the fittest to populations of computer programs that reproduce, generation by generation, with mutation and recombination of their genetic material (program code). Chapter 2 will give a more detailed review of genetic programming and the field.

I have chosen genetic programming as the focus of this project because it possesses several significant traits:

- It has been successfully applied in a wide range of problem domains.
- It has not only solved diverse toy problems, but also produced human-competitive solutions to real-world problems (see Koza (2000) for a review).
- It scales well with faster CPUs and/or more CPUs.

All of these traits suggest that genetic programming has the potential to become a useful tool for discovering new algorithms and solutions to hard programming problems.

1.2 This project

This project consists of two related components:

1. The development from scratch of a linear genetic programming environment (called LJGP) in the Java programming language. The educational goal of this work was to develop a deeper understanding of the mechanics and design considerations inherent in the construction of a contemporary genetic programming system. The practical goal was to build a genetic programming system suited to harnessing the computational resources available to researchers such as myself, who do not have grant money to purchase hardware to dedicate to research. This work is described in detail in Part III of this document.

2. The investigation of techniques designed to allow genetic programming to evolve significantly more complex, modular, and functionally expressive code. Rather then developing a system from scratch, the research in Part II of this document builds upon the PushGP system developed by Spector (2001). PushGP uses a stack-based language with multiple stacks for operating on different data types. One stack stores program code and allows for interactive construction and modification of executable functions, modules, and control structures as the main program executes.

   The primary question addressed in Part II is what sort of modularity and structure evolve when their very composition arises from the evolutionary modifications of program code, rather than as a hard-coded layer imposed by the genetic programming system. The secondary question is the effect of this technique on computational effort compared to more traditional genetic programming systems (like Koza's GP system with automatically defined functions).

These two components were selected to provide a wide exposure to the field of genetic programming. The first engages implementation issues and theoretical considerations for genetic programming as it commonly practiced. The second explores the boundaries of current research, attempting to advance the state of the art of genetic programming.

1.3 Summary of chapters

Part I: Background
Describes the overall goals of the project and the background in genetic programming necessary to understand the work in parts II and III.

> **Chapter 1: Introduction**
> Briefly describes the project and the layout of the document.

Chapter 2:Genetic Programming Review
Describes genetic programming in more depth, assuming that the reader is familiar with computer science, but not machine learning or genetic programming. Includes both a brief two page overview, and much more in-depth coverage of the contemporary techniques of the field. Readers familiar with genetic programming may skip to part II or III of the document.

Part II: PushGP
Contains the research pertaining to PushGP and its use in the evolution of complex control structures and modularity.

Chapter 3: The Push Language & PushGP
Introduces the theoretical motivations for evolving modularity and structure as an outgrowth of program code itself, rather than as a layer imposed by the genetic programming system. It then describes the Push language and PushGP, the genetic programming system that provides the experimental basis for exploring practical outcomes of these theoretical goals.

Chapter 4: PushGP Compared to GP2 with ADFs
Systematically compares the computational performance of PushGP, and the nature of the modularity evolved, for problems that were shown to have modular solutions by Koza in his work with automatically defined functions in Genetic Programming II (1994). This is done with two classes of problems: symbolic regression and even-N-parity classification.

Chapter 5: Variations in Genetic Operators
Investigates variations in mutation and crossover that attempt to address several problems inherent in the default genetic operators used by PushGP. Focuses on remedies for code growing beyond the maximum program length, and also addresses the limited likelihood of some program code to be modified by genetic operators. This is forward looking research, in that none of these variations are implemented in the runs documented in other chapters.

Chapter 6: New Ground – Evolving Factorial
Presents initial experiments with evolving factorial functions and proposes future refinements that might lead to more successful results.

Part III: LJGP
Contains the discussion of the work on LJGP, the genetic programming system authored in Java that was built for both its educational value and practical use.

Chapter 7: Linear Coded Genetic Programming in Java
Introduces LJGP and describes the theoretical and practical motivations behind the LJGP system and its implementation.

Chapter 8: LJGP User's Guide
Describes now to encode new problems in LJGP, how to read LJGP programs, and discusses the different packages and classes that make up LJGP.

Chapter 9: LJGP Applied
A narrative account of the experiments conducted with LJGP to test its functionality and pilot several research ideas. The research explores the computational constraints of foraging behaviors in simulations designed to capture some of the important issues encountered in real world foraging and navigation.

Conclusion
Summarizes findings from the work with PushGP and LJGP and discusses future research directions.

2 Genetic Programming Review

This chapter describes genetic programming in more depth, assuming that the reader is familiar with computer science, but not with machine learning or genetic programming.

Genetic programming was developed and popularized by John Koza. For a thorough treatment of the topic the reader is encouraged to consult the seminal works in the field, *Genetic Programming* (Koza, 1992) and *Genetic Programming II* (Koza, 1994).

2.1 What is genetic programming: a brief overview

Genetic programming is a collection of methods for the automatic generation of computer programs that solve carefully specified problems, via the core, but highly abstracted principles of natural selection. In a sentence, it is the compounded breeding of (initially random) computer programs, where only the relatively more successful individuals pass on genetic material (programs and program fragments) to the next generation. It is based upon two principles:

1. It is often easier to write an algorithm that can measure the amount of success a given program has at solving a problem, than to actually write the successful program itself.

2. A less-than-fully-successful program often contains components, which, appropriated in the right way, could contribute to designs for more successful programs.

The first principle states why genetic programming is a useful technique: it can make developing difficult algorithms easier. To evolve an algorithm using a preexisting GP, the programmer writes a problem-specific algorithm that takes a computer program as input, and returns a measurement of how successfully that program solves the problem of interest. This problem-specific algorithm is called the fitness function (or fitness metric). In order to be useful it must return more than just whether the program works perfectly or not. Since genetic programming is a gradual process, the fitness function must be able to differentiate between very good solutions, fair solutions, poor solutions, and very, very bad solutions, with as many gradations inbetween as is reasonably possible.

The second core principle suggests why genetic programming might perform better in some cases than other fitness-driven automatic programming techniques, such as hill climbing and simulated annealing. Rather than blindly searching the fitness space, or searching from randomly initialized states, genetic programming attempts to extract the useful parts of the more successful programs and use them to create even better solutions. How does the system know which parts of a program are useful, and how to combine them to form more fit solutions? By randomly selecting parts of the more successful programs, and randomly placing those parts inside other successful programs. Genetic programming relies upon the fitness function to tell if the new child received something useful in the process. Often the child is worse for its random modification, but often enough the right code is inserted in the right place, and fitness improves. It is worth noting just how often this happens, and why, is an active area of debate (see Lang (1995) and Angeline (1997)). This process of randomly combining relatively fit individuals, however, is still standard in genetic programming, and subsequently is the method used in the research described in this document.

Given the two core principles described above, the one significant implementation issue remaining is a programming language that allows for random modification of code. The solution is to treat code as high-level symbols, rather than as sequences of text. This requires that the symbols combine in uniform ways, and that a dictionary of symbols is created that describes what inputs the symbols take. Specified in this way, a programming language such as Lisp is easily modified and spliced at random. Other languages are also amenable to this process, when constrained syntactically.

Given the programming language and a fitness metric, the steps executed by a genetic programming algorithm are straightforward:

Initial Population	With an algorithm that allows random generation of code, an initial population of potential solutions can be generated. All will be quite inept at solving the problem, as they are randomly generated programs. Some will be slightly better than others, however, giving evolution something to work with.
Fitness Ranking	Via the fitness metric, the individual programs are ranked in terms of ability to solve the problem.
Selection	The closer (better) solutions are selected to reproduce because they probably contain useful components for building even better programs.
Mating	At random, chunks of those selected programs are excised, and placed inside other programs to form new candidate solutions. These “children” share code from both parents, and (depending on what code was selected) may exhibit hybrid behavior that shares characteristics of both.
Mutation	To simulate genetic drift/stray mutation, many genetic programming systems also select some of the more fit programs and directly duplicate them, but with a few of their statements randomly mutated.

Repetition Until Success	From here, the process starts over again at Fitness Ranking, until a program is found that successfully solves the problem. Not every child will be more fit than its parent(s), and indeed, a very large percentage will not be. The expectation, however, is that some children will have changes that turn out to be beneficial, and those children will become the basis of future generations.

Note that the random makeup of the initial population has a large effect on the likelihood that GP will find a successful program. If a single run does not succeed after a reasonable period of time, often it will succeed if it is restarted with a new random population.

These steps constitute the core of most genetic programming systems, though all systems tweak, or completely change, many aspects of these steps to suit the theoretical interests pursued by their designers. The field is still young and there is no standard of what a genetic programming system must include, nor how it must proceed from step to step.

2.2 Contemporary genetic programming: in depth

This section is intended to acquaint the reader with the inner workings of genetic programming, thus providing a context in which to understand the motivations behind my own work. It starts with a detailed description of a typical genetic programming system and concludes with a summary of other popular methodologies pursued in contemporary research.

The following example is meant to be easy to understand and mirror the typical construction of a genetic programming system. It is certainly not the only way that genetic programming is employed, but it is representative of the field. It describes a tree-based genetic programming system and the steps required to make it evolve a mathematical function that fits a set of data points.

2.3 Prerequisite: a programming language amenable to (semi) random modification

In order to understand the issues discussed in the next few sections, the construction of a programming language suitable for evolution must first be discussed.

Conventional genetic algorithms operate on bit strings or other representations that lend themselves to random modifications. Genetic programming requires a programming language that is similarly mutable. Many such languages have been created for use in genetic programming. In Koza’s first genetic programming book, he demonstrated how Lisp can be constrained to a more uniform subset which allows random modification of programs. Lisp already has a very uniform structure, in that every statement is an S-expression that returns a value to its caller. An S-expression is a list of expressions, where the first is the function to apply and each additional expression is an argument to that function, or another S-expression, whose result, when evaluated, is an argument to that function).

For example an S-expression that adds 2, 3, and 5, and the result of multiplying 2 and 3 is written as `(+ 2 3 5 (* 2 3)).` S-expressions are typically drawn/represented as trees, with the root node of each branch the function, and each leaf the arguments for that function. Hence, when discussing the modification a S-expression, it is usually stated in the terms of modifying a tree.

Koza's modifications to Lisp consisted of removing most of the functions and writing wrappers for others so that they do not crash when given invalid arguments and always return a value. All returned values were of the same type, so that they could serve as arguments for any other function. Finally, every function was defined as taking a fixed number of arguments.

This means that a random expression, or entire program, can be generated by choosing a function at random as the base of the tree, and then filling each of its arguments (nodes) with more randomly chosen functions, recursively, until the amount of code reaches the target size. At that point, growth of the tree is halted and the remaining empty arguments are filled in with terminals (constant values, variables, or functions that take no arguments).

Note that the uniformity of the language also means that inserting a randomly selected or generated chunk of code into an already existing program is also easy. It just requires selecting a single terminal or function at any depth within a tree and replacing it with that chunk.

2.4 Steps specific to each problem

Most of the code and parameters that make up a genetic programming system do not change as it is applied to different problems. In this section the few steps that *are* necessary for defining the problem are outlined.

In this example, a tree based genetic programming system (meaning that the programs evolved are represented by tree structures) will attempt to evolve a program that, given an input value, produces an output value matching the function $\mathbf{y = x^4 + x^3 + x^2 + x}$. The creation of a function that matches a collection of output values is known as symbolic regression.

2.4.1 Create fitness function

The fitness function, being the only real access the system has to the problem, must be carefully designed, with a maximum amount of discrimination possible, for a given run time. Binary responses (perfectly fit, or not) will not work because the selection of who in the population mates must be informed by their closeness to the solution – otherwise the children will be no more likely to have higher fitness than any other member of their parents' population.

In symbolic regression, the only explicit knowledge of the target function that the fitness metric has access to is a table of **x** values and corresponding **y** values. In general, whenever there are multiple inputs for which a program must answer correctly, each input is termed a fitness case.

While the fitness metric could answer whether or not the program submitted for testing gets all fitness cases correct, this level of detail is insufficient for evolutionary purposes, as explained above.

Another possibility is to report how many fitness cases were answered correctly, with a higher number representing a more fit individual. This metric is sensible for some types of problems, but, in this case, a more fine-grained metric is possible, and generally the more precise the metric, the better evolution performs.

Since the fitness cases and the actual output of the program are both numeric, the number line distance between both is easily calculated. Furthermore, this value is likely a good way to gauge which among a collection of many poorly performing programs is closer to the goal. The smaller the value, the more fit the program is for that fitness case.

All that remains is to combine these per-fitness-case metrics together to produce a fitness measure representing the performance of the program across all fitness cases. Obviously, adding each fitness value together would produce one such summary value. If, perhaps, there were reason to think that some fitness cases were more important than others, then those results could be multiplied by a scaling factor, so that they contribute more to the measure of fitness.

Note that, by convention, a fitness metric reports the amount of error between the tested program and the target solution, meaning that the perfect program has a fitness of zero. In cases where the easiest measure of fitness increases as the program becomes more fit, it is standard to calculate the maximum possible fitness, and then report back to the evolutionary subsystem the difference between this maximum possible fitness and the measured fitness of the individual. This way zero still represents a perfect individual.

The final step related to measuring fitness is selecting the number of fitness cases to include in each fitness evaluation. More cases produce a higher resolution total fitness value, but also means that measuring fitness will take longer. There is no hard and fast rule, but the standard practice is to start with a small number and increase it if evolution fails. In this example of symbolic regression, a common number of fitness cases is 10, and 50 would not unreasonable. For this example 20 strikes a good balance between the time required to test all cases and level of accuracy provided by that number of samples.

2.4.2 Choose run parameters

In general there are several parameters which control the operation of evolution that may be worth changing, depending on the problem being attacked.

Population Size	The number of individuals created for each generation. Larger values slow down the time it takes to produce a new generation, and small values often lead to lack of diversity in a population and, eventually, premature convergence on substandard solutions. For most difficult problems, bigger is better, and the actual size selected depends on what will easily fit in RAM. Common values range from 500 to 16,000. For an easy problem like this, 500 is a reasonable value.
Maximum Generations	The number of generations to evaluate before assuming that a particular run has converged prematurely, and should be aborted. Typically, evolution progresses most quickly early on, and further improvements take more and more generations. The exact point that it becomes counterproductive to continue a run is only approximateable via statistical analysis of a collection of runs after the fact, but common values are 50 and 100 generations. Generally, the size of the population is increased instead of the number of generations, though recently using more than 100 generations has started to gain popularity. For this problem 50 is a typical choice.
Maximum Initial Program Length/Tree Depth	When creating the initial population, this specifies the maximum size of each new individual. The meaning of this value is wholly dependant on the representation of the programming language used. For tree-based representations, this specifies the maximum allowed depth from the root node of the program. Overly large values will slow the initial evaluation of fitness. Overly small values will not allow enough variation between individuals to maintain population diversity, and will decrease the likelihood that there will be interesting differences in fitness between randomly generated members.
Maximum Program Length/Tree Depth	This setting determines how large individuals can grow during the process of a run. Unless the fitness function biases against non-functional code, evolved programs tend to grow in size with each passing generation. This is not necessary a bad thing, as long as the growth is limited to keep programs from consuming exceedingly large amounts of computational resources.

Success Criteria	For some problems it is easy to tell when the solution has been found. For symbolic regression, this is frequently when fitness reaches 0. On the other hand, perfect fitness can, at times, be very hard to achieve (particularly, if floating point numbers are manipulated in the process of producing the result). For those cases, the success criteria might be that the total fitness be within 5% of perfect, or that the performance on each fitness case has no more than 1% error. For this problem, since it deals with integer fitness cases, zero error (i.e., perfect fitness) is reasonably achievable, and so the success criteria in this example is zero error.
%Crossover, %Mutation, %Duplication, etc.	In any genetic programming system that has more than one method for producing children, the percentage that each “genetic operator” is applied must be specified. Usually this is done by setting the percentage of children in each generation that are produced by the different operators. For this problem, a reasonable setting is 10% duplication, 45% mutation, and 45% crossover, though any number of other combinations would also function reasonably. The common genetic operators will be described in section 2.5.4

2.4.3 Select function / terminals

One of the major areas of variation when encoding a problem for genetic programming is the selection of the actual types of code (function, variables, etc) made available to evolution. One option would be to offer the entire language and allow natural selection to build whatever program evolution favors. This can be problematic, however, as the more flexibility given to genetic programming, the more dead-end paths exist for evolution to explore (algorithms that might produce reasonable fitness, but can never be tweaked to produce the actual solution).

For instance, when doing symbolic regression of a simple curve, it may be evident by visual inspection that trigonometric functions are not required to fit the data. In such a case, it will only slow evolution down to provide sine, cosine, etc., in the function set. In practice, the set of functions, variables, and constants provided as the building blocks at the beginning of a run should be as small as possible.

For symbolic regression, it is common to provide only a few basic mathematical functions, such as addition, subtraction, multiplication, and protected division (division which returns a reasonable value (such as 0) instead of crashing when given a dividend of 0). Depending on the target values being evolved, it is also common to include either some random integers or random floating-point numbers to be used in the creation of constant values.

2.5 The genetic programming algorithm in action

All of the following procedures are the core steps executed by the genetic programming system, and typically remain unchanged between problems.

2.5.1 Generate random population

In this initial step, a wholly random population of individuals is created. It is from this seed population that natural selection will proceed. Each new program is created by iteratively combining random code together, following the rules that specify the legal syntax of the language, and randomly stopping at some point before the maximum initial size of an individual is reached.

For a tree-based code representation, a function for the root node is randomly selected, and then the contents of each leaf is set randomly. Then recursively the leaves of any new nodes are filled in, until the randomly selected depth is reached. This process is repeated for each new member of the population until a full population is created.

2.5.2 Measure fitness of population

In this step, the first of the main loop of a genetic programming system, each new member of the population is evaluated using the problem-specific fitness function. The calculated fitness is then stored with each program for use in the next step.

2.5.3 Select the better individuals from the population

In this step, the population is ranked with respect to fitness, and the more fit members are selected to reproduce. Various methods exist for selecting who reproduces. The simplest method is to repeatedly take the very best members of the population, until enough parents are chosen to produce the next generation. However this can be problematical. Just selecting the most fit programs leads to early convergence of the population to the best solution found early on. This does not allow less initially promising paths to be explored at all. Since the real solution is rarely the one easiest to evolve in just a few generations, it is important to allow some variation in the level of fitness required to reproduce.

A common way to allow such variation, while still ensuring that the better individuals are more likely to reproduce, is to select a few members of the population at once (say 4), and let the most fit of those reproduce. Sometimes all will have poor fitness, and a less fit individual will have a change to survive (as is the goal). Usually, however, the sample will represent a wide range of fitnesses, and the best one will have above-average fitness relative to the rest of the population.

2.5.4 Apply genetic operators until a new population is generated

Once parents are selected, they are used as input into the child producing algorithms known as genetic operators. There are many ways to produce children; the three most common are crossover, mutation, and duplication. These algorithms are called genetic operators because of their conceptual similarity to genetic processes.

Crossover takes two parents and replaces a randomly chosen part of one parent with another, randomly chosen part of the other. This is often very destructive to the structure and functionality of the child program. It is, however, the means by which valuable code can be transferred between programs and is also the theoretical reason why genetic programming is an efficient and successful search strategy. Even though it often produces unfit children, it does produce parent-superior fitness occasionally, and those individuals often posses the critical improvements that allow evolution to progress to the next round of fitness improving generations.

Mutation takes one parent and replaces a randomly selected chunk of that parent with a randomly generated sequence of code. One of the advantages of this operator is it maintains diversity in the population, since any of the function/terminal set can be inserted into the program, whereas crossover can only insert code present in the current generation's population.

Duplication takes a single parent and produces a child who is exactly the same as its parent. The advantage to this is that a well-performing parent gets to stick around in the population for another generation, and act as a parent for children of that generation. Without duplication there is the risk that a large percentage of the next generation will happen to be degenerate mutants with very poor fitness. By not saving some of the higher fitness individuals, evolutionary progress will be set back. Of course, if there is too much duplication, the population will never evolve.

Through the repeated application of these operators to the selected parents of the old generation, a new generations is formed, some of the members of which will hopefully be more fit than the best of the last generation.

2.5.5 Repeat until a program solves the problem or time runs out

At this point a new population is available to be evaluated for fitness. The cycle will continue, until ether a single member of the population is found which satisfies the problem within the level of error designated as acceptable by the success criteria, or the number of generations exceeds the limit specified.

As discussed earlier, if a run does not succeed after a large number of generations, it has probably converged onto a semi-fit solution and the lack of diversity in the population is drastically slowing evolutionary progress. Statistically, the likelihood of finding a successful individual is, at that point, most increased by starting the entire run over again with a wholly new random population.

2.5.6 Example run

Using the settings above, a tree based genetic programming system was able to evolve a solution to the problem $\mathbf{y = x^4 + x^3 + x^2 + x}$, given only 10 x and y value pairs. In one run, it found the solution after 20 generations:

The best randomly generated member of generation 0 was reasonably good for a random program. It outputted the correct response for 9 out of 20 fitness cases. Its code, as follows, is clear and easily recognizable as $x^2 + x$:

```
 (+ X (* X X ))
```

By generation 17, fitness had improved such that 16 out of 20 fitness cases were correctly answered by the best program found. The program, however, no longer bears much resemblance to the best program from generation 0:

```
(+ (+ (* (% (* (+ X (% (+ X X) X)) (% X X)) (% (% (+ X X) X) X)) (* X (+ X (* X X)))) X) (* (% X (%
X X)) X))
```

This program, in fact, seems to share little with the function $\mathbf{y = x^4 + x^3 + x^2 + x}$ (note that % stands for protected division). Inside it, however, lurks the building blocks of the correct solution, which is found at generation 20:

```
(+ (+ (* (% (* X (+ X (- X X))) X) (* X (+ X (* X X )))) X) (* (% X (% X X)) X))
```

Note the similarity between both programs (identical code in both is underlined). By inserting and removing code, a perfect solution was found from an imperfect, but reasonably close, solution.

It is worth noting that this solution looks very little like the sort of solution a human programmer would produce for this problem. The straightforward Lisp solution that follows from the structure of the function is `(+ (* x x x x) (* x x x) (* x x ) (* x))`. Given selective pressure towards smaller programs, genetic programming could have found a solution of this length. Nothing in the fitness metric rewarded smaller programs over larger programs, however, so code that did not impede fitness (but didn't add to it either) was still likely to remain between generations.

Generally, this tendency to evolve programs that contain unnecessary code goes by the term *code bloat*. While it causes programs to become larger than required and can impede evolution, measures that attempt to remove it can cause problems of their own. In some cases it even appears that code bloat is actually a useful part of evolution early on in a run. See Langdon, et al. (1999) for further discussion and research pertaining to code bloat.

2.6 Other types of GP

In Koza's work genetic programming is applied to Lisp program representations. One very common variation in genetic programming systems is the nature of program representations. Along with variations in tree-based representations that bear strong resemblance to Lisp, researchers have tried more radical departures. Genetic programming requires a uniform, systematic language that allows for mutation and splicing at arbitrary points. S-expressions are not the only syntax which provide this.

This section will briefly summarize some other programming representations utilized by genetic programming. The point of interest is that genetic programming can be applied to more than just the evolution of a single Lisp tree, and, indeed, could be applied to most programming paradigms, given a uniform programming language structure. While there are often practical advantages to these variations, it is important to note that no one GP implementation is the best. Different problems have different constraints that will be best exploited by different systems.

2.6.1 Lisp with Automatically Defined Functions (ADFs)

In this variation multiple trees of Lisp code are evolved per individual (Koza, 1992 & 1994). One tree, like in the genetic programming example described in sections 2.3-2.5, does the computations that are returned as the output of the program. This tree is called the result-producing branch. The other trees define functions that can be called by the program in the process of doing its calculations. This allows evolution to decompose one problem into several sub-problems, and combine those building blocks into the full solution.

For symbolic regression, for instance, ADFs might evolve that calculate the common terms in a function. In the result producing branch, those ADFs could then be referenced multiple times, rather than requiring the duplicate evolution of their functionality in separate places inside one tree.

When using ADFs it is necessary to specify how many functions will be defined by evolution, and the numbers of arguments each ADF requires. This can be done before the run begins, or during the run, by the addition of genetic operators that act on ADFs (Koza, 1994). Such operators take an individual and either add or subtract ADFs, or modify the number of arguments a specific ADF takes. Whatever way ADFs are managed, their addition requires new, more complex genetic operators that respect the structure of programs with ADFs.

This additional complexity, however, has benefits. Koza has shown that for problems that are decomposable into sub-problems, the addition of ADFs can reduce the computational effort required to find a successful solution (Koza, 1992, 1994 & 1999).

2.6.2 Linear genetic programming

In linear GP systems, programs are flat sequences of instructions with well-defined boundaries (Banzhaf, et al. 1998). Instructions can vary in length and are usually specified so that any order of complete instructions is a valid program. Crossover and mutation usually operate at the boundaries between instructions, making both types of operators very simple and efficient to implement. One of the advantages of linear GP, therefore, is the potential for faster wall-clock implementations of the algorithms that drive evolution.

2.6.3 Machine code genetic programming

Usually a variation upon linear GP, in these systems the actual genome evolved is the machine code of the host CPU. This allows for extremely fast, native execution of the program, but with the downside that the variations in syntax must match the underlying CPU, rather than what is most appropriate for the ease of evolution. As with other linear genetic programming systems, the crossover and mutation operators work at the instruction boundaries, so sequences of instructions are modified arbitrarily, rather than the bits that make them up. See Nordin (1994) and Nordin & Nordin J. (1997) for more details.

2.6.4 Stack-based genetic programming

Stacks can be added to most genetic programming systems with a few carefully designed instructions that push and pop data onto a stack. There is significant difference, however, between such an addition to a tree or stack based language and a truly stack-based system. In stack GP, *all* instructions/functions retrieve their arguments from the stack *and* push their results back onto the stack. See Bruce (1997) for more details.

2.7 The no free lunch theorem

The no free lunch theorem (Wolpert and Macready, 1997) has widespread ramifications for genetic programming research. Briefly stated, it proves that no single learning algorithm is better than any other for all problems. Indeed, for every learning algorithm, an infinite number of problems exists for which is it the best approach, and conversely, an infinite number of problems exist for which it is the worst possible approach. This proof applies equally to genetic programming and all learning systems, as well as for search in general, so it is not an issue specific just to genetic programming.

This theory means that for any GP to perform better than another on a problem, it must exploit some features of that specific problem that the other GP does not. The only reason that one could expect that a given GP system to perform better than another on a set of problems is if all of those problems share features that can be better exploited by one GP than the other.

This applies not just within genetic programming systems but also between entirely different techniques. Comparing genetic programming to blind search, for instance, genetic programming will not always perform better. Only in cases where the solutions that perform reasonably well are similar to the solution that solves a problem perfectly does a genetic programming system perform better than variants of a blind search.

In summary, empirical tests of genetic programming systems are most useful in terms of measuring performance on a given problem. Predictions made from them of performance on different problems are only valid if those new problems have the same features for the system to exploit.

2.8 Review of previous and contemporary GP results

The following sections will summarize a few samples of the problems to which genetic programming has been applied. The purpose is to demonstrate the wide range of domains in which it has seen profitable use.

2.8.1 Data classification

Symbolic regression

In the example of genetic programming given earlier, the problem solved was symbolic regression of a particular function. In that case the answer was already known. The point of symbolic regression, however, is it can also be applied to problems where the answer is unknown.

Given any set of data points, with any number of inputs and any number of outputs, genetic programming can attempt to find a function that transforms one into the other. Koza demonstrated that this could be used, among other things, to do symbolic differentiation of a known function, symbolic integration of a known function, and economic modeling and forecasting from real world data. See Koza (1992) for more details.

Sound localization

In research conducted by Karlsson, Nordin, and Nordalh (2000), genetic programming evolved a program that could classify the angle to the location of a sound relative to two input microphones placed inside a dummy head.

Programs were given raw digital audio as input, with one sample from each microphone per execution of the program. Two variables were also provided to store state information between invocations. Programs were then executed repeatedly over all the audio samples recorded for each fitness case. The solutions evolved were able to classify the location of sounds with varying success. Performance was good with saw-toothed waveforms presented at fixed distances from the microphones. Performance was much worse with a human voice presented at varying distances. For all the fitness cases, however, performance was better than random.

2.8.2 Software agents

Pac Man controller

This experiment, described in Genetic Programming (Koza, 1992), demonstrated that genetic programming could evolve an agent that took into account multiple sub-goals.

The domain was a full simulation of a traditional Pac Man game. The goal was to evolve a player that collected the most possible points in a given period of time. The sub-goals a successful player had to take into account include path finding, ghost avoidance, and ghost seeking after power up consumption.

In Koza's experiments, genetic programming was applied to high level functions, such as distance-to-ghost, distance-to-power-up, move-to-ghost, and move-from-ghost, etc. The Pac Man simulation was deterministic (no random numbers were used during the simulation), so the behavior of the evolved Pac Man probably was not very generalized. On the other hand, Koza was able to evolve relatively successful agents for this problem that pursued all three sub-goals, and opportunistically changed between them as the situation varied.

Multiagent cooperation

This experiment, conducted by Zhao and Wang (2000), demonstrated the ability of genetic programming to evolve multiagent cooperation without requiring communication.

The simulation involved two agents placed in a grid world, with fitness measured by their ability to exchange places. In-between them was a bottleneck through which only one agent could pass at a time. This restraint meant that the agents had to learn not to get in each other's way. Genetic programming successfully evolved a program that, when executed by both agents, allowed them to exchange locations in several different test worlds.

Emergent behavior: food collection

In this example, Koza (1992) showed that genetic programming could evolve programs that exhibit collectively fit behavior when executed in parallel by many identical agents.

In the food collection problem, the goal is to evolve agents that move spread-out bits of food into a single pile. Fitness was the summed distance of each bit of food from all other bits of food (lower=better). The instruction set consisted of pick-up-food, drop-food, move-randomly, and conditionals that checked if the agent was near food or holding food.

Using this starting point, evolution quickly found programs that, when executed in parallel, collected separate food pieces into larger and larger piles, until all the food was concentrated in one area.

2.8.3 Simulated and real robotics controllers

Wall following behavior in a continuous environment

In Genetic Programming (1992) Koza used evolution to create a program that controlled a simulated robot with wall following behavior. The fitness function selected for programs that found the edge of a room and then followed it all the way around its perimeter. The agent was given sensors that reported the distance to the wall at 8 equally spaced angles around the robot.

In Koza's work, evolved code was constrained into a subsumption themed structure. Specifically, code was inserted into a nested if statement three deep of the form (if statement: action; else if statement...). Each if roughly corresponded to a layer of subsumption. Programs were given access to all of the sensor values, a conditional instruction, and movement functions. Notably, there were no ways for the layers to operate in parallel, so the theoretical relationship to subsumption is limited.

Evolution was able to find a program that fit within the constraints of the code representation and satisfied the fitness function. Starting in a random location, the program would head toward the nearest wall, and then follow it all the way around the room.

RoboCup players

The RoboCup simulation league is a competition that pits teams of agents against each other in a simulated game of soccer. The practical use of these game playing agents may be limited, but it's an interesting and difficult problem that involves sensing an environment, reacting to that environment in real time, goal directed behavior, and interaction with highly dynamic groups of agents, some friend, some foe. In short, it's a rich problem that can gauge the progress of AI and the performance of contemporary agent theories and techniques.

For the 1997 competition, Luke (1997) used evolution to create a program that would competitively play soccer. Single programs were evolved, which were instantiated 11 times to control each player separately. Programs were tree-based, with a conditional instruction and soccer specific instructions like is-the-ball-near-me, return-vector-to-ball, and kick-ball-to-goal. Co-evolution was used to measure fitness by pitting two programs against each other at a time. Fitness was simply the number of goals scored by each team over several short fitness cases.

Using this framework, evolution was able to create a program that defeated two human-developed teams in the RoboCup 97 competition.

Walking controllers

In research conducted by Andersson, et al. (2000), genetic programming was shown to be able to evolve a controller that produces walking behavior for a real robot with four legs and eight degrees of freedom.

In this research, programs were evolved that were executed multiple times per fitness case. For each execution, a program outputs the angles that each motor should turn to. To maintain state between executions, a program is given the output angles calculated by the previous execution. Fitness was the distance the robot moved forward during the time the program controlled it. The instruction set only included basic mathematical operations.

Given this structure, genetic programming was able to evolve controller programs that moved the robot forward from the origin using a variety of different methods. These included chaotic paddling behavior, crawling, carefully balanced walking where three legs always maintained contact with the floor, and dynamic walking gates (like used by most animals when running) where balance is maintained via forward momentum. Frequently evolution progressed through several stages, starting with paddling behavior, and eventually finding balanced walking or dynamic gates.

Part II

PushGP

3 The Push language & PushGP

PushGP is a conventional genetic programming system that uses Push, a non-conventional programming language designed to be particularly interesting for evolutionary programming work. Both were designed by Spector (2001). Push was built to allow a wide range of programming styles (modularization, recursion, and self-modifying code) to arise from the syntax and instructions of the language, rather than from structure imposed from outside by the genetic programming engine.

The PushGP system was intentionally designed to use conventional GP algorithms and methods. The focus of this research is the advantages and overall impact of using Push. Whatever interesting results are found should be the result of the language, not experimental features of the genetic programming system that use it.

3.1 Overview of Push/PushGP - theoretical motivations

Typical genetic programming systems (such as Koza, 1994) add useful features like automatically defined functions (see section 2.6.1) by imposing structure externally. Since these features are specified externally, evolution cannot control their parameters just by modification of program code. If these parameters are to change during a run, additional genetic operators must be added that mutate features instead of code. While this separation between code modification and feature modification certainly adds more complexity to the GP system itself, it is an open question as to how evolution would change if the separation were not necessary. Push is a language that allows exploration of that question.

Push is a uniform-syntax, stack-based language designed specifically for evolutionary programming research. Instructions pop their arguments from the appropriate stacks and push their results back onto those stacks. The inclusion of different types of stacks (floating point, integer, code, etc.) allows multiple data types to be used in Push programs. If an instruction requires a certain type of data and that stack is empty, the instruction is just skipped entirely. Code operating on different data types, therefore, can be intermixed without any syntactical restrictions; any sequence of Push instructions is a valid Push program.

Most important of all, Push can move complete sequences of code (as well as individual instructions) onto the code stack and then execute them repeatedly, allowing for programs to create their own executable modules, as well as modify them on the fly.

3.2 Push as a tool of unconstrained solution discovery

Allowing evolution to determine the features of the search space makes possible, at least in theory, the discovery of new algorithms not preconceived by the researcher. Push is one way to allow evolution to dynamically control the search space in regard to the types of modularity and problem decomposition.

Highly specified run parameters constrain GP within likely solution spaces, but at the same time, limit solutions to those conceived of by the experimenter. Those limits are set because shrinking the search space tends to hasten the rate of evolution. For instance, if a problem decomposes most naturally into two sub-problems, providing only one ADF will not allow evolution to find that decomposition. On the other hand, if several more ADFs are included than necessary, the search space is increased, and the problem may even be over-decomposed.

As problems get harder and more interesting, it becomes more difficult to guess what parameters are most appropriate. When the solution to a problem is unknown beforehand, choosing the right parameters is even more problematical. If the researcher guesses the nature of the solution space incorrectly when setting parameters, evolution will be constrained from exploring the solutions that can solve the problem.

One way to address this issue is to manually try many different parameters, or build a meta-GP system to evolve the parameters between runs. Trying many different parameters, however, still relies on the intuitions of the researcher. A meta-GP system would also try many different parameters, but with the exploration of the parameter space controlled by evolution. The fitness of the particular parameter configuration would derived from the end-of-run fitness of the population, and the more successful runs would have their parameters spliced and mutated to form new parameters to use. Using evolution to determine parameters, however, drastically increases the number of runs necessary, since every fitness evaluation of a parameter configuration will require many runs of the underlying problem to get a reliable measure of its effect. Note that this idea has been implemented only for GAs at this time (Fogarty, 1989), but the process and issues would be exactly analogous for a meta-GP system.

PushGP minimizes the number of constant run parameters to a similar number as a standard GP system without ADFs. The Push language, however, provides for the same sort of flexibility for decomposing problems as ADFs do, along with the ability to create other forms of decomposition and program control. Plus Push lets evolution dynamically specify details like the number of functions, how many arguments they take, and mechanisms for iteration.

By leaving these settings up to evolution, purely through the modification of program code, evolution has potential to find truly novel control structures, on a level that simply is not possible with pre-specified ADFs or dynamically generated ADFs via genetic operators. Whether or not it does produce novel structures, of course, is an empirical question that can only be addressed by examining the actual program code it produces. This question will be addressed in the next few chapters.

3.3 The Push language

This section will describe the Push language with enough detail to allow comprehension of the evolved code produced by the research in the following chapters. Space, however, precludes a complete definition of the language itself. For more details on Push, the reader is encouraged to consult Spector (2001), which is reproduced in the appendix of this document.

As briefly described in section 3.1, Push is a uniform-syntax stack-based language. It provides multiple data types by including multiple stacks, one for integers, one for floats, and so on. Instructions pop their arguments from the stacks and then push their results back on. The stack used by each instruction is dynamically specified by the type stack, which also can be modified by program code. The use of a type stack hopefully provides some protection against destructive crossover, since spliced in code will continue to use the same type on the top of the stack as the code before it, until the spliced code explicitly changes the stack.

Push provides a code stack, where entire push programs can be stored and modified, just like any other stack, except that the code on it can be executed iteratively or recursively. The purpose behind this stack is to allow for programs to create their own modules and modify them on the fly. Human constructed Push programs have demonstrated that this is possible; one of the goals of this research is to see how much evolved programs take advantage of this.

3.3.1 Some simple Push programs

Push programs are represented as lists of instructions and symbols that manipulate stacks. For instance, the following push program adds 5 and 2:

```
(5 2 +)
```

The interpreter starts with the leftmost symbol, the **5**. Because it is an integer, it is pushed onto the integer stack. The same is done with the **2**. Now the integer stack holds the numbers **5** and **2**. Next the interpreter encounters the **+** function, which is defined as adding two numbers. Since there are two number stacks (the **float** stack and the **integer** stack), the **+** function consults the **type** stack, to find out which stack it should act on. When the **type** stack is empty, it has a constantly defined set of types that cannot be popped, which ensure that an instruction will have a type stack to consult. At the top of this stack is **integer**, so the **+** function pops the two top items from the **integer** stack, adds them together, and then pushes the result back onto the top of the **integer** stack.

A program that adds two floating-point numbers would look like this:

```
(5.1 4.3 float +)
```

In fact, it does not matter where the **float** instruction (which pushes the float type onto the type stack) is in the code, as long as it is before the **+** instruction. Both `(float 5.1 4.3 +)` and `(5.1 float 4.3 +)` produce the same output.

If an instruction consults the stack and does not find enough arguments to satisfy the function, then the instruction does nothing (i.e, it degenerates into the equivalent of the **noop** instruction). For instance, **(3 + 2)** pushes **3** and **2** onto the stack, but does not add them, since when **+** is executed, only **3** is on the stack.

3.3.2 Push stacks in detail

Every instruction (except **noop**) manipulates at least one stack. The following is a list of all the current types of Push stacks:

Integer	Whole numbers are stored on this stack. When the interpreter encounters an integer value in code (`5, 3, -5,` etc.), it is pushed here.
Float	Floating-point numbers are stored on this stack. When the interpreter encounters floating point value in code (`1.3, 7.0, -50.999`, etc,) it is pushed here.
Type	The type stack determines where a function that can deal with multiple data types retrieves its input. The type stack is not automatically popped by instructions that consult it, so code will continue to manipulate the same type until it is explicitly changed.
Code	Push instructions can themselves be pushed onto the code stack and other push instructions can selectively execute those instructions. The instructions that allow iteration, recursion and modularity all access code pushed on the code stack.
Boolean	The Boolean stack is the target of comparison functions (`=, <, nor`, etc) and input for conditional functions (`if`, etc).
Name	The name stack allows the storage of values and code for future reference without having to retrieve them from somewhere deep in the stack. Named values are stored and retrieved with the **get** and **set** functions, and the name (i.e. variable) being referenced by those instructions is whatever is at the top of the name stack.

3.3.3 Selective consideration of parenthesizes

Push programs are represented as nested lists of instructions, but are typically evaluated from left to right, ignoring those parentheses. For example, `(2 3 (+ 5 /))` is equivalent to `(2 3 + 5 /).`

Nesting comes into play with the quote function, which allows entire hierarchies of code to be pushed onto the code stack at once. For instance, executing `(quote 1 2 + )` pushes **1** onto the code stack, **2** onto the integer stack, and then the **+** does nothing since it only has one argument on the integer stack. The program **(quote (1 2 +) do*)** pushes an entire subprogram **(1 2 +)** on the stack and executes it with **do***:

The next example will show how this is used for something more useful.

3.3.4 A more advanced example: factorial

The following code (taken from Spector (2001)) pops the top of the integer stack and pushes the factorial of that value back onto the stack.

```
(code quote
      (quote (pop 1)
      quote (code dup integer dup 1 - do *)
      integer dup 2 < if)
      do))
```

This works as follows: **code** sets the type stack to **code**, and **quote** pushes all but the final **do** onto the code stack. Execution then advances to the final **do**. That final **do** causes the previously quoted code to be executed (but does not pop it off the stack):

```
(quote (pop 1)
      quote (code dup integer dup 1 - do *)
      integer dup 2 < if)
```

The two **quote**s then push **(pop 1)** and **(code dup integer dup 1 – do *)** onto the code stack. Execution continues with **integer dup 2 < if**, which **dup**licates the top of the integer stack, and compares the top value to 2, executing **(pop 1)** if it less, and **(code dup integer dup 1 – do** *) if it is more. Note that **dup** is used so that the current accumulating result is stored on the stack for future calculations, since **<** pops its arguments off the stack. The **(code dup integer dup 1 – do** *) part does the real work. It duplicates the top of the code stack:

```
 (quote (pop 1)
      quote (code dup integer dup 1 - do *)
      integer dup 2 < if)
```

subtracts 1 from the top of the integer stack, and then starts executing the top of the code stack again. Eventually, the top of the integer stack will be less than 2, and execution will pass to `(pop 1),` replacing the top of the integer stack with 1. Then execution will return from the recursive calls to **do** with a descending list of numbers on the integer stack, which are then all multiplied by the trailing ***** instructions after each **do** call. The final value remaining on the integer stack is the factorial of the integer that was pushed onto it beforehand.

3.3.5 Points & subtrees

Push programs are measured in points. A point is either a single instruction or an entire subtree. The length of a program is measured as the number of referable points in that program, which means that code nested in parentheses count as increasing the length of the program by one, even if the parentheses have no functional impact on the code. For example, the following push program measures three points in length: (+ 1). This is important to note in the case of genetic operators and push instructions, since these all operate on points. The three-point program, for instance, can be mutated at all three points, the first of which will replace the entire program with the new mutation.

3.3.6 The Push instruction set

This list is heavily based upon the Push interpreter's source code comments by Spector (2001). Each instruction references the type stack to determine the current stack to operate on, except for the exceptions noted.

Quote	Specifies that the next expression submitted for interpretation will instead be pushed literally onto the current stack.
Dup	Duplicates the top element of the current stack.
Pop	Pops the current stack.
Swap	Swaps the top two elements of the current stack.
Rep	Deletes the second item of the current stack. So called because in some contexts it is natural to think of this as REPlacing the second element with the first.
Convert	Pushes onto the stack of the most current concrete type a value converted from the item on top of the stack of the second most current concrete type (as determined from the type stack plus the type stack bottom.
Set	Stores a binding of the given type, binding the name on the top of the name stack to the item on top of the current stack.
Get	Pushes the binding of the given type for the name on top of the name stack onto the current stack. If there is no such binding then pushes the default value for the given type.
Pull	Pulls an item from an indexed position within the current stack and pushes it on top of the stack. This version is not for use with the integer type (which requires special handling because of the integer index argument).
Pullint	A version of &pull specialized for integers.
=	Pushes T onto the Boolean stack if the top two elements of the current stack are equalp.
+	Adds the top two elements of the current stack and pushes the result.
-	Subtracts the top two elements of the current stack and pushes the result.
*	Multiplies the top two elements of the current stack and pushes the result.
/	Divides the top two elements of the current stack and pushes the result. Division by zero produces zero.
/Int	A division instruction specialized for integers; always produced an integer result (via truncation).
<	Pushes T onto the Boolean stack if the second element of the current

	stack is less than the first element.
>	Pushes T onto the Boolean stack if the second element of the current stack is greater than the first element.
And	Pushes the result of a Boolean AND of the top two elements of the Boolean stack.
Or	Pushes the result of a Boolean OR of the top two elements of the Boolean stack.
Not	Pushes the result of a Boolean NOT of the top two elements of the Boolean stack.
Do	Recursively invokes the interpreter on the expression on top of the code stack. After evaluation the stack is popped; normally this pops the expression that was just evaluated, but if the expression itself manipulates the stack then this final pop may end up popping something else.
Do*	Like &do but pops the expression before evaluating it.
Map	Treats the item on top of the current stack as a list (coercing it to a list if necessary) and the second element as a body of code to apply iteratively to each element of the list. The arguments are both popped prior to the recursive evaluations. The results are collected and pushed as a list.
If	If the top element of the Boolean stack is true this recursively evaluates the second element of the code stack; otherwise it recursively evaluates the first element of the code stack. Either way both elements of the code stack (and the Boolean value upon which the decision was made) are popped.
Noop	Does nothing.
Car	Pushes the car (first item) of the top element of the current stack (which is coerced to a list if necessary).
Cdr	Pushes the cdr (all but the first item) of the top element of the current stack (which is coerced to a list if necessary).
Cons	Pushes the result of consing the second element of the current stack onto the first element (which is coerced to a list if necessary).
List	Pushes a list of the top two elements of the current stack.
Append	Pushes the result of appending the top two elements of the current stack, coercing them to lists if necessary.
Nth	Pushes the nth element of the expression on top of the current stack onto that stack (after popping the expression from which it was taken). The expression is coerced to a list if necessary. If the expression is NIL then the result is NIL. N is taken from the integer stack (which is popped) and is taken modulo the length of the expression into which it is indexing.
Nthcdr	Pushes the nth cdr of the expression on top of the current stack onto that stack (after popping the expression from which it was taken). The expression is coerced to a list if necessary. If the expression is NIL then the result is NIL. N is taken from the integer stack (which is popped) and is taken modulo the length of the expression into which it is

	indexing.
Member	Pushes T onto the Boolean stack if the second element of the current stack is a member of the first element (which is coerced to a list if necessary). The result is just a Boolean value (not the matching tail as with Common Lisp's MEMBER). Comparisons are made with equalp.
Position	Pushes onto the integer stack the position of the second element of the current stack within the first element (which is coerced to a list if necessary). Comparisons are made with equalp. Pushes -1 if no match is found.
Length	Pushes the length of the first element of the current stack onto the integer stack, coercing it to a list first if necessary.
Size	Pushes the number of points in the first element of the current stack onto the integer stack, coercing it to a list first if necessary.
Insert	Pushes the result of inserting the second element of the current stack into the first element. The index of the insertion point is taken from the integer stack, modulo the length of the expression into which the insertion is being made. Pops all arguments.
Extract	Pushes the subexpression of the top element of the current stack that is indexed by the integer on top of the integer stack.
Instructions	Pushes a list of the instructions that are implemented for the type on top of the type stack onto the current stack.
Container	Pushes onto the current stack the result of calling containing-subtree on the first and second elements of that stack.
Atom	Pushes T on the Boolean stack if the top element of the current stack is an atom; pushes NIL otherwise. Does not pop any stacks.
Null	Pushes T on the Boolean stack if the top element of the current stack is NIL; pushes NIL otherwise. Does not pop any stacks.
Subst	Pushes the result of substituting the third element of the current stack for the second item in the first item. All three of the arguments are popped. Comparisons for the substitution use equalp. There are several problematic possibilities; for example dotted-lists could result. If any of these problematic possibilities occurs the stack is left unchanged.
Contains	Pushes T on the Boolean stack if the first item on the current stack contains the second item.

3.4 The base PushGP system

PushGP implements a straightforward genetic programming system. Individuals are selected for reproduction via tournaments, and every generation an entire new population is generated via crossover, mutation, and perfect reproduction.

The following table summarizes the steps executed in a PushGP run and describes the algorithms used by each step.

Random individual generation	The initial population is generated via random tree growth starting at the root node, and extending to a depth randomly selected between 1 and the maximum-initial-program length (25 in all PushGP runs described in this document).
Tournament selection	Parents for input to the genetic operators are chosen via tournament selection as described in the GP overview section 2.5.3. For all the PushGP experiments described in this document, the tournament size was 5 individuals.
Mutation	The mutation operator takes one parent and randomly replaces one point of the program (either a tree or a single instruction) with a tree of a randomly generated depth of up to 10 points. If the tree is of depth 1, mutation does not bother to encode it as a tree and just inserts it as a single element. In the case that mutation produces a child larger than the maximum program length, mutation returns the original parent unmodified (i.e., it degenerates into perfect duplication).
Crossover	The crossover operator takes two parents and replaces a random point in one parent (instruction or tree) with a random point (instruction or tree) from the other parent. In the case that crossover produces a child larger than the maximum program length, crossover returns one of its original parents unmodified (i.e., it degenerates into perfect duplication).
Generational population	Every generation an entire new population is created via crossover, mutations, and duplication. For the most fit individual to survive between one generation and the next, it must be selected by the tournament and have the duplication operator applied to it (or receive a nondestructive application of mutation or crossover). While this might seem likely to lose track of fit individuals, rarely in practice does the best member of the population not reproduce between generations.

4 PushGP compared to GP2 with ADFs

In this chapter PushGP is applied to a symbolic regression problem and several even-parity classification problems. These same problems were used by Koza in Genetic Programming II (1994) to show that adding ADFs improved the performance of his genetic programming system. Koza selected these problems because they exhibited regularities that predisposed them to be more easily solved by modular programs.

He argued that ADFs improve performance precisely because they successfully allowed genetic programming to evolve modular solutions. For each problem, he showed that many of the programs evolved did use ADFs to reuse code and find the solution by decomposing it into smaller problems. Furthermore, he argued, based on the results of experiments with different sized even-parity problems, that the decomposition allowed by ADFs lets genetic programming scale to larger problems more efficiently than without ADFs.

The motivation for comparing PushGP to Koza's work is to empirically test the ability of PushGP to also evolve modular solutions to problems known to benefit from modularity. In the case that modular solutions do evolve, the secondary issue is to examine how PushGP creates modularity.

The goal is not to show that one system is better than other, since, by the No Free Lunch theorem (see section 2.7) both will have strengths and weaknesses, but rather to explore the modularity abilities of PushGP in comparison to a well-known set of prior experiments.

4.1 Can a more flexible system perform as well?

Genetic programming is analogous to a heuristic search algorithm that traverses the space of computer programs by producing successors via mutation and crossover. In general, the larger the search space (that is, the more functions, constants, and variables made available), the slower the search (Koza, 1992). The theory behind PushGP is that while the number of instructions necessary for its base level operation is higher than in traditional GP, the added search space allows the evolution of modularity without explicitly stating the nature of that modularity in the run parameters.

If this theory is correct, then PushGP may allow the exploration of problem spaces with less prior knowledge of the solution space (such as the type of modularity necessary for efficient evolutionary progress).

In this chapter, initial research on PushGP's ability to evolve modularity is described. In addition, the computational effort required on these problems is compared to Koza's ADF work, for a rough comparison of how much more effort these simple problems require with larger search spaces. The expectation is that for harder problems, the flexible modularity of Push will turn out to compare well with GP with ADFs, but not for the simpler problems.

4.2 The computational effort metric

In the development of genetic programming systems, it is useful to have a metric for evaluating the effectiveness of a given set of parameters and algorithms. By the No Free Lunch theorem, a metric can only measure the effectiveness of a single configuration of genetic programming system on a specific problem. Just changing the problem or the parameters of a run can have a significant effect on how well a system performs. Nonetheless, the performance on a specific problem can be used to roughly predict the performance on similar problems, as long as the features of the problems are the same.

One metric that is widely used to analyze genetic programming systems is the *computational effort* required to solve a problem. It is an empirical estimate of the minimal number of individuals for a given population size that must be processed in order have a high probably of finding a successful program. In another words, it estimates the number of runs, and the maximum number of generations processed per each run, to have a very high probability of finding a solution.

This metric does not consider wall clock time explicitly. In most genetic programming runs, however, the significant speed bottleneck is the fitness evaluation, so the number of times it must be applied (which is equal to the number individuals processed) is a good indicator of the wall clock performance of a system.

4.2.1 Mathematical definition of computational effort

See Koza (1992, page 191) for a formal development of computational effort.

Computational effort is calculated in two steps that analyze the results of a number of genetic programming runs which are identical except for differing initial random seeds. The more runs used in the calculation, the more exact the estimation of the true computational effort.

First calculated is the probable likelihood that any given run will have found a solution by generation N (0…maximum generations). This value will increase with generations (the steeper the increase, the better the system is functioning).

Then, for each generation N, it is calculated from the probability of success across all runs how many separate runs must be executed up to generation N to find at least one successful individual with high likelihood (the convention is 99% probability).

The following pseudo code calculates the computational effort for a given maximum number of generations.

```
ComputationalEffort(PopulationSize, MaximumGeneration,
      DesiredProbabiltyOfSuccess) =
PopulationSize * (MaximumGeneration +1) *
RoundedUpToTheNearestInterger(
      Log(1- DesiredProbabiltyOfSuccess) /
      Log(1-ProbabiltyOfSuccessPerRun(PopulationSize,
            MaximumGeneration))
```

The computational effort reported is simply the lowest effort found (number of individuals processed) while varying the MaximumGeneration from 0 to the highest generation where a successful individual was found. Note that this means the result is not necessarily representative of the lowest computational effort possible, but rather an estimate of what the lowest effort would have been within the range of generations completed.

When computational effort is reported in this document, two values are given. One is the estimated minimum number of individuals that must be processed to solve the problem with 99% likelihood. The other is the maximum number of generations each independent run should be executed in order to minimize computational effort. This is the number of generations after which continuing the run no longer increases the likelihood of finding a solution, as opposed to restarting it run from scratch. These two numbers will be labeled as the computational effort *x* at generation *y*.

Common Lisp code that calculates computational effort is included in Appendix A.

4.3 Measuring modularity

PushGP is a new system and it is currently unknown what sort of performance it has on problems that involve modularity, and whether it will evolve modular solutions to those problems. Furthermore, while Push clearly can represent modular programs (see the factorial example in section 3.3.4), it is not yet clear what instructions are necessary / useful for allowing evolution to find modular solutions.

The experiments that follow will address these issues with the following two questions:

- Does the amount of computational effort required compare favorably to Koza's work (with or without ADFs) in terms of absolute performance, and in terms of scaling as the problems become harder?

- Are the solutions evolved by PushGP modular?

While the first question is interesting in terms of practical use of PushGP as it exists today, these initial experiments will not be broad enough to address it fully, since time constraints prohibit running PushGP over a large range of domains and problem scales.

The second question, however, lies at the root of what is theorized to be interesting about PushGP. If it does evolve modularity, it will be particularly interesting to see what sort of modularity. If PushGP does not evolve modularity for at least some of these problems, however, then it will bring into question its ability to evolve modularity at all.

Measuring modularity is difficult, however, especially without constructs like ADFs that explicitly mark where new modules start and end. Even when instances of modularity *are* easy to recognize, it is not clear how to reduce the analysis of modularity into a mathematical equation yielding a single value that describes how modular a program is.

Instead, modularity will be qualified in terms of the core idea that modularity involves program code that is used more than once. In PushGP this can be recognized in two ways:

- A given instruction (or sequence of instructions) that appears N times in a program is executed at least N+1 times. This measure varies along two dimensions: the number of times each sequence is executed, and how many such sequences there are in a program.

- The result of a calculation is stored more than once (typically via the **dup** command), and then made use of in entirely separate pieces of code. This does not count if the dup command is only used so that a value remains on the stack so it can be used by the next instruction.

These measurements of modularity can only reasonably be applied to simplified Push programs. The goal of the analysis to see if programs use modularity to solve a problem, not to see if modularity can be used in a novel way to produce **noop**s.

4.4 Solving symbolic regression of $x^6 - 2x^4 + x^2$

Symbolic regression, as described in section 2.4.1, is the problem of finding a function that describes a target output curve over a collection of input values. Symbolic regression is one of the easiest types of problems to encode for GP, and is often used in the benchmarking of a new system. This particular problem was chosen because it allows comparisons with rigorous computational effort values from Koza's research with it in Genetic Programming II.

4.4.1 Koza's results with and without ADFs

In Genetic Programming II, Koza (1994) demonstrated that his genetic programming system could solve the symbolic regression problem of $x^6 - 2x^4 + x^2$. Furthermore, he showed that, for his system, the addition of ADFs resulted in lower computational effort to solve the problem. Without ADFs, the computational effort was 1,440,000 at generation 39. With ADFs, computational effort was 1,176,000 at generation 48.

4.4.2 Using PushGP with a minimal instruction set

This run was designed to make as close a comparison to Koza's non-ADF results as possible. All parameters were kept the same as those used in Koza (1994), except for those necessarily different because of the use of Push. Those changes are an instruction set that adds the DUP instruction (to allow a calculation to be used as input to more than one mathematical operation) and that the input value of X is pushed onto the stack, rather than as a terminal that can be inserted anywhere in the program tree.

Parameters for the symbolic regression of $x^6 - 2x^4 + x^2$

Fitness Cases	50 random input values for x over the interval –1.0, +1.0
Population Size	4000
Maximum Generations	51
Maximum Program Length	50 points
Instruction Set	+ - * / dup ephemeral-random-float
Crossover, Mutation, Duplication	90%, 0%, 10%
Input	The value of x for this fitness case, pushed onto the integer stack.
Fitness Measure	The sum of the error between the correct output for each fitness cases and the actual output
Termination	When the error *for each fitness case* is less than 0.01

Results - Effort

Out of 51 independent runs, seven successfully found the target function within the maximum error rate allowed. The computational effort required to solve this problem was estimated to be 1,864,000 individuals processed, if each run is executed up until generation 1. The computational effort on this problem is very similar to In Koza's non-ADF result of 1,440,000 individuals.

The low number of generations for which it is worth running this trial is surprising. It is caused by the result of one run. In that run a successful individual was found after only one round of mutation and crossover was applied to the initial random population. This is not an isolated result. In other trial runs on this problem another individual was found that perfectly fit the data after only 1 generation.

This indicates that the combination of instructions used on this problem and the nature of initial random population can give rise to individuals very close to the solution for this particular symbolic regression problem. It is worth noting that the most efficient maximum number of generations to execute, discarding the generation 1 success, is 44 (with a corresponding computational effort of 6,660,000). This suggests that if the population does not start out in a fertile point in the fitness landscape, this problem becomes much harder for PushGP to solve. In Koza's results, one run found a solution only after 5 generations, so it seems that his GP system was also able to produce random individuals near the solution to this problem.

Results – Individuals

In the run that evolved a solution by generation 1, the following was the Push program it created:

```
(DUP (DUP DUP * (*) - /) DUP *)
```

The parentheses are a byproduct of the genetic operators used by PushGP. They are non-functional, since this program includes no code manipulation instructions. Removing them, the program becomes:

```
(DUP DUP DUP * * - / DUP *)
```

This can still be simplified more, however, since the / instruction does nothing because it requires two items on the stack as input, and when it's executed only one item is on the stack. Removing the /, the final, simplified, but functionally identical program is:

```
(DUP DUP DUP * * - DUP *)
```

Though ephemeral random constants were included in this run, this program does not make use of them (probably part of the reason why it converged so quickly). It works instead by finding a formula that includes no coefficients. The first three **dup**s add duplicates of the top of the stack back onto the stack, leaving four total copies of X on the float stack. Then the two multiplication operators, * *, cube X, pushing it onto the stack, and leave one copy of X unmodified. The **-** then subtracts the cube of X from X, and the final **dup *** pair squares that result. The mathematical formula represented is $(x - x^3)^2$, which is equivalent algebraically to $x^6 – 2x^4 + x^2$. While there is no code modularity here, in that each instruction is executed only once, it is fair to say that PushGP found a way to break the problem into mathematically modular parts via the stack, making the problem easier to solve.

The other solutions evolved for this program are much more messy, but generally contain the same core program. For instance, in one run, the following solution was found

```
(* DUP DUP DUP * * - DUP * * * 0.061086297035217285 DUP DUP *
0.3783543109893799 + * 0.061086297035217285 DUP * POP POP + DUP
0.006923556327819824 0.061086297035217285 / -0.12430727481842041
-0.49549615383148193 * * DUP + 0.006923556327819824 0.0646294355392456 *
* * -)
```

This program is unsimplified except for the removal of extraneous parentheses. It begins with an ignored * operator (because only one number is on the stack), and then functionally duplicates the generation-one program discussed above. Thereafter, the entire program is devoted to performing a selection of instructions that mathematically transform the output, and then mostly reverse those transformations by the end. Simplifying this program to its core is tricky, since removing one instruction at a time from the second half of the program destroys its symmetry and thereby obscures the correct answer generated by the first half of the program. Without knowing that the first few instructions do the actual calculation, it is not easy to simplify this program just by inspection.

Conclusion

For this program, PushGP (with a solely mathematical instruction set) compared favorably with the computational effort found by Koza in his work without ADFs. Since no code manipulating instructions were included, no modularity of code execution was possible, however the solution did exhibit mathematical modularity via clever use of the stack.

4.4.3 PushGP with a full instruction set

In this trial the question was how well PushGP would perform on the same problem with a drastically larger instruction set. One of the goals of PushGP is to allow less constrained exploration of a problem's search space. If the solution to a problem is not known beforehand, an experimenter's selective inclusion of instructions may constrain the search space to non-fertile areas. The performance on this trial addresses what would happen if the solution to this problem were not known and the entire instruction set was included.

Customizing the function set, when well informed about the exact solution of a problem, usually improves fitness considerably. Since the instruction set used in 4.4.2 was based upon the known solution to this problem, the performance in 4.4.2 should represent the best performance PushGP can achieve on this problem. PushGP, therefore, should perform relatively poorly on this trial; the question is, how much worse?

Parameters for the symbolic regression of $x^6 – 2x^4 + x^2$

Fitness Cases	50 random input values for x over the interval –1.0, +1.0
Population Size	4000
Maximum Generations	51
Maximum Program Length	50 points
Instruction Set	Full PushGP function set (44 instructions and 5 type specifiers) float, Boolean, and symbol ephemeral random constants.
Crossover, Mutation, Duplication	90%, 0%, 10%
Input	The value of x for the current fitness case, pushed onto the integer stack.
Fitness Measure	The sum of the error between the correct output for each fitness cases and the actual output
Termination	When the error for each fitness case is less than 0.01

To keep the problem as close to Koza's encoding as possible, all the parameters duplicate those of his runs, except for the instruction set, which, of course, is necessarily PushGP specific.

Results

Out of 50 runs, no successful individual was found. Furthermore, the fitness of the population did not improve much over the best of generation 0 individual. Typical best of generation 0 individuals have a fitness of 2.35, whereas the end of run individuals had an average fitness of 2.1. The very best individual found in all 50 runs only had a fitness of 1.8, which is comparatively not that much better than the average fitness. The amount of improvement seen in these runs over 50 generations suggests that, as configured, PushGP will take a very long time to find a single successful individual.

Conclusion

This problem was hard for PushGP to solve using a very carefully tuned instruction set in 4.2.2 and it appears almost impossible to solve as configured in this trial. It is surprising that increasing the function set so drastically impeded performance on this problem.

One possible reason for the poor performance on this trial was that no mutation was allowed. This was done to keep the PushGP run parameters maximally similar to those used in 4.2.2 and in Koza's work. Since the question of how PushGP performs with maximally similar settings has now been answered, there is no reason to continue using the parameters.

In most genetic programming research, mutation is included to maintain genetic diversity in the population. Diversity helps keep populations from becoming stuck in local minima. Judging from the small increase in performance on this problem over 50 generations, each run did, in fact, get stuck in local minima very quickly. Allowing some mutation on this problem, therefore, might well allow successful individuals to be found, even with a full instruction set.

This issue will be examined in more depth in future research, when time and available computing cycles permits running this problem with different configurations.

4.5 Even parity as a GP benchmark

Even parity is a Boolean function of N arguments, which answers true if an even number of the arguments are true (with 0 included in the set of even numbers). Odd parity, incidentally, is the inverse Boolean function of even parity.

Even parity is often used as a simple tool for detecting data corruption. A parity bit is added to the word size of the data stream, and its value is set so that the word + parity bit satisfies the parity function. If any one bit in that word is flipped thereafter, the parity of the word + parity bit will no longer be even, marking the data in that word as corrupted.

The parity problem is defined as learning to correctly classify the parity of bit-patterns of N length, given a parity truth table for all 2^n bit strings.

Koza (GP II) selected even parity problems as a benchmark for genetic programming with and without ADFs for several reasons:

- Parity problems are easily scaled in difficulty just by changing the arity (the size of N). Furthermore, as the problem scales in difficulty the fundamental nature of the problem is still the same. There is less reason, therefore, to think that differences in the performance at higher arity parity are the results of odd changes in the representation or dynamics at that particular arity.

- Parity problems are difficult for machine learning systems in general, not just genetic programming.
- Parity problems can be solved via modularization, and the modularization of a parity problem makes it much easier to solve from a human standpoint. Typically, parity solutions of high arity are calculated with the aid of lower order parity functions. If ADFs provide an effective method for evolving modularity, then their computational advantage should be measurable on the parity problem.

It is the last point that makes parity particularly interesting as a test of the modularity of PushGP. If PushGP can evolve modularity, then it too might show an advantage when the instructions that allow modularity are enabled.

4.5.1 Koza's work on even-four-parity

In the even-four-parity problem there are 16 fitness cases, consisting of each four possible combinations of Boolean true and Boolean false. Fitness is the sum of the number of incorrectly classified fitness cases. The success criteria is for this problem is 0 fitness.

In Genetic Programming II (Koza, 1994), Koza showed that his genetic programming system could find solutions for the even-four-parity problem both with and without ADFs. He used relatively straightforward encoding for the problem: the terminals were each of the four Boolean inputs and the functions were AND, OR, NAND, and NOR.

The only remarkable parameter is a population size of 16,000. This size was chosen so that computational effort could be compared with the same parameters across different difficulty parity problems, including some where a large population was required to find a solution. Even-four-parity without ADFs had a computational effort of 384,000 at generation 23. With ADFs, performance improved to 176,000 at generation of 10.

4.6 Solving even-four-parity using PushGP and stack input

The most straightforward encoding of the parity problem for PushGP was to provide the four Boolean values of each fitness case by pushing them onto the Boolean stack, and then, after executing the push program, using the top item on the Boolean stack as the answer.

4.6.1 Full PushGP instruction set

This section addresses the question of how well PushGP can evolve a solution to the even-four parity problem given the unconstrained search space of the entire Push language.

Because of the drastic difference in this function set, no attempt was made to keep the run parameters maximally similar to those used by Koza. This means that a smaller population size was used and both crossover and mutation genetic operators were enabled.

Parameters

Fitness Cases	16 total possible combinations of 4 Boolean arguments
Population Size	5000
Maximum Generations	100
Maximum Program Length	50 points
Instruction Set	Full PushGP function set (44 instructions and 5 type specifiers) and integer, Boolean and symbol ephemeral random constants
Crossover, Mutation, Duplication	45%, 45%, 10%
Input	Four Boolean values pushed onto the Boolean stack
Fitness Measure	The sum of the number of incorrectly classified fitness cases
Termination	Fitness = 0

Results – Computational Effort

Out of 30 runs, 21 terminated with a successful individual. The computational effort was 945,000 at 26 generations. This result clearly illustrates that PushGP can solve the problem in this form, without *any* tweaking at all. Though there are many differences between these parameters and Koza's (most notably smaller population size and larger function set), it is still notable that PushGP's very unconstrained search resulted in only about two and a half times more computational effort than Koza's GP without ADFs (E=384,000) and just over six times GP with ADFs (E=176,0000).

If PushGP were configured more similarly to GP with ADFs (most notably with a larger population and more tuned function set), it would probably perform better. Absolute computational effort is of less interest, however, than how performance scales on real problems. That question will be addressed when a more difficult even parity problem is run.

The next section will address what sort of modularity PushGP evolved for this problem.

Results - Individuals

The programs evolved were long and appeared complex. Upon simplification, however, each one analyzed was elegant, short, and performed the same calculations using the same methodology.

In one run, the following, perfectly fit, program evolved by generation 26:

```
(DO* SWAP ((APPEND)) SET ((GET EXTRACT CONVERT T) EXTRACT (MEMBER (AND
(CAR RAND CAR) (CONS ((CONVERT APPEND =) IF) (CONVERT APPEND =)) ((/)
(CONS) (INSERT) SUBST MAP)) DUP) NTHCDR))
```

This very complex appearing solution is too difficult to follow in its current form, like many solutions evolved by genetic programming. This complexity, however, is mostly the product of dead code. Once that code is removed the analysis becomes much easier.

Finding a systematic method for removing dead code, however, is tricky, since dead code can interact with other dead code that cancels out its effects. Two methods have been used in this research. One is to iteratively flatten points, remove points one and two at a time, and replace instructions with **noop**s, backtracking whenever the fitness of the program decreases.

The other is to execute a genetic programming run upon a small population initialized with the unsimplified program. Evolution selects for program fitness primarily and program length secondarily. By only using genetic operators that convert points into **noop**s, flatten points, and delete points, functional code is maintained while the code that does not contribute to fitness is removed. Using a GP algorithm instead of hill climbing allows multiple operators to transform a program before it is discarded because it is no longer functional.

Both systems are effective in finding the simplified version of a given program. The second one executes slower, but has the potential to simplify more complex programs than the first. Whichever one is used, however, the simplified programs are the product of removing dead code and eliminating structure that does not effect program execution.

After simplification of the program above, the following, equivalent individual is found:

```
(DO* CONVERT CONVERT = CONVERT =)
```

This curious program appears to include no Boolean functions (the = instruction is non-functional for Boolean arguments in the implementation of Push used in this document). Careful inspection of this program's inner workings, however, shows that it has found ways to do higher-level Boolean logic (not-xor, specifically, which is also known as equality) using non-Boolean functions.

This program calculates even parity by checking the equality of the first two inputs, then the second Boolean inputs, and outputs true (i.e., classifying the data as even-parity) if those two equalities are also equal.

This works because if *both* pairs of inputs are equal, then each satisfies even-two-parity, which means the whole set of inputs is also even (even + even = even). On the other hand, if *neither* two pairs of inputs satisfy even-two-parity, they must both satisfy odd-two-parity (and odd + odd = even). Only when one set of inputs satisfies even-two-parity and another odd-two-parity does the whole set not satisfy even-four-parity.

The following table shows the order of execution of instructions and describes exactly how the above calculation is carried out.

Ref #	Instruction	Description
Startup		Before execution begins, the Push program is placed on the code stack. The Push interpreter does this so the program has access to its own code. The interpreter keeps a complete copy of the program passed to it, however, and that is what is executed, not the copy on the code stack. The type stack starts empty, so when queried, it instead returns default values, with integer at the top, and Boolean below it. Until other types are pushed on the stack, all instructions that consult the type stack will use these default values. Instructions that take one datatype will operate on the integer stack, and all that take two datatypes (namely, convert) will operate on the integer and Boolean stacks.
1.	DO*	The initial **do*** pops & executes the entire contents of the code stack, which is an exact duplicate of the push program already being executed by the push interpreter. This has the effect of executing the program twice, since after the processing initiated by **do*** is complete, execution by the top-level push interpreter continues on the code after the do. In short, including a **do*** in a program with no other code stack manipulations, is analogous to telling Push to "run the program twice". This is the source of modularity in this program.
2.	DO*	Byproduct of initial **do*.** Since the last **do*** already popped the entire code stack, there is no code to execute, so this instruction degenerates into a **noop**.
3.	CONVERT	Both **convert** instructions pop the top of the Boolean stack and push the equivalent value onto the integer stack (nil = 0, t = 1). The Boolean stack ends up smaller by two, and the integer stack ends up larger by two.
4.	CONVERT	
5.	=	Tests the equality of the two top items of the integer stack and pushes the result onto the Boolean stack. The end result of # 3,4,5 is the calculation of the equality of the top two items on the Boolean stack. Push does not include a Boolean equivalence operator, but PushGP found a short way to perform that calculation.
6.	CONVERT	Converts the Boolean equivalence calculated in #5 onto the integer stack.
7.	=	Since the integer stack has only one element, and equality requires two inputs, this degenerates into a **noop**. The **=** instruction is executed here as a byproduct of the duplicate

		code execution caused by the **do*** in #1. The next time this = is encountered there will be enough arguments on the stack.
8.	CONVERT	Performs the same calculation (equivalence) as in #3,4,5 for the second two Boolean arguments on the stack, and stores the result on the Boolean stack. Note that the value pushed on the stack by #6 is not popped by the #8,9,10 sequence, and still remains for use in #12.
9.	CONVERT	
10.	=	
11.	CONVERT	Converts the value from #10 onto the integer stack. Now both calculated equivalences are on the integer stack (#8,9, and10leave untouched the item pushed on by #6)
12.	=	Pushes onto the Boolean stack the equivalence of the two equivalences that have been converted onto the integer stack.

It is notable that the above program, after removing the **noop**-equivalent instructions and **do*** structure modifying instruction, is exactly equivalent to the following 11 point Push program:

```
(CONVERT CONVERT CONVERT CONVERT = = CONVERT CONVERT =)
```

Oddly enough, while there was no selective pressure towards shorter programs (and indeed the evolution always produced programs longer than 11 points), no program evolved this non-modular, but syntactically-simpler version. In fact, all solutions evolved included the **do*** trick for doubling the execution of their evolved code. The following table shows a sample of the first 5 the solutions PushGP produced for this problem. All are simplified for readability.

Solutions to Even-Four-Parity:

`(CONVERT DO* CONVERT = CONVERT =)`
`(CONVERT DO* CONVERT = CONVERT =)`
`(DO* CONVERT = CONVERT CONVERT =)`
`(DO* CONVERT CONVERT = CONVERT =)`
`(DO* CONVERT CONVERT CONVERT CONVERT = =)`

As can been seen just by inspection, all solutions are highly similar in nature. All rely on converting Boolean values into integer values so that the = comparison can be used as a not-xor operator.

The first two solutions are exactly the same and differ from several others only in the location of the **do***. Since **do*** causes the program to be executed again (but just once, since it pops the code stack), it can be placed in multiple locations, as long as the right balance of **convert** and = instructions is maintained.

Results - Summary

PushGP performed reasonably well on this problem, though not as well as traditional GP. The comparison is not equal, however, because default PushGP settings were used (including the entire Push instruction set). If Push had been more carefully tuned to the problem, the computational effort probably would have been lower. On the other hand, an initial attempt to concisely encode the problem probably would not have included both the **convert** and **=** instructions, so human tweaking of the parameters might actually have hurt performance. This illustrates one of the theories behind PushGP: a system that does not require a significant amount of pre-knowledge about the problem can evolve novel solutions that would be missed otherwise.

It is also notable that, for this problem, all evolved solutions were modular by the metric that code existing only once in the program definition was executed more than once. PushGP found the modularity inherent in the problem and exploited it. Though the performance was not as good as Koza's results with ADFs for this problem, it appears that code manipulation did play an important part in allowing PushGP to find the solution to this problem. Given more time, it would be interesting to see how performance would change if the **Do*** instruction were removed and the problem run again. Clearly, it would make impossible the solution exclusively favored by evolution in this trial, and it would probably lower performance. It is possible, however, that Push would find another way to modularly solve this problem without too much additional effort.

4.6.2 Minimal function set with list manipulation

The previous section showed that PushGP is flexible enough to find solutions to the even-four-parity problem, without any tuning to the problem, and still do so with a reasonable computational effort.

This section addresses the computational efficiency of PushGP when the instruction set is drastically reduced so that no modularity is possible. In this experiment, the instruction set was limited to list manipulation, minimal stack manipulation, and Boolean operators. List manipulation instructions were provided so that PushGP would be able to store results of Boolean calculations and access them out of order.

Parameters, even-four-parity with list manipulation

Fitness Cases	16 total possible permutations of 4 Boolean arguments
Population Size	4000
Maximum Generations	100
Maximum Program Length	50 points
Instruction Set	List car cdr dup noop pop and or nand nor

Crossover, Mutation, Duplication	45%, 45%, 10%
Input	Four Boolean values pushed onto the Boolean stack
Fitness Measure	The sum of the number of incorrectly classified fitness cases
Termination	Fitness = 0

In Koza's experiments, the population size was 16,000, which would surely lead to different computational effort values. A smaller value was used in these runs, however, to facilitate efficient use of memory on the available hardware. Any comparison in computational effort, therefore, will have to address this issue.

Results

After 10 runs, this trial was aborted, because no individuals evolved by run termination (generation 100) achieved better performance than the best member of initial-random population. All scored a fitness of 7, corresponding to slightly better performance than guessing all true or all false. Furthermore, in a separate test, an additional 400,000 individuals were generated at random using these parameters, and none had a fitness better than 7. Without some positive variation in the landscape, there is no real way for fitness to improve with generations beyond a blind, random search.

Since evolution was unable to improve over blind search, it seems clear that the problem as encoded in this trial is unsolvable, or at least very hard to solve. Instead of collecting exhaustive data with more runs, this trial was aborted in favor investing computational cycles in more promising areas.

4.6.3 Minimal function set with rich list manipulation

The previous trial suggested that the parameters used could not solve the problem. In this trial, a similar instruction set was used, but with all the rest of the list manipulation instructions added. All other parameters are left unchanged.

Parameters, even-four-parity with rich list manipulation

Fitness Cases	16 total possible permutations of 4 Boolean arguments
Population Size	4000
Maximum Generations	100
Maximum Program Length	50 points

Instruction Set	*Rep* *pull* *cons* *append* list car cdr dup noop pop and or nand nor
Crossover, Mutation, Duplication	45%, 45%, 10%
Input	Four Boolean values pushed onto the Boolean stack
Fitness Measure	The sum of the number of incorrectly classified fitness cases
Termination	Fitness = 0

Note: changes between this trial and the last are shown in italics.

Results

Out of 41 complete runs, no individual was found with a fitness higher than 7, the same fitness found by all runs in the initial random population. This suggests very strongly that neither this instruction set, nor that employed in the previous trial, is sufficient to allow evolution to progress beyond the fitness of the initial random population.

The empirical conclusion, then, is that a stack-based programming language is seriously limited for this kind of task unless given the ability to build higher-level constructs. By inspection, this seems to be a very hard solution to reach via stack manipulation. Without any sort of modularity, processing would have to progress by performing most of the Boolean functions required upon the inputs on the top of the stack, before even having access to the bottom of the stack, where the other inputs are located.

The list manipulation instructions were provided to allow a possible outlet to this problem, with the idea that intermediate values might be stored in lists. It does not appear, however, that the beginnings of such a solution was findable via blind random code mutation. Perhaps if the fitness metric were not binary (i.e., it measured more than correct or incorrect on a given fitness test), this method might work. On the other hand, for a Boolean function, it is not obvious how the fitness measurement could be more fine-grained than reporting either success or failure for each fitness case.

Conclusion

It seems unlikely, without some other drastic change, that this function set will be successful by adding more runs, since fitness has yet to climb out of the initial local minimum in all the tests run. For that reason, this method of inputting values was abandoned, and future trials experimented with an alternative method of Boolean value input that was more similar to what Koza used successfully in his work.

Another direction, currently unexamined, is to run this set of parameters, but add some functions that allow modularity, or enable ephemeral random symbols so that Boolean values could be stored in variables. Since neither of these approaches is similar to Koza's known successful approach to this problem, such experiments are left for future work.

4.7 Even-four-parity with input functions

Examination of the code evolved by Koza for the even-four-parity problem (with and without ADFs) in Genetic Programming II suggest that very frequent, non-sequential access to the four inputs is necessary for the solving of the problem. At the very least, the solutions Koza evolved all exhibited this behavior.

In Koza's work, the inputs were made available as terminals – variables that could be inserted directly into the tree at the point where their value is needed. In this section, the same access is given to PushGP via the addition of zero argument functions that push the Boolean value of their respective input onto the stack.

4.7.1 Minimal PushGP instruction set and max program length 50

Parameters, even-four-parity, with input instructions

Fitness Cases	16 total possible permutations of 4 Boolean arguments
Population Size	4000
Maximum Generations	100
Maximum Program Length	50 points
Instruction Set	T1 T2 T3 T4 dup noop pop and or nand nor
Crossover, Mutation, Duplication	45%, 45%, 10%

Input	Functions T1…T4 push their respective Boolean values onto the stack.
Fitness Measure	The sum of the number of incorrectly classified fitness cases
Termination	Fitness (total number of errors) = 0

Results

Out of 40 independent, runs no successful program was found. Fitness at run termination, however, did improve over that of the best of first generation fitness (7). The best program scored an error of 3, and the average error at run termination was 4. It was hypothesized that the lack of success was due to the program length limit of 50 instructions, and so this run was aborted in favor of a new parameter set.

4.7.2 With minimal instruction set and max program length 100

In this trial the only parameter changed was the maximum program length, which was increased to 100.

Results

Out of 15 independent runs, no successful solution was found. Fitness at run termination, however, did improve over the last trial. The best program scored an error of 2, and the average error at run termination was 3. It was again hypothesized that the lack of success was due to the program length limit, and so this run was aborted in favor of a new parameter set.

4.7.3 Minimal PushGP instruction set and max program length 200

In this trial the only parameter changed was the maximum program length, which was increased to 200.

Results

Out of 30 independent runs, two successful solutions were found. With such low success rate, the computational effort cannot be calculated with any reasonable certainty, but it can be roughly estimated to be 62,000,000 at generation 57.

One of the successful solutions emerged at generation 62. Unsimplified, it consisted of the following 168 points of Push code:

```
(((T3 T2) POP) (POP) (POP (AND T4 ((NAND T2 ((NOR T4) NOR)) NOR)) ((OR))
(OR (T4)) (((OR T3) (NOOP AND (NAND OR T4)) (T2 ((T1 AND) NAND)) ((DUP
(OR OR T1 T2) (OR T3 T2)) ((((T1 (NOOP DUP)) (NOOP (DUP DUP (T4)) (POP
NOR)) (NAND) POP T2 (((T4) (OR) NAND) AND NAND))) ((T4 (NOOP) OR) (T3)
T2) ((T1 ((OR T2 T1) NAND AND)) ((NAND) ((T3 T2) OR) AND)) NOOP) T4
(NAND (NAND))) ((T1 (T2)) T4 ((OR) DUP) NAND)) (T3) POP NOR OR POP))
((((AND T3 POP)) T1) T2) T2 T4 DUP (NOOP OR) OR (POP ((OR) DUP NOR
OR)))
```

The parentheses have no effect on the execution of the code, since there are no code modifying instructions. After simplification (removal of dead code an non-functional structure), the functionally equivalent, 60 point program is as follows:

```
(T3 T4 T2 T1 AND NAND OR T1 T2 OR T3 T2 T1 T4 NOR NAND T2 T4 OR NAND AND
NAND T4 OR T3 T2 T1 OR T2 T1 NAND AND NAND T3 T2 OR AND T4 NAND NAND T1
T2 T4 OR NAND T3 NOR OR AND T3 T1 T2 T2 T4 OR OR OR NOR OR)
```

Even simplified, the complexity of the program evolved is too high to easily comprehend or explain its function. It is notable, however, that despite the presence of stack manipulating code in the unsimplified program, it makes no functional use of the stack to store values for multiple calculations via the **dup** command. Every value processed by a Boolean function is either pushed directly onto the stack before it is accessed or is the output of a previous calculation. It is fair to say, therefore, that this Push program exhibits no modularity either in code reuse or partial calculation reuse.

The other notable point is that this program is minimally represented as 60 points, supporting the theory that 50 points is not enough space within which to evolve a solution to even-four-parity given this encoding. On the other hand, it is short enough that in theory it could have been evolved in the 100 point maximum length run. Because of code bloat, however, it was probably too hard to find a solution without considering individuals over 100 points long during the evolutionary process. Indeed, unsimplified, the program evolved took up 168 points, much larger than the limit allowed in the previous run.

The other successful individual evolved by this run contained 199 points unsimplified and 56 points when simplified. Found at generation 45, its simplified version is as follows:

```
(T4 T2 T1 NOR T3 OR T1 T4 T3 NOR T2 NOR NOR AND T4 NOR T1 T1 T4 NAND NOR
OR NOR T1 T3 NOR T1 T2 T3 NAND T4 NAND T2 T3 NOR NOR NAND OR T4 T1 AND
T1 T4 NOR T3 T2 NAND T3 T2 OR NAND NOR OR AND OR)
```

This program little resembles the other solution by visual inspection, and it is reasonable to suspect that each uses a different system for determining parity. The only recognizable similarity was again that no use of the stack was made to store values for multiple calculations. This program, therefore, also cannot be considered modular.

Since PushGP had such difficulty in finding solutions in these runs, it was hypothesized that perhaps even a 200 point limit was too constraining a space within which to search, and another run was initiated to examine the performance with a larger limit.

4.7.4 Minimal PushGP instruction set and max program length 300

In this trial the only parameter changed was the maximum program length, which was increased to 300.

Results

Out of 32 independent runs, four successful solutions were found. With such a low successrate, the computational effort cannot be calculated with high significance. Roughly, however, it is 47,000,000 at generation 62, which is a quarter less effort than the last trial.

The following table summarizes the details of the successful programs evolved in this trial:

Found at generation	# of points, unsimplified	# of points, simplified
54	234	80
50	207	44
95	274	48
62	247	76

Notably, all of the unsimplified programs are over 200 points in length, however, after simplification all are less than 100 points in length. Thus it appears that the reason performance improves on this trial is not because the solution to the problem is hard to implement in less than 200 points. Rather, it appears that it is just hard to evolve a solution without lots of space to work with. Again, it appears that code bloat is preventing the direct evolution of solutions that take up less than 100 points.

Analysis of the code of the four successful individuals shows that all are similar. They only make use of the Boolean terminals (T1…T4) and the Boolean logic functions. None use **dup** to store partial results, and hence none can be considered modular in any way.

The shortest evolved program, after simplification, is no different in basic structure than the other evolved solutions to the four-parity problem. As can be seen by inspection, this 44-point program looks very much like the two programs presented in the last trial:

```
(T2 T3 T1 T4 NAND T1 T4 OR AND OR T3 T1 T4 NAND T1 T4 OR AND NAND AND
NOR T2 T3 T1 T4 NAND T1 T4 OR AND OR AND T3 T1 T4 NAND T1 T4 OR AND NAND
AND OR)
```

The only difference is that this program has more length-efficient ordering of terminals and logic functions. Even at this length, however, this program is not particularly easy to interpret. Since it is clear that it has no modularity and does not take any advantage of any Push specific features, no analysis will be made of the algorithm created by this program.

4.7.5 Conclusions of PushGP research with terminals

Of the two successful trials with terminals, the one that did the best allowed programs of up to 300 points in length. This is odd, given that one of the programs evolved in that trial only took up 44 points once simplified. It appears that evolution with PushGP, terminals, and the standard PushGP genetic operators, heavily encourages code bloat.

At this point it seems clear that using terminals and no code manipulation with PushGP to evolve even-parity functions is not a very effective strategy. The performance of the system is over 100 times worse than the results Koza achieved without ADFs.

For future research, it would be interesting to see if performance could be improved by just adding the instructions to allow code manipulation and execution. If so, this would be a powerful confirmation that PushGP can make productive use of the code stack to produce modular solutions. Clearly, in the research conducted on the even-four-parity problem with the full Push instruction set, the modularity of the problem was always exploited by evolution. It seems likely, therefore, that PushGP would also make use of modularity if the instructions were available when solving the problem with terminal inputs.

4.8 Even-six-parity

In the six-even-parity problem, the training set is 64 fitness cases, consisting of each possible six combinations of Boolean true and Boolean false. Fitness is the sum of the number of incorrectly classified fitness cases.

In Koza's work in Genetic Programming II, he was unable to evolve solutions to this problem in 19 complete runs without the use of ADFs. Koza speculates that, with enough runs/generations/large population size, a solution could be found to this problem without ADFs. Since no run succeeded, the true computational effort without ADFs cannot be estimated.

Koza did, however, estimate its lower boundary. This estimate was formed by showing that if one of the 19 runs had succeeded after all runs had completed one additional generation, the computational effort would equal 70 million. There is no way to predict the true number of runs that would have been required, and none of the 19 runs were close enough to suggest that only one more generation was required. The true computational effort required, therefore, is probably significantly higher.

Instead of continuing those runs to find the true value, however, Koza proceeded to demonstrate that by using ADFs, he could find a solution with a much lower computational effort of 1,344,000. By any measure for this problem, adding ADFs produced a drastic increase in performance.

4.8.1 Solving even-six-parity with full PushGP instruction set and stacks

This section addresses the question of how well PushGP can evolve a solution to the even-six-parity problem given the unconstrained search space of the entire Push language.

As with the lower arity versions, the most straightforward encoding of the parity problem for PushGP was to provide the six Boolean values of each fitness case by pushing them onto the Boolean stack.

Because of the drastic difference in this function set, no attempt was made to keep the run parameters maximally similar to those used by Koza.

Parameters

Fitness Cases	64 total possible permutations of 6 Boolean arguments
Population Size	5000
Maximum Generations	100
Maximum Program Length	50 points
Instruction Set	Full PushGP function set (44 instructions and 5 type specifiers) and integer, Boolean and symbol ephemeral random constants
Crossover, Mutation, Duplication	45%, 45%, 10%
Input	Four Boolean values pushed onto the Boolean stack
Fitness Measure	The sum of the number of incorrectly classified fitness cases
Termination	Fitness = 0

Results – Computational Effort

Out of 50 independent runs on this problem, 22 successful individuals were found. The computational effort was 2,280,000 at generation 37. This compares very well with the performance found by Koza by some measures, and not so well by others. In terms of absolute computational effort, performance with ADFs was better. However, the drastic difference in instruction set does not make this a particularly fair or worthwhile comparison.

In terms of scaling, PushGP took only a little over twice as much effort to solve even-six-parity as even-four-parity (2,280,000 vs. 945,000). With ADFs, however, the change in effort between even-four-parity and even-six-parity was an increase of more than seven-fold (1,344,000 vs. 176,000). PushGP scaled much better as the problem difficulty increased. If this is indicative of scaling for even harder parity problems, then, with a high enough arity problem, Push should start to perform better than ADFs in absolute computational effort. Whether or not this is the case, howver, is question to be addressed by future research.

Results – Individuals

The successful individuals, as evolved, are long, complex, and bloated with nonfunctional code. For instance, in one run, the following program emerged in generation 25:

```
((CONVERT CONVERT =) - (>) ((CONVERT CONVERT =) (SET) CONVERT) (DO*
APPEND (= ((<) CDR) (DUP ((INSERT SWAP) POSITION) (DO*) DO (<)) AND)
(AND ((QUOTE INTEGER SIZE) IF NAME)) DO) LIST +)
```

Simplified, however, the program's operation becomes much more clear:

```
(CONVERT CONVERT = CONVERT CONVERT = CONVERT DO* =)
```

This program is similar in structure and function to those evolved for even-four-parity. Examining the other successful programs evolved in this trial, the use of the same instructions and basic algorithm is found throughout. Consider a random sampling of four other successful individuals from this run:

```
(CONVERT CONVERT CONVERT = CONVERT = CONVERT = DO*)
(CONVERT CONVERT = CONVERT CONVERT = DO* CONVERT =)
(DO* CONVERT CONVERT = CONVERT CONVERT = CONVERT =)
(CONVERT DO* = CONVERT CONVERT = CONVERT CONVERT =)
```

It is clear just by visual inspection that all are very similar. While the exact placement of the instructions vary, each program uses combinations of **convert** and **=** to create a Boolean equality operator, and uses **do*** to double the number of times the program's code is evaluated. Thus, all these programs can be considered modular. Furthermore, this modularity arose, just as in the even-four-parity problem, without any explicit selective pressure for modularity. The problem was inherently decomposable, and PushGP exploited that modularity to find the solution.

Not all evolved solutions, however, are slight variations of the programs shown above. In one case, PushGP found a solution after 30 generations with a somewhat different method for determining if the inputs satisfied the even parity problem. Unsimplified, it is not immediately clear that is different from the other programs:

```
(CONS ((CDR (APPEND PULL (CONVERT (CONVERT) CONTAINER))) ((= (CONVERT
CONVERT =) CONVERT) (= (DUP) DO)) (AND) (AND DO) (MEMBER AND
((INSTRUCTIONS CONS CONTAINER REP) (AND)))) < POSITION)
```

Upon simplification, however, it is clear this program works in a different way:

```
(NOOP (CDR NOOP NOOP CONVERT CONVERT NOOP = CONVERT CONVERT =
CONVERT = NOOP DO NOOP NOOP NOOP) NOOP)
```

Interpreting this program is more difficult than the other solutions, since its method of producing modularity is more complex. It is the first Push program evolved so far where the structure of the code (i.e., parenthesizes) actually plays a part in its execution. It is also the first Push program where the location of non-functional instructions is important. Removing the **noop**s causes the program to quit working.

All of the additional structure in this program is caused by the fact that it uses **do** instead of **do*.** It therefore has to pop the code stack itself in order to stop execution from recursively evaluating it until Push aborts because the program has run too long. Popping of the code stack is achieved by the **cdr** instruction, which will remove one instruction at a time from the front, until finally **do** is removed. The recursive call ends at that point, but by then enough instances of **convert** and = and have executed to solve the even-six-parity problem.

It is interesting to compare this solution to the other programs evolved that solved even-six-parity. In this case the modularity was produced not by a single instruction, but by a large collection of code and hierarchical structure. Still, evolution favored maintaining modularity over duplicating the functional code enough times such that each instruction only needed to be executed once to produce the answer.

Furthermore, it is notable that PushGP has now found two rather different methods of creating modularity to solve this problem. By letting the Push language itself control how modularity is built, evolution can select the easiest path to the solution, rather than relying on the programmer to decide if the problem is best solved iteratively, recursively, or with some other control structure.

Results – Conclusion

In this problem PushGP again demonstrates that evolution can find modular solutions when the problem predisposes itself to be solved via modular code. The absolute computational effort on this problem was not as good as Koza's results with ADFs. Based upon the scaling of the amount of effort in comparison to the even-four-parity problem, however, it appears that PushGP might begin to have a better absolute computational effort than genetic programming with ADFs, if the arity of the parity problem is sufficiently increased.

Since, however, increasing the arity of parity problems drastically slows down the evaluation of fitness, this possibility will not be tested formally at this time. Instead, the next problem undertaken with PushGP attempts to bypass the time consuming evolution of separate, increasingly-larger parity problems.

4.9 Solving even-N-parity

As an alternative to evolving separate solutions to increasingly high arity even-parity problems, this section pursues evolution of a single program that can determine even parity for an arbitrary size input.

This is not a problem well suited to traditional Lisp-based genetic programming. If the Boolean values were provided as terminal functions in a tree representation, there would be no way to add higher arity inputs to the tree after evolution ends with a successful individual found.

This is a problem particularly well suited to PushGP, however, because of its stack architecture. The number of Booleans pushed onto the stack can set the number of inputs to the even-parity classification problem. Three inputs = even-three-parity, six = six-even-parity. Since an arbitrarily large number of inputs can be pushed onto the stack while still providing a uniform interface for extracting the values, there is a potential for PushGP to generalize beyond the input cases and produce a program that classifies any size list of Booleans as satisfying even parity.

4.9.1 Fitness cases

Selecting fitness cases is particularly tricky for the even-n-parity problem. The requirement is to provide enough different cases that finding the general solution to even-parity is easier for evolution, than finding a way to explicitly, and separately, deal with each variation in input size. At the same time, the more fitness cases included, the slower each fitness evaluation, and the slower the run. In order to allow a reasonable number of runs to complete within the time remaining for this project, 88 fitness cases were selected.

These cases include

- All 64 fitness cases from the even-6-parity problem
- All 16 fitness cases from the even-4-parity problem
- All 8 fitness cases from the even-3-parity problem

These cases were chosen because they represented both problems with even and odd numbers of inputs, and because a total of 88 fitness cases was still small enough for a single run to complete within a couple days on the hardware available.

4.9.2 Measuring generalization

The ultimate goal is to evolve a program that generalizes even parity to any arity input. This is not the only form of generalization, however, that might evolve. In lieu of full generalization, one possible partial generalization is a program that just correctly classifies all of the inputs in the training set. This would be a generalization from calculating parity of a single arity, to parity of 3,4 and 6 arity, and it may turn out to be a difficult task in of itself.

Even more impressive would be a program that generalizes to lower arity inputs not contained in the training set, such as even-5-parity and even-2-parity.

Most impressive of all would be a program that also generalizes to higher arity problems, namely those above the cases it was tested on. If such a program evolves, it is possible that it would just generalize to a few of the higher arity parity problems, such as 7 or 8 even-parity. It might also be possible to evolve a solution to an arbitrary size parity problem, limited only by member and machine cycles.

4.9.3 Trial 1: maximum program length = 50 and 100

As a starting point for this problem, the same PushGP settings were used as for the even-six-parity runs (excepting, of course, the fitness cases). This problem is probably harder than the even-6-parity problem, so another run was also initiated that used a maximum program length of 100. Since only a few runs of both conditions were completed (henceforth referred to as maxlen-50 and maxlen-100), the results of both will be discussed together.

Trial Parameters

Fitness Cases	All 88 combined fitness cases from even-3, 4, and 6-parity problems.
Population Size	5000
Maximum Generations	100
Maximum Program Length	50 points (maxlen-50 condition) 100 points (maxlen-100 condition 2)
Instruction Set	Full PushGP function set (44 instructions and 5 type specifiers) and integer, Boolean and symbol ephemeral random constants
Crossover, Mutation, Duplication	45%, 45%, 10%
Input	Three, Four, or Six (depending on the fitness case) Boolean values pushed onto the Boolean stack
Fitness Measure	The sum of the number of incorrectly classified fitness cases
Termination	Fitness = 0

Results

In one run, a program was found at generation 84 that answered correctly all 88 fitness cases. To achieve this result, 20 independent runs were started, with their configurations divided between maxlen-50 and maxlen-100. Not all runs completed, however, before the machines employed had to be returned to other uses, so no statistics could be collected in terms of which configuration (maxlen-50 or maxlen-100) was most effective. All of but one of the five most fit programs evolved, however, came from the maxlen-1000 configuration, as did the only program with 0 fitness.

The program with zero fitness was 92 points long before simplification:

```
((CONVERT (CONVERT)) (((CONVERT LIST SET ((= CONVERT DO* (MEMBER -)) -))
(INTEGER MAP) ((+ CAR (SIZE ((NULL =) (CONS) QUOTE) (- ((GET (CONTAINER
-)) T ((CONVERT SET (= (CONTAINS) NTH SUBST (ATOM)) ((CONVERT SET DO*
(NAME - (LIST) 6)) -)) = (DO (BOOLEAN GET) CONVERT / IF))) POSITION) DUP
SET) IF) (BOOLEAN GET) SWAP (CONVERT OR) IF)) / LIST))
```

As always with unsimplified individuals evolved by PushGP, the algorithm employed by this program is unclear. Simplification reduced the program to only 19 points:

```
(CONVERT CONVERT CONVERT = CONVERT DO* - - INTEGER + SIZE = QUOTE NOOP
DUP BOOLEAN CONVERT OR)
```

This 19 point program is functionally identical to the 92 point program, both for the fitness cases it was trained on, and in terms of ability to generalize to parity problems

Even with simplification, it is not easy to tell how this program works, and that analysis will not be attempted here. It is, however, easy to test this program's ability to generalize on the even-n problem. It already satisfies the least stringent test of generalization; that of solving even-3, 4 and 6 parity. The more important question is can it generalize to inputs it has not seen? Yes and no.

It successfully generalizes to all even-5-pairty fitness cases, none of which it was exposed to during evolution. It does not, however, correctly classify even-2-parity. Indeed, it classifies all four fitness case incorrectly. The program also does not successfully generalize to any higher arity problems, such as even-parity 7 or 8. For the higher parity test, it can only answer correctly for the cases where the parity of the first 6 Booleans is equal to the parity of all the Booleans pushed onto the stack.

4.9.4 Future directions

Judging from the performance of the program describe above, it appears that PushGP has only evolved the ability to classify parity of arities between the lowest and highest arities it was tested on. It will take more runs (and varied collections of fitness cases) to discern if this is really the case. Whether this is true or not, the issue of manipulating PushGP into generalizing to more than just one arity beyond what it was evolved with is still unsettled.

One possible solution is to pass the program an explicit count of the arity problem it should solve. This numeric value could easily be provided by pushing it onto the integer stack. The difficulty is to make sure that the evolved programs don't just ignore the integer, as clearly evolution does not need it now to find a solution that satisfies all the fitness cases. One way to make sure that the pushed integer is considered is to include a few special fitness cases where the number of Booleans pushed on the stack is larger than the integer value. If the extra Booleans change the parity of the fitness case, then the program will have to consider the arity argument pushed onto the integer stack to answer the problem correctly.

In a way, however, this makes the problem harder than the original desired solution. Before, the goal was only to find the parity of all the Booleans on the stack. Now the program must be flexible in two dimensions: the arity of the parity problem it solves, and the number of Boolean values it must extract from the stack. Admittedly, since the two numbers are the same, it may not make the problem not that much harder. Still, it means that the algorithm must end processing by explicitly consulting the integer stack, rather than just exhausting the Boolean stack in the process of performing calculations on its contents.

For future research, it would also be worthwhile to run more instances of the problem as encoded above to see if all solutions evolved possess the same level of generalization, or if there is variation between programs.

4.10 Conclusions drawn from this chapter

In this chapter, programs evolved by PushGP have been dissected in search of instances of modularity. The performance of PushGP has also been compared to that of genetic programming with ADFs.

On the first count, many examples of modular code were found. PushGP was able to exploit the stack such that mathematically modular solutions could be evolved, without resorting to explicit code modifications.

In many trials, PushGP was also able to make use of the code stack to produce modular solutions that matched the decomposition of the problem, and it did so without any explicit selective pressure for modularity. The modular, decomposable nature of the problem itself was sufficient to repeatedly bias evolution towards building modularity.

Many programs demonstrated the use of the code stack as static location to store a functional block for execution on demand. These trials indicated that the code stack could be used to evolve modular code directly. One program also showed that the code stack could be used to create modularity by dynamically modifying the program on the stack as it executed it.

This chapter also described preliminary research on the performance of PushGP. Performance was measured in terms of the computational effort for each run. PushGP did not show any absolute improvements in computational effort over ADFs. Preliminarily results suggest, however, that the modularity that PushGP evolves allows it to scale well to harder problems, in the same way that ADFs do. At least on parity, it is possible that PushGP scales even better than ADFs do, and that PushGP would have a computational effort advantage on sufficiently hard parity problems.

5 Variations in Genetic Operators

The base PushGP system implements straightforward crossover and mutation operators (see section 3.4). Since they were designed with an emphasis on simplicity, there are a number of potential areas where they exhibit less than optimal behavior. Nonetheless, for consistency, these operators were used in all the other research on PushGP discussed in this document.

In this section, however, variations in these operators are explored with the idea of finding operators better tuned to the PushGP representation, for use in future work.

5.1 Performance of base PushGP operators

One of the main motivations behind developing alternative operators is both implementations degenerate into duplication operators whenever the children produced are larger than the **`max-points-per-individual`** for the run.

Quick tests on two symbolic regression problems suggest that, as runs progress, the number of children produced which are above the size limit quickly becomes significant. The following table reports the averages observed from 5 separate runs of each problem:

Problem	$5x^2 + x – 2$	$x^6 – 2x^4 + x^2$
Population size	5000	4000
Max points	50	100
% Children above size limit		
Generation 15	20%	0%
Generation 50	45%	20%
Generation 100	50%	30%

While a high rate of duplication might be useful in some problem domains, 30%-50% duplication is an abnormally high setting for most problems. If such a high ratio is desired, it should be specified directly by the researcher, rather than mandated by the choice of crossover and mutation operators.

A secondary problem discovered in the examination of the evolving populations was that mutation and crossover were most likely to select the leaf nodes of trees as the target of their modifications. This was a result of random point selection. Programs contain many more leaf nodes than internal nodes, so random point selection usually selects a leaf. Compounded over the run, this meant that deeper structure of programs (i.e., their internal nodes) showed *much* less variation across the population than did the leaf nodes.

The following sections detail the variations that ensure that mutation and crossover never produce children that must be discarded due to size. Some modifications also addressed the issue that the inner nodes of the population stagnate because genetic operators most often selected leaf nodes as their target.

5.2 Variations in crossover

All of the following operators take two parents and output a new child, with the requirement (unlike traditional PushGP crossover) that the child is some sort of combination of the parents.

5.2.1 Smaller-overwrites-larger crossover

This operator will select any point within a program for crossover, which means that, in many cases, the two crossover points are just single instructions. In this case, the program cannot grow, and crossover proceeds as it does with the default PushGP algorithm. Behavior varies, however, when at least one of the crossover points is a tree.

In this case, two new children are produced by swapping code starting at the root nodes of each point (be it a tree or instruction). The smaller of the two resulting programs is the child ultimately returned by this operation. In effect, smaller code replaces larger code. As long as neither parent is larger than the maximum, the child produced will always be below the size limit.

The downside of this approach is that children never grow larger than their largest parent, so the average length of the population will tend to decrease overtime. For effective search, therefore, this operator must be matched with another that makes some children that are larger than their parents.

5.2.2 Smaller-overwrites-larger-on-overflow crossover

Since the previous operator strongly tends toward shrinking the size of its children, this operator was created to allow a bounded amount of growth. It operates like the default PushGP crossover, except in the case that the child produced is over the size limit.

In that case, it instead performs the crossover operation defined in the previous section (smaller-overwrites-larger), which is guaranteed to produce a child of crossover of valid size. In practice this means that code can grow or shrink as specified by the fitness landscape until it gets near the maximum size, at which point it is much more likely that crossover will produce a smaller child.

5.2.3 Smaller-tree-overwrites-larger-tree-on-overflow crossover

In subtree-only crossover, each node selected is an internal point in the parent's trees. This ensures that crossover rarely acts like a single point mutation operator. In the rare case that a parent has no internal points, this operator reverts to choosing any node at random.

The purpose behind this operator is to increase the variation in the internal nodes of the individuals in the population, counteracting the observation that most variation is only seen in the leaf nodes.

Overflow (programs larger than the maximum program limit) is avoided by the same method used by smaller-overwrites-larger-on-overflow: when randomly produced children are too large, a guaranteed legally-sized child is produced instead by using the larger tree as the target point and replacing it with the smaller tree.

5.2.4 Subtree-biased, smaller-overwrites-on-overflow crossover

Simply, an operator that applies

- smaller-tree-overwrites-larger-tree-on-overflow 90% of the time
- and smaller-overwrites-larger-on-overflow the rest of the time.

This attempts to deal with the stagnation in internal nodes, while still allowing external nodes to be the target of crossover some of the time. The 90%-10% ratio was selected because it mirrored the percentages used by Koza in the subtree-biased crossover used in Genetic Programming I and II.

5.3 Variations in mutation

The new mutation operators are designed to address the issue of the creation of children over the size limit. All ensure that the application of the mutation operator is much more likely to return a mutation of the parent independent of the size of the child.

5.3.1 Bounded mutation

This variation is much like traditional PushGP mutation. A target point is randomly selected for mutation (internal or leaf node) and it is replaced with a randomly generated tree or instruction. The only difference is that when a randomly generated tree is generated to replace the mutation point, the size of that tree is limited such that it, plus the length of the parent, is never larger than the limit on maximum program length.

For instance, if a parent is 95 points and the limit is 100 points, the randomly generated tree is allowed no more than five points. This means that, as programs grow large, not only is there no chance of surpassing the size limit, but it also becomes more likely that mutation will decrease their total length.

5.3.2 Node mutation

Node mutation randomly modifies a single instruction or constant within a program, replacing it with another single point of Push code. This mutation does not select points that are entire subtrees, so the structure of the individual does not change.

The number of nodes modified is controlled by a probability calculation that describes the likelihood that each individual piece of code will undergo mutation. A probability of 20% means that, on average, 1 out of 5 nodes is modified, whereas a 1% probability means only one out of 100 is modified.

Since replacing an instruction with another instruction never changes the size of an individual, this operator never results in a child over the size limit, as long as it is given a parent who is not over the limit already.

5.4 Empirical tests with new operators

In this section, combinations of the new operators are compared against the performance of the standard PushGP operators. While the standard operators are probably a good fit for problems where high amounts of duplication is useful later on in the run, in problems where this is not the case, these new operators have the potential to improve performance.

A simple symbolic regression problem was chosen as striking a balance between being representative of problems where performance might improve, and having a fast enough wall-clock performance to quickly allow measurement of performance over a reasonable number of runs.

The symbolic regression problem was that of finding $5x^2 + x - 2$ from 10 data points produced by x=0…10 in the space of integers.

Problem parameters for symbolic regression of $y = 5x^2 + x - 2$

Fitness Cases	10 y values created from x=0 …9
Population Size	5000
Maximum Generations	100
Maximum Program Length	50 points
Instruction Set	+ - * / dup pop noop 1 ephemeral-random-integer
Crossover	45%
Mutation	45%
Duplication	10%
Input	Value of x pushed onto the integer stack
Fitness Measure	The integer distance between the top of integer stack and the target y value, summed over all fitness cases
Termination	Fitness = 0

Results

Each condition was tested over 100 independent trials. For each trial the estimated computational effort is reported. Computational effort is calculated via the methods described in section 4.2. Each trial was given a short name to facilitate referring to it from the other results.

The following four conditions tested the effectiveness of bounded mutation and smaller-overwrites-larger-on-overflow crossover as compared to standard Push GP operators.

Trial name: STD-STD
Crossover: PushGP standard
Mutation: PushGP standard
Effort: 375,000 at generation 24
This value represents the baseline performance of PushGP. For the new operators to be an improvement, they will have to beat this.

Trial name: STD-SOOO
Crossover: Smaller-overwrites-larger-on-overflow
Mutation: PushGP standard
Effort: 420,000 at generation 27
This value represents the change in performance caused by just adding this crossover operator. The performance is not as good as the baseline PushGP performance shown in STD-STD, though the amount of difference is only about 10%. Since the computational effort is just an estimate anyway, this amount of difference is not enough to draw any conclusions.

Trial name: BM-STD
Mutation: Bounded mutation
Crossover: PushGP standard
Effort: 450,000 at generation 29
This result shows the effect of just using the new mutation operator, combined with the standard PushGP crossover. The effort required with this setup is roughly one quarter higher than that of the STD-STD. In isolation, it appears that neither BM-STD nor STD-SOOO improve performance on this problem.

Trial name: BM-SOOO
Mutation: Bounded mutation
Crossover: Smaller-overwrites-larger-on-overflow
Effort: 380,000 at generation 37
This result shows the effect of combining both the new mutation and crossover operators. Performance is roughly the same as STD-SOOO. This implies that the two new operators work better together than when either is separately paired with the old operators. In the end result, however, the performance is so similar to STD-STD to suggest that neither has much advantage over the other for this problem.

The following conditions compare the effects of biasing crossover to operate on inner nodes of trees, and the combined effect of including this bias and using the new mutation operators.

Trial name: STD-STOOO
Mutation: PushGP standard
Crossover: Smaller-tree-overwrites-larger-tree-on-overflow crossover

Effort: 660,000 individuals, at generation 32
This result shows the effect of restricting crossover just to swapping trees, instead of instructions. Computational effort is almost twice as high as compared to the PushGP baseline (STD-STD). This trial's performance can also be compared to STD-SOOO, since both crossover operators use the same method for avoiding overly large children. This comparison also shows the performance of STD-STOOO to be poor, with about half again higher computational effort than STD-SOOO. Requiring that trees are targets of crossover does not help on this symbolic regression problem.

Trial name: BM-STOOO
Mutation: Bounded mutation
Crossover: Smaller-tree-overwrites-larger-tree-on-overflow crossover
Effort: 555,000 individuals, at generation 36
This trial shows the results of combing crossover restricted to inner-nodes and small-overwrites-larger length control with the bounded mutation operator. The addition of bounded mutation improves performance slightly compared to the previous trial (STD-STOOO), but performance is still not as good as trials like STD-STD or STD-SOOO, which have no bias towards inner node selection.

Trial name: BM-SBTOOO90
Mutation: Bounded mutation
Crossover: Subtree-biased, smaller-overwrites-on-overflow crossover
Effort: 510,000 individuals at generation 33
Perhaps the last two trials did so poorly because they never allowed leaf nodes to be the target of mutation. This trial shows the attempt to improve upon result of BM-STOOO by using inner-node crossover 90% of the time, but also allowing crossover rooted at the leaf nodes 10% of the time. While this did increase performance very slightly over BM-STOOO, the change is not large enough to be significant.

Since the collected results suggest that none of the new crossover operators improve performance on the test problem, it was hypothesized that perhaps the test problem was particularly brittle to crossover. Perhaps crossover hardly, if ever, produces a more fit individual than its parents. If this is true, decreasing the instances of crossover degenerating into duplication would not produce better performance on this problem. After all, if crossover is always destructive (or does not lead evolution into new areas that contain better fitness), then decreasing the number of applications of crossover should speed evolution.

To test the possibility that crossover is not providing children useful for evolutionary progress, several trials were run that used no crossover operators (but still with 10% duplication as in the previous trials).

Trial name: STDMUT
Mutation: PushGP standard

Crossover: none
Effort: 310,000 at generation 30
This trial just used standard PushGP mutation and duplication and achieved the lowest computational effort so far. This suggests that if crossover is a useful operation to apply to this problem at all, it provides less average gain across generations than does mutation.

Trial name: MUT-MPOINT1/20
Mutation: 45% PushGP standard, 45% node mutation w/ 1/20 probability
Crossover: none
Effort: 340,000 at generation 33
Since the last trial showed that this problem is well suited to search via mutation only, it seemed worthwhile to see how performance would vary with the addition of the node mutation operator. The node mutation operator's behavior is controlled by it the probability assigned to it that it will modify a given node in the program. When this operator was applied (only 45% of the time) a mutation probability of 1/20 was used (meaning that 1 in 20 points was modified on average, per individual). The resulting computational effort was slightly worse, but very close to that of STDMUT. For this problem then, adding the node mutation operator (with 1/20 probability) did not improve performance of evolution.

Trial name: MUT-MPOINT2x
Mutation: 45% PushGP standard, 45% node mutation w/ 2xSize probability
Crossover: none
Effort: 495,000 at generation 32
This trial used the same settings as MUT-MPOINT1/20, but with a different probability of node mutation, scaled such that any size program was likely to have 2 nodes changed when the node mutate operator was applied. This means that the mutation rate per program will stay constant as evolution progresses, rather than constant per number of instructions processed. For this symbolic regression problem, however, this setting did not lead to an improvement; rather it increased the computational effort by almost half.

5.5 Conclusions drawn from these runs

These runs showed that the default PushGP genetic operators can perform relatively well, given their failings. Attempts to address those failings did nothing to improve performance on the test problem selected, and, in many cases decreased it significantly.

It may be that the new methods designed to deal with code growth and leaf biased mutation and crossover had their own failings that caused just as many problems as the default PushGP operators. It also could be the case that the default PushGP operators strike a good balance between producing well formed children most often and being unable to successfully apply the operator in all cases. If this is so, it may be unreasonable to expect performance to be improved, at least on this symbolic regression problem.

It is also notable that the characteristics of the test problem may not be as representative as initially thought of problems where high levels of duplication and little internal node mutation would impede performance. Particularly, the fact that removing crossover entirely actually improves performance suggests that this problem benefits fairly rarely from crossover. Depending on the nature of the improvements in the crossover operators tested here, this might mean that a generally less flawed crossover operator might actually perform worse on this problem.

Given a problem like even parity, then, where the symmetry in the problem should reward crossover, the comparative performance may change. Nonetheless, the conclusion for now is that the default PushGP operators are probably not impeding performance that much. Designing new operators, then, probably does not need to be the highest priority for further research with PushGP.

6 New ground – evolving factorial

Factorial requires more sophisticated control structures than the symbolic regression problems encountered to date. Most symbolic regression problems, in fact, are encoded with a purely mathematical function set, without the capacity for iteration and conditionals. Factorial requires some form of both. If a solution to the factorial problem can be evolved with PushGP, therefore, it will further demonstrate that evolution can make effective use of Push's controls structures.

6.1 Can it be done?

This research does not attempt a detailed analysis of exactly what control structures are necessary and sufficient for factorial; that is not the issue at hand. PushGP, however, contains at least one such set of instructions, which can be shown by the existence of the human constructed factorial program written in Push described in section 3.3.4. If Push can represent factorial, then, by extension, PushGP should be able to find it, given enough trials. The question is, with what settings, and with how many individuals processed?

One side issue is that factorial may be a GP-hard problem, in that it has a deceptive fitness landscape, with the easily reached local minima evolutionarily far away from the global minima. If this is the case, evolving a solution to factorial will be difficult, independent of the successfulness of evolving PushGP control structures. The analysis of the results in the next sections will consider this issue in more detail.

6.2 Factorial trial 1

Factorial parameters

Fitness Cases	10 points from x=0 to 10
Population Size	10,000
Maximum Generations	100
Max points per individual	50
Instruction Set	REP DUP < * - / QUOTE IF DO CODE INTEGER ephemeral-random-integer
Crossover	45%
Mutation	45%
Duplication	10%
Input	Value of x pushed onto the integer stack
Fitness Measure	The integer distance between the top of integer stack and the target y value, summed over all fitness cases
Termination	Fitness = 0

In this trial, default PushGP values were used, except for a large population size (selected because the problem was deemed particularly hard). The instruction set was also designed to have similar instructions as the hand coded solution, without duplicating it exactly.

Results

Out of 100 runs, no solution or near solution was found. Without success, computational effort cannot be calculated for use as a metric of effectiveness of these parameters. Instead, the raw results of the best and worst results are shown here for examination of trends in the data.

The most fit program evolved had a total error of 3130, summed across the results from x=0…9. Simplified to remove unused instructions, it consists of the following push program:

```
(DUP *  -3 - 9  /  DUP * -2 -  DUP 9  / DUP DUP * * 6 * *)
```

Note the complete lack of use of **code**, **quote** and **do**, which would be needed to make factorial operate as per its recursive or iterative definition. PushGP's best attempt is a constant function, and no more.

Errors, from the best 7 run results (includes every run terminating with < 10k fitness):

Individual	Total Error	Error, X = 0	1	2	3	4	5	6	7	8	9
1	3130	1	1	2	6	24	54	144	666	2070	162
2	3490	1	1	2	6	24	114	378	348	2490	126
3	5314	1	1	2	6	24	120	440	3272	800	648
4	7279	511	479	130	71	24	219	1360	2928	1152	405
5	9084	62	62	61	57	39	129	729	3377	3857	711
6	9568	1630	316	1	149	131	119	403	3409	2935	475
7	9619	0	0	1	5	23	107	98	141	8518	726

As this table shows, no best of run individual was successful at tracking the trends of the data. On average, the least error is seen for the lowest few values of X, and 9. This suggests that evolution is stuck in solutions that produce small values for small X, and a really big value close to the right value for 9, rather than abstracting the relationship between successive values. Getting the right value for X= 0,1 seems very important in a problem where every calculation on higher inputs uses those values in producing the right answer per the definition of factorial.

Further suggesting that evolution is stuck in local minima far from the correct solution space, the one individual (#7) that gets the first two outputs perfect has the worst total error of the group. Note also that its total error is almost the same as #6, who has the worst error on x=0. It could be debated as to whether the x=0 case is critical to solving the problem, but as the encoding stands now, it seems that evolution is not particularly favoring the programs that fit that case over others.

Another issue is how the worst trials of the 100 runs performed, in comparison to the best. Are the same trends observable?

The worst runs from this trial (all the individuals with error above 100k):

Individual	Summed Error	Error, X = 0	1	2	3	4	5	6	7	8	9
1	100159	1	1	2	145	843	5	6581	22581	63216	6784
2	114208	3	3	295	1154	4052	7028	5333	13972	79492	2876
3	125536	1	87	88	636	920	370	5274	37210	68224	12726
4	130415	1	1	2	390	1080	861	5624	31760	88776	1920
5	141982	1	1	2	66	264	3408	9648	43632	74880	10080
6	149805	1	1	2	69	24	980	9864	30240	71040	37584
7	159222	8	174	7	531	2447	4341	57	24019	72585	55053

By and large, these values mirror those of the best runs, except that even more of these individuals are finding reasonable values for x=0,1. This suggests that getting the initial cases right is not really an evolutionarily successful pathway to better overall fitness, using this fitness metric. Again, it is also noticeable that the first few fitness cases and the last fitness case are the ones that show the least error of their immediate neighbors, indicating that, for both the best performing and worst performing runs, the middle fitness cases are not being fit very well.

If evolution is pushing individuals into dead-end solutions, then this problem is indeed particularly hard for GP. It may be, however, that these issues are artifacts of other parameters in this run. Trials 2 and 3 will attempt to address this.

6.3 Trial 2

The shortest human-authored version of factorial (discussed in section 3.3.4) consumes 24 points of Push code. The maximum program length allowed in the previous trial was 50 points, which, while enough space to represent the human coded factorial, may not leave enough space for an evolved factorial solution, given the code bloat that goes hand in hand with genetic programming.

In short, perhaps the results seen in the last trial stem from 50 points not being enough space for modular code structures to evolve. To address this question, trial 2 increases the maximum program length to 100.

Results

In 100 runs, no solution or near solution was found. The best individual evolved had a total error of 4332, which is worse the best individual evolved in trial one. On the other hand, 26 runs evolved individuals with a fitness of <10k, vs. only 7 in trial one. This indicates that the average fitness of the population improved, but all of those individuals were stuck in even worse local minimums.

The trends in errors between x=0…9 for this set of runs also mirrors the findings of trial one, so the raw results will not be shown here. In short, on average the amount of error was lowest for x=0,1 and x = 9, as compared to the error on neighboring x values.

The conclusion is that the new search space included more easily reached local minima when the maximum program length was increased, but that the overall results of this run was not even as close to factorial as in trial one.

6.4 Trial 3

The final trial completed for this research project attempts to make more use of knowledge about factorial and Push to evolve a successful result.

Since a Push program already exists that solves factorial, its source code was used to guide the selection of parameters The modifications (described below) may be deemed excessive, but it seems better to start with a successful run, and then try making the encoding more general. If over-constraining the search to a supposedly fertile space does not succeed, however, then it suggests that space is not as fertile as thought.

The change between this trial and the last was to select an instruction set that contains the same instructions as present in human-coded factorial Push program. Each instruction was weighted so that it had the same likelihood of being selected as its frequency of occurrence in the human coded solution.

Of course, as discussed throughout this document, by constraining the search space the likelihood of finding a novel solution is decreased. The goal here, however, is not to find a novel solution, but to see if PushGP can evolve an already known result, given a search space as biased toward that solution as possible. If it not, it suggests that solving the problem will require more than just tweaking the instruction set.

Parameters changed between trials one and three

Max points per individual	100
Instruction Set	*Triple weighted:* QUOTE DUP *Double weighted:* CODE INTEGER 1 REP DO *Single weighted:* - * 2 < IF

Triple weighted instructions are three times as likely to be selected at random whenever a new instruction is generated by the genetic operators. Double weighted instructions work analogously.

Results

Out of 100 runs, no solution or near solution was found. The most fit individual evolved had a total error of 4142, worse that the first trial, and only slightly better than the second trial. Furthermore, only 6 runs terminated with fitness under 10k, suggesting that the strategy pursued in this run has little advantage over trials one or two. Finally, the raw results of the 6 best runs follow the pattern already established in the last two trials: the amount of error was lowest for x=0,1 and x = 9, and much worse for the middle values.

It seems that weighting the instructions toward an already known solution is not helpful, at least in the context of the rest of the current encoding of this problem.

6.5 Future Research

Out of 300 runs and 3 variations in the parameters of the problem, no remotely close solution was found. This suggests that more radical changes are required than just increasing the population size and the length of the run. Judging from the trends in the errors for different values of X, it seems possible that combinations of very different magnitude target values have led evolution away from iterative and recursive solutions and towards exponentially-scaling polynomials. This suggests that the problem, at least as represented here, is particularly hard for genetic programming because the crossover of two exponentially scaling polynomials is unlikely to yield anything but another polynomial.

One possible remedy would be to equalize the importance of minimizing error for each input so that a polynomial that only matches the two extremes of the output is less fit. This would mean that matching the shape of the curve would be more important than minimizing the error. Just matching the shape of the curve might make it hard to find an exact match for factorial, however, so perhaps the raw fitness plus the overall shape of the curve would be the best fitness metric for this strategy.

Another consideration is that calculating the right output for the lower values of X might be key to finding the correct solution later on (since, by the mathematical definition, those values will be used in the calculations for every higher X value). Rather than equaling the effect of error across each fitness case, perhaps the low X fitness cases should be given much higher impact on fitness (say, by multiplying them against a really large scaling value). This would mean that an error of 400 on x=0 has a much higher impact on fitness than the same error on x=9. This would emphasize that, while each case is important to the solution, fitting the initial values is most important.

Finally, given the three trials above, it seems that using control structures (**if**, **do**, etc) is not the easiest way to reach the local minima for this problem. This means those control structures are not well distributed throughout the fit members of the population, making the jump to the iterative or recursive definition of factorial hard via crossover.

Perhaps a system that rewards the execution of **do** and **if** in the calculation of the output would help. It would allow individuals with control structures more evolutionary time to find solutions that actually work better than individuals who are just high order polynomials. Building in such biases cleanly would be difficult, however, and, at this time, no clear method for doing so exists.

Part III

LJGP

7 Linear Coded Genetic Programming in Java

This chapter describes the linear-coded virtual-CPU based genetic programming system built in Java called LJGP (Linear Java Genetic Programming). I built this system as an experiment in creating a full genetic programming environment from scratch, to gain a hands-on understanding of the issues that go into such a system. I also built it with the goal of designing a system that could be used by researchers without large clusters of machines to devote to their work.

The next sections of this chapter will explain the practical motivations for creating a GP system in JAVA and the theoretical motivations behind the design of this specific GP system.

The next chapter is a user's guide to LJGP. It shows how to encode new problems for LJGP, how to read the programs produced by LJGP, and includes a quick summary of each package in the system and the important classes. The final chapter in Part III will describe, in narrative form, some of the runs conducted while building this system. The purpose of these runs was to both test the implementation of LJGP, and do pilot experiments on problems I hope to investigate further in the future.

7.1 Feature overview

LJGP currently implements the following features:

- Portability: 100% Java 1.1 compatibility, successfully tested on Internet Explorer 5.x and Netscape Navigator 4.x for both Macintosh and PC platforms, as well as the PC SUN JRE, the MAC OSX JRE, and the Symantec Visual Café JRE.
- Flexible hosting, either as part of a Java application or applet.
- Linear program genomes, executed via a virtual CPU interpreter optimized for speed and easy implementation of new instructions. The Virtual CPU supports arbitrary numbers of registers and immediate values for each instruction, and allows for the addition of multiple data types through subclassing the main **vCPU** class and **Instruction** class.
- A genetic operator package that performs crossover and mutation upon any Java array, not just Virtual CPU programs.
- A memory-efficient steady-state tournament selection system that is generalized to operate upon any individual possessing a fitness value, not just Virtual CPU programs.
- An SMTP-based report system that emails the results of a run hosted on a remote computer back to the host computer.
- Network support for shipping individuals, populations, and entire runs (in progress, or not) back and forth between machines with network privileges.
- Mid-run backups to disk or network servers in case the host machine crashes or is shut down.

- A logging system that centralizes the control of what textual data is recorded about the run, allowing easy, system-wide modification of the verbosity of the log.
- Optional GUI interface for monitoring a run and examining the status of the population.
- Automatic conversion of evolved programs into human readable text source code.
- Automatic conversion of text source code into executable Virtual CPU programs.
- Generic visualization class for displaying the behavior of agents in gridworld simulations.
- Emphasis on efficient algorithms to compensate for the Java performance hit over compiled code.

7.2 Practical motivation for using Java

LJGP was developed with the following goals:

- Portability
- Easy installation on host machines
- Easy distribution of runs
- Maximal utilization of non-homogenous computing environments

These goals stem from the fact that not all researchers and students have access to machines that can be dedicated to the full time execution of genetic programming runs.

Often, however, these same people work in environments where many Macintosh and/or PC desktop machines are dedicated to other tasks during the workday, but are unused otherwise. While it might be possible to harness these machines with software analogous to SETI@Home, but designed to execute a genetic programming runs, installing any sort of software often presents a significant barrier of entry.

Java offers a solution to this problem. Loading a web page in any Java enabled web browser, represents a much lower level of invasion, if a higher level of manual effort (though scripting software can easily make this too automatic). Furthermore, the same Java programs can target both PCs and Macs with ease, which is especially worthwhile since many academic environments contain a large mixture of both.

While the performance of Java runtime environments varies, the general consensus is that JAVA code executes slower than natively compiled C code. On the other hand, a Java GP engine has the potential to harness computer cycles that would otherwise be wasted completely. Furthermore, Java just-in-time compilers are improving all the time, so the efficiency issues of Java should become less noticeable in the future.

7.3 Overview of design decisions

LJGP incorporates the following two features that differ from the kind of genetic programming implementations seen in contemporary systems such those described in Koza's 1992 and 1994 work:

- A virtual machine language analogous to the opcode set of a RISC CPU, but designed for maximal human readability.
- A steady-state tournament-based population selection system with a generic interface that would allow other methodologies to be plugged in.

This section describes the motivations behind these two features (which are not unique to LJGP, but are unique among the current Java-based GP systems).

7.3.1 Linear machine code vs. trees

Goals: speed, simplicity of implementation, and ease of comprehension.

Much traditional genetic programming research evolves S-expressions, or some variant quite unlike the internal code executed by the underlying CPU. If the code evolved is to be used for more than just a research curiosity (i.e., in a production environment) then keeping the code closer to the level executed by the CPU should ease the ultimate translation into efficient code. Indeed, some research projects have experimented with mutating and evolving sequences of actual CPU opcodes, and evaluating the result simply by loading the code directly into the CPU (for instance, Nordin, 1994).

While the speed advantages are very compelling, such representations have important disadvantages. One, they tie the code to a specific CPU, and structure the instruction set to the limits of that CPU, rather than allowing the opcodes to be tuned to the particular problem, or even the general needs of evolution. Second, the limits of real CPUs (for instance, x86's small number of registers, and large number of opcodes hard-coded to only use certain registers) often mean that the machine code is hard to read, whether its source is evolved or not.

Some would argue that no machine code is easily interpreted by humans, but this is a mater of taste. Certainly, some are more readable in that they are uniform and possess a reasonably large number of registers. Since evolved code is rarely easy to interpret, be it in Lisp, C, an ANN, or a list of machine opcodes, it makes sense to at least start with a language designed for readability, rather than efficient implementation in silicon.

Implementation:

The virtual CPU created for LJGP addresses the above issues. Any feature of the CPU can be tweaked at will to fit the problem. Furthermore, the opcodes can be designed for maximal human comprehensibility (at least when the constraints of the problem do not require otherwise). Finally, to aid in interpreting the code, the vCPU package includes an optional code stepping feature that shows the contents of the registers at each time step and how they change as each instruction executes.

In LJGP, the virtual CPU (vCPU) is register based, with an infinite possible number of registers (the exact count specified by run parameters). There is no memory store or stack by default, but such constructs can be implemented by the addition of instructions that operate upon those memory structures and move their stored values into registers. The current list of implemented instructions is described in section 10.3 (which also contains some sample code).

Note that all genetic operators (one point crossover, two point crossover, point mutation, and point deletion) operate upon generic Java objects in arrays. This means that LJGP, while currently set up to use the vCPU system, can be easily switched to a different linear code representation should one be of interest in the future, without changing the features or operation of any other parts of the code.

7.3.2 Steady-state tournament selection

LJGP uses steady state tournament selection to control the evolution of the population (though the class hierarchy allows other systems to be swapped in easily). Tournaments were chosen because of their implementational simplicity, as well as for the relatively low computational costs of maintaining their internal structure (linear array without any need for sorting). Tournaments were made steady state to avoid having to keep the entire old generation in memory while producing a new generation.

The tournament produces one new individual per invocation, meaning it must be run repeatedly to produce a new generation. The algorithm used selects x (default: 4) individuals and performs a randomly selected genetic transformation upon the most fit member of that sample. If another individual is needed for the transformation (e.g. crossover), another individual is selected using the same method as above.

Under the default setting, the resulting child individual is then placed back into the population by selecting x (default: 4) members of the population and replacing the loser (the one with worst fitness) with the new output. This happens regardless of whether the new individual is more fit than the loser. Under an alternate setting, **keepBetterLosers**, the loser is only replaced if the new child is more fit than it.

Always replacing the tournament loser maintains population variation, while on average protecting the best members of the population from being overwritten. Depending on the problem this allowance for some backtracking in fitness can be useful, or not. While it can be turned off, in all of the following experiments with LJGP it has been left enabled. This was out of a desire for consistency between runs. Perhaps in the future research will address the question of what sort of problems benefit from using the **keepBetterLosers** setting instead.

7.4 Distributed processing

Traditionally, when GP (and GA) runs are distributed between machines, members of the population migrate between separate machines as the run progresses, bringing with them diversity and new code for crossover. See Tomassini (1999) for a review of contemporary implementations of this model of distribution.

This method does have problems when all the machines involved in a run are not necessarily likely to continue processing until run termination. This is a distinct possibility if the machines used are public terminals, or office machines that have other duties besides computational research.

If an instance of a distributed run is halted, then the amount of computational effort expended on that run (the number of individuals processed) will vary from the norm. If the rest of the machines continue on, the size of the total population between them will have decreased from that point on. It is unclear how results would change with a shrinking population size, but this dynamic source of semi-random changes is sure to be an extra cause of variation in results between runs.

It should be clear that when attempting to make considered decisions about the results of a GP run, adding a source of variability is undesirable because it reduces the ability to compare the effects of intended changes between runs. Therefore, in the current LJGP code, runs are completed separately, without any swapping of members of populations. Instead, LJGP uses the network to backup populations mid-run, and to send the results of a run back to a central host.

The use of backups allows runs to finish even if one run takes longer than the contiguous amount of time that the machine is dedicated to the use of LJGP (overnight, for instance). Periodic backups also allow semi-public machines to run LJGP, with the web page reloaded by the researcher after periods of high use of the machines for their official function. When LJGP is run as an applet, it is not possible to store populations to disk; due to the Java sandbox, therefore, populations are backed up to a server. In cases where LJGP can be run as an application, populations can be backed up to hard disk without requiring any interaction with the server.

Should a distributed model that does swap individuals be desired in the future, such a system should be trivial to implement, since already entire populations of individuals are transferable between machines.

8 LJGP User's Guide

This chapter explains how to encode new problems for LJGP. It also includes an overview of the source code that makes up LJGP, to aid understanding of where to start if more major changes and additions are desired other than creating a new instruction or fitness function. Finally, it contains a section on how to read the vCPU programs produced as output.

8.1 Encoding a problem

Problems are encoded by creating a new class that implements the TaskBox and FitnessMetric interfaces, which, respectively, specify how run parameters are initialized and how the evolutionary processing is started, and the functions to call to obtain the fitness of an individual.

The **/task/symbolic_regression/IntTournament.java** file in the LJGP distribution implements the traditional symbolic regression problem of finding the function $\mathbf{y = x^4 + x^3 + x^2 + x}$ given a collection of paired x and y values. It is heavily documented and is intended for use as a template for creating new problems. It is reproduced here as both a guide to how to encode a new problem in LJGP, and as an example of a problem encoded in LJGP.

/task/symbolic_regression/IntTournament.java

```
/**
This is the prototypical example of how LJGP operates.
Extra care has been taken to ensure that every step
is carefully commented and explained.
*/

// all problem definitions are placed in the task package
 package task.symbolic_regression;

import darwin.*;    // Fitness driven selection
import vcpu.*;      // machine code suitable for mutation, etc
import util.*;      // log, stopwatch, other useful classes
import transform.*; // Classes that mutate, crossover, etc. genes.
import task.*;      // Classes that help manage a problem description
import task.base.*; // etc.

public class IntTournament implements
    FitnessMetric,          // interface for fitness algorithms
    TaskBox,                // interface for initializing a run of the GP system.
    java.io.Serializable // allows the class to be saved and sent over the network.
{
```

```
//// main(String args[])
// Launches a run from the command line

public static void main(String args[])
{
Log.global = new Log();  // start a console log
// Log is a utility class that records text info about a run
// A version exists for runs started from an applet that displays to a
// text control. This one just outputs to the console.

Log.global.recordFitnessImprovment(true);
// Rather than commenting out code in several places
// when no longer wanting to display a certain type of
// data about a run, the convention is to always
// send text to the log, and the log decides what
// type of data to display/record, and what to ignore.
// For instance, when reporting an improvement
// in the fitness the best-of-run individual,
// output is sent via Log.global.addFitnessImprovement("...");
// The above line declares that fitness improvements should be displayed

Stopwatch timer = new Stopwatch();
// When tuning performance of an algorithm it can be helpful to know
// how long it took to run. The stopwatch utility class makes this easy

timer.Start(); // start timing NOW!

new IntTournament().execute();
// IntTournament (this class) is a type of TaskBox, which means that to
// initialize the run, you call the constructor, and change any settings
// of interest, and then call execute() to start the calculations.

timer.Pause(); // end timing

Log.global.addString(timer.asString());
// output the amount of time the run took to the log
// since this data is of no particular type uses
// addString, which will always output, unless
// the log is entirely disabled.
 }  // end of main() function

vCPU cpu;

// A vCPU is a class that represents an abstract,
// CPU-like interpreter for use in genetic programming.
// Each individual will be loaded onto this for
// execution during the fitness test.

Tournament evolver;

// Tournament is a type of the SelectionSystem class.  It implements
// standard tournament selection on a steady state population.
// If, for some reason, you didn't want to use steady state populations,
// you could change this class to something else (say, GenerationalTournament)
// and that would change what drives natural selection.
// See the Darwin package for more details.

////  IntTournament()
// constructor, initializes the evolutionary subsystems.
 public IntTournament()
{
Log.global.addStatus("Starting new Tournament");
RandEmit.init(0); // setup the random number generator
//The random number generator accessible via a global static
//function. Seeding it with 0 tells it to query the real time
// clock for its initial value.  Seeding it with another number
// means that the run will always progress in the same fashion.
// Because the random number generator is accessed through a
// globally visible class, seeding it here once is all that is necessary.

evolver = new Tournament();
// this selects the type of individual selection system used
// to drive evolution.

// set the parameters of the tournament-style selection system:
evolver.setSelectionTourSize(4); // size of the tournament when looking for parents
evolver.setReplacementTourSize(4); // size when choosing who to replace with the child
// changes in these values controls the selective pressure of the system –
// larger values result in quicker convergence (for good or bad).

// initialize the vCPU used to interpret/execute the programs evolved:
 cpu = new vCPU(4,  // number of registers
                -1024,    // minimum immediate value
                +1024);  // maximum immediate value

// select the list of instructions that will be inserted into randomly
// generated code:

RandomChoiceFast opcodes = new RandomChoiceFast( 6);
// Random choice fast is a version of the random choice interface; it returns
// elements from an array at random. The "fast" version just selects instructions
// at random, with no explicit weights setting which should be selected most often.

opcodes.add(new SetRegIm() ); // add set (register-dst, immediate) instruction
opcodes.add(new MulRegReg()); // add multiply (register-dst, register-src) instruction
opcodes.add(new AddRegReg()); // add add (register-dst, register-src)
opcodes.add(new SubRegReg()); // add subtract (register-dst, register-src)
opcodes.add(new DivRegReg()); // add division (register-dst, register-src)
opcodes.add(new SetRegReg()); // add set (register-dst, register-src)
// note that if you want to include other instructions in the list of possible opcodes
// here is where you would add them.

RandomVCPUinstruction opcodeEmiter = new RandomVCPUinstruction (cpu, opcodes);
// Since random generation of instructions requires knowledge of the number of
// registers and such, the RandomVCPUinstruction class forms a bridge that
// unites that knowledge in the production of new, random instructions.

evolver.setGeneEmiter(opcodeEmiter);
// Passes the random instruction generator just constructed above
// to the evolutionary engine so it can generate random individuals.

RandomChoiceFast geops = new RandomChoiceFast(5);
// Selects the genetic operators to be applied by the selection system.

geops.add(new GeneticOperatorMutate(opcodeEmiter) );
geops.add(new GeneticOperatorPointInsert(opcodeEmiter) );
// both mutation and point insertion need to have a source of new, randomly
// generated opcodes, hence the passing in of the opcode emitter generated above

geops.add(new GeneticOperatorPointDelete() );
geops.add(new GeneticOperatorCrossover());
geops.add(new GeneticOperatorTwoPointCrossover());
// These operators just modify sequences of opcodes, rather than
// the opcodes themselves, so no random opcode generator is needed.

GeneticOperator.maxGenomeLength = 1000; // specify the maximum program length

evolver.setGeneticOperators(geops);  // hand off the list of genetic operators

evolver.setFitnessMetric(this);
// sets the fitness function (or rather, the class that implements it)

// set the initial population's parameters:
evolver.setPopulationSize(1000);  // number of members of the population
evolver.setInitialGenomeSize(20); // the size of each randomly generated program
```

```
// set the end of run criterion:
evolver.endpoint = new float[2];  // criterions for both measures of fitness
evolver.endpoint[0] = 0;  // no more than  0 error
evolver.endpoint[0] = 20; // no more than 20 instructions

evolver.setGenerations(5000); // set the maximum number of generations for this run

evolver.setNumberOfFitnessReports(50); // How often to report average fitness, etc.
evolver.visualizeRun(false); // toggle: no GUI visuals enabled for this run.
// If a run does use visuals, then it should draw upon the canvas passed in
// to the constructor.
}

//// initRandomPopulation()
// create a random population and calculate its fitness:

 public void initRandomPopulation()
{
evolver.createRandomPopulation();
evolver.calculateAllFitness(); // make sure every member of the population
// has a fitness before calling evolve, as this allows for more efficient
// design of evolve itself.
}

//// finalInit()
// called just before evolution begins, to make sure that
// any parameters that have been changed are applied before
// evolution begins. Usually, creating the initial random population
// is all that needs to happen here.

public void finalInit()
{
initRandomPopulation();
}

//// evolver()
// accessor so the outside world can query the SelectionSystem interface

 public SelectionSystem evolver()    { return evolver; }


//// execute()
// calling the constructor initializes a run. Calling execute() actually runs it
```

```
public Individual execute()
{
finalInit();
return evolver.evolve();
}

//////////////////////////////////
// create fitness cases

// in a real world application, we don't know what the function is.
// in this example, we do - and hence we can let Java do the calculations
// of the actual x,y fitness cases.

static int function (int n)
{
return n + n*n + n*n*n + n*n*n*n;
}


static int fitlist[] =   // fitness cases (input, output) pairs
{0,0,
 1, function(1),
 2, function(2),
 3, function(3),
 4, function(4),
 5, function(5),
 6, function(6),
 7, function(7),
 8, function(8),
 9, function(9)} ;

// These are the data points from which evolution will judge the fitness
// of candidate solutions. If this were a real world problem, instead of
// calling function(N), the known output values from the unknown
// function would be entered.


///////////////////////////
// fitness metric interface

//// classifyRigorous(Individual individual)
// The classifyRigorous function takes an individual, and
// if possible, returns the most detailed (but slowest)
// fitness evaluation possible. In this example, I just
// use the default fitness metric, but one way to use this
// for symbolic regression is to include more fitness
// cases, which only this metric checks.  This way, normally
// the wall clock time is not impacted, but whenever
// you want to check if the program is over-fitting
// the data, you can call classifyRigorous.
// It's a good idea to periodically check for data over-fitting,
// etc, hence the formalization of a function that does it.

public float [] classifyRigorous(Individual individual)  {
    return classify( individual); // just use the normal metric
    }

//// classify(Individual individual)
// the default (and in this case, only) fitness metric:

public float [] classify(Individual individual)
{
float fitness[] = new float[2];
// fitness metrics return arrays of floats. The first value
// in the array has the most significance, and when a
// fitness comparison is applied to two individuals,
// the second value in the array is only considered
// when the first value in each individual's array
// is equal.  This allows for evolution to secondarily
// consider other fitness issues, such as program length.

// In this fitness representation the first value
// in the array is the sum of the error in output value
// for each input value. The second value is the length
// of the program, meaning that if two programs have
// equal error, the shorter one will get selected. If
// short length bias is not desired, make a fitness
// metric that only returns the error in the solution.

cpu.setInstructions( (Instruction[]) individual.genome);
// This loads the vCPU with the instructions stored in
// the individual this function was given to test. Rather
// than creating the CPU from scratch each fitness test
// it is just re-initialized to save time.

// The following code calculates fitness for the individual.
// If reusing this code to encode a different symbolic
// regression problem, or different problem entirely,
// the following, and the fitness case data, are the only parts of
// this file that are guaranteed to need changes.
float totalFitness = 0;  // store the sum of the error
for (int test = 0; test < fitlist.length; test += 2) // the fitness cases store
    {               // one input and one output value, so step through it in 2s.
    cpu.boot(); // initialize the CPU (zero the registers, etc)

    cpu.setReg(0,fitlist[test]); // load the input value into the CPU
    // this sets one of the CPU's registers to the fitness case's input
    // value (the CPU's number of registers is set elsewhere).
    // if multiple inputs were used in this problem, each
    // would be loaded into separate registers

    cpu.runUntilTermination(); // execute all instruction loaded into the vCPU
    // since the opcodes for this program include no loops
    // (or other control code) and the genetic operators are
    // bounded in size) it is safe to just execute all
    // the code. Otherwise, use vCPU.tic() (which executes
    // just one instruction at a time) inside a for loop.

    totalFitness += Math.abs(  Math.abs(fitlist[test+1]) // target
                             - Math.abs(cpu.getRegValue(0))); // actual output
    // Extracts from register 0 the result of the CPU's calculations, and
    // adds the error on this fitness case to the overall tally.
     }

// fitness[] is the array of floats this program returns as its measure
// of the fitness of this individual.
 fitness[0] = totalFitness; // the error between target values and actual value
fitness[1] = individual.genome.length; // the length of the program.

return fitness;
}

//// classifyVisual(Individual individual)
// If a problem has a visual representation showing
// how a program's behavior varies from correct, this
// is the fitness function that displays it (and
// it also returns the fitness classification of the
// individual, just like any other fitness metric).
// no visual output was written for this problem,
// so this function just calls the default metric.
// When LJGP is spawned from an applet, one of the
// GUI objects made available is a Canvas to draw upon
// and a button that toggles on and off visual display
// of fitness evaluation. When on, this function is called.
// When off, the classify(...) function is called.


public float [] classifyVisual(Individual individual)
{
return classify(individual);
}

} // end of class
```

Note: using the encoding here, LJGP finds a successful solution to this problem, on average, in 30 generations. This takes about 4 seconds on a 433 Mhz Celeron using the Symantec 4.0a JRE. See section 10.3 for an example of a vCPU program evolved that solves this problem. See Appendix C for a discussion of different JRE performances with LJGP.

8.2 LJGP packages and classes overview

This section contains a brief overview of the LJGP source code organization. Each package in LJGP is described and the most significant classes are highlighted. A JavaDoc produced collection of 128 documents exists with descriptions of what each class does and why it was designed. The intent of this section, however, is to discuss how the system works in general, rather than providing a complete (and very lengthy) description of the entire system's inner workings.

LJGP consists of three basic packages designed to interface through each other via a well-defined and small set of interfaces that would easily allow each package to be selectively removed and replaced with a different system. Briefly, these three core packages are:

Darwin	The classes in Darwin drive evolution – this is where fitness ratings are used to decide who reproduces and who does not. The **SelectionSystem** class is a generic starting place for any algorithm for choosing who reproduces, who mates, and who dies. By design it leaves the method for selecting who reproduces to subclasses. It provides, however, all the other features needed for evolution, such that new selection strategies can be quickly implemented. The **Tournament** class, for instance, implements tournament style selection by overriding just one function.
Transform	The classes in this package are all genetic operators – given an array of any type of Java objects, they perform crossover, mutation, etc., upon that array. Classes are split into two hierarchies – those that derive from **GeneticOperatorDual** (and take two parents as input) and those that derive from **GeneticOperatorSingle** (and take only one parent).

vCPU	The vCPU package simulates a GP-friendly CPU that is not particularly modeled after any specific CPU. It implements a generic RISC architecture with a user specifiable number of registers (as low as 1, and as high as will fit in memory). The **vCPU** class does bookkeeping chores (keeping track of registers, etc). Instructions are added by deriving from the appropriate class; depending on what arguments the instruction takes (**InstructionRegReg**, for instance, is used as the basis for all instructions that take two registers as input). See next section for a list of the instructions implemented, and an example program.

The following packages perform periphery functions, such as network support and run visualization. While not core to the operation of LJGP, they make it a more useful and flexible system.

Task	Different problems encoded for LJGP are stored in the Task package by convention. Two example problems are encoded: *symbolic regression* and *grazing* (a variation on Koza's lawnmower problem that models the constraints of the real world more closely). The **task.base** sub-package includes utility classes that can be used to extend how a given problem run is performed. The **MailRun** class executes a run and then sends the results to a different host via email. The **RunSource** class allows programs to be manually typed in as source code and evaluated using the same fitness metrics used during normal runs.
Util	These classes exist to aid other parts of the system, but are general purpose enough to be used in many different places. Some of the most used classes included here are the **Log** class (for controlling what information is recorded during the run, the **Mail** class (which sends mail via SMTP), and the randomization classes (**RandEmit**, **RandomChoice**, etc) for returning random numbers and random elements of arrays.
World	World is a helpful package that contains a basic grid-world implementation, which includes useful functions for the simulation of a grid world, as well as functions for drawing the state of the grid-world on the screen.
GPnet	This package allows a problem already encoded in LJGP's **TaskBox** class to be stored on a server and sent between the client and server as appropriate. Currently, this is used to back up partially complete runs in case the client is shut down mid-run. In that event, the partially completed run can be farmed out to another machine for completion. Also included

	are the network functions necessary to implement demes spread across a collection of machines, though no deme-aware evolutionary system has been implemented yet.

8.3 vCPU programs

This section is designed to help the researcher read vCPU programs, understand how they work, and write new instructions. It provides an overview of the basic vCPU instructions, some annotated vCPU programs, and a heavily annotated implementation of the increment function as examples.

8.3.1 vCPU instructions

Similar to instructions in a RISC CPU, most vCPU instructions operate upon register contents, not immediate values. When a constant value is needed for an instruction that only accesses registers (say to add 4 to a register) the constant must first be loaded in a separate register with the Set command.

The vCPU package implements a core set of mathematical instructions. New problem encodings are free to add or remove from the list. The complete list is as follows:

Opcode	(R = register, I = Immediate value)
Set R I	Set the contents of register R to the immediate value I.
Set R1 R2	Set the contents of R1 equal to the contents of R2
Multiply R1 R2	Multiply the contents of R1 and R2, and store the result in R1
Add R1 R2	Add the contents of R1 and R2, and store the result in R1
Subtract R1 R2	Subtract the contents of R2 from R1, and store the result in R1
Divided R1 R2	Divide the contents of R1 by R2, and store the result in R1 (or zero, if R2=0)
LessThanOrEqual R I	If the value in R is less than I, execute the next instruction, otherwise skip to the next one

8.3.2 vCPU encoding of $x^2 + 2x$

This example calculates $\mathbf{x^2 + 2x}$, given the value of **x** in register 0, and places the result in register 1.

Instruction	Description
Set R1 R0	Backup value of x in R1
Multiply R1 R1	Square R1 (x) and store the result (x^2) in R1
Set R3 2	Stores the constant value 2 in register R3

Multiply R1 R3	Multiply R1 (x) times R3 (2), and store the result (2x) in R1
Add R1 R0	Adds R1 (x^2) and R0 (2x) and stores the result (x^2 + 2x) in R1

This encoding is roughly as complex in structure and number of instructions/functions as the Lisp encoding of the same function: **(+ (* x x) (* 2 x))**. Which one evolution would favor, of course, is an open question, and would be highly dependant upon the dynamics of the problem being attempted, as well as other implementation details of the genetic programming systems used.

8.3.3 vCPU encoding of $x^4 + x^3 + x^2 + x$

This function can be encoded in the straightforward manner that a human would probably produce if given this problem, where **x** is placed in four separate registers, the correct number of multiplications is applied to each register, and then all registers are added together. While conceptually simple, this ignores the flexibility allowed by a register system. A much shorter sequence of instructions was found by evolution that makes multiple use of partial results. Starting with the value of **x** in register 0:

Instruction	Description
Set R2 R0	Copy R0 (x) to R2
Multiply R2 R2	Square R2 (x)
Add R0 R2	Add R0 (x) to R2 (x^2) and store the result in R0
Set R3 R0	Copy R0 (x^2 + x) to R3
Multiply R0 R2	Multiply R0 (x^2 + x) times R2 (x^2) and store the result in R0
Add R0 R3	Add R0 ($x^4 + x^3$) to R3 (x^2 + x) and store the result in R0

By having access to partial results such as **(x^2 + x)** and **(x^2)**, vCPU code can exploit the redundancy in this mathematical function, and calculate the solution with just 6 instructions. Without some way to store intermediate values, the best Lisp encoding for this problem is **(+ (* x x x x) (* x x x) (* x x) (* x))**, which is considerably longer. Again, which one evolution would favor is an open question, and would be dependant upon the dynamics of the particular implementation of genetic programming used.

8.3.4 Adding a new instruction to vCPU

Writing a new instruction for the **vCPU** is meant to be very simple. Most of the functionality is encapsulated in the base class **Instruction** (and its subclasses). The subclasses are organized by the arguments they take. Instructions that operate on one register should inherit from **InstructionReg**. Likewise, instructions that require two registers inherit from **InstructionRegReg** , and instructions that take a register and an immediate constant argument inherit from **InstructionRegIm**. Every instance of an instruction is a separate instance of class that defines that instruction. The member variables of that class specify what registers (and immediate values, if any) the instruction operates upon.

The following complete implementation of the increment instruction (which adds 1 to the value stored in a register) is defined in /vcpu/IncReg.java. This file is heavily documented and is intended for use as a template for creating new instructions. It is reproduced here as both documentation for making a new instruction and as an example thereof.

/vcpu/IncReg.java

```
package vcpu;  // instructions can be part of any package, but this one is stored here.
// That is because it is a general purpose instruction that
// might be useful for many different problems, rather than just
// for a specific problem.

/**
Description: Adds one to the target register.
*/

// Since this instruction takes one argument,
// it inherits from the base class of all
// one argument instructions: InstructionReg

 public class IncReg extends vcpu.InstructionReg
{
////////////// name()
// Every instruction needs a textual representation to be
// used when reading in source from a file, or printing out
// a series of instructions as source code.
 public String name()
   {
    return "Inc";
   }

//////// run(vCPU cpu)
// The run function is called every time the vCPU executes
// this instruction. It is where the functionality of an instruction is
/ implemented.  Passed in is the vCPU that this instruction
// is to extract its input from, and send its result to.

 public void run(vCPU cpu)
```

```
  {
// setReg and getReg are accessor functions for the CPU's registers.

 cpu.setReg(reg, 1 + cpu.getRegValue(reg));

// The member variable reg comes from the superclass.
// It is set at random when this instruction is generated to
// the register that this instruction will modify.
// If this instruction took two registers, run() would access them
// via reg and reg2. Both variables would also already
// have been set to random targets by the code that creates
// instances of newly initialized instructions.
  }

} // end of class
```

Note that this only defines the instruction. In order for it to be used during random code generation it must be added to the opcode list. This would be done with the following line of code:

InstanceOfRandomChoiceThatContainsOpcodes.add(new IncReg ());

See 10.1 for an actual example of adding instructions to the opcode list (and how to create the opcode list in the first place).

9 LJGP Applied

In the following chapter, my initial experiments with LJGP are recounted. Far from being carefully controlled and considered experiments meant to settle theoretical issues decisively, this work can be characterized as many pilot studies in the effectiveness of LJGP and the possibility of evolving interesting programs with it. It is included here to demonstrate the preparatory/exploratory work done.

9.1 Lawnmower pilot study

The lawnmower problem has already been investigated by Koza in GP II (1994). In his version, however, the problem was highly abstracted from the nature of the task as known in reality. Namely, in Koza's work, the world was toroidal, the agent always started in the same location, and the agent could teleport at any time from one location in the grid to another. These features no doubt make the problem easier to solve, but at the expense of any hope of building general-purpose agents that could navigate in the real world.

Of course a grid world is very different from the continuous, noisily sensed world of modern robotics. Any programs evolved in a grid world would be rather far from effective robotics controllers, even if the other abstractions were more accurate. Nevertheless, navigation of a grid world does present many of the issues that real world navigation does, namely the efficient discovery of the world’s layout, and the efficient coverage thereof without excessive backtracking. From this viewpoint, the abstraction to a grid world removes some factors that increase the difficulty of evolving solutions, but not those critical to making the problem interesting or representative of some of the essential issues. Abstraction, however, is a debated issue and the usefulness of experiments that involve grid worlds is not something all researchers would agree with. That debate I will leave to others.

The lawnmower problem is also interesting from a cognitive science viewpoint, in that it abstractly relates to the behavior of grazing creatures and anything that eats an abundant, nonmoving food source. While there is little reason to think that algorithm used by the most fit evolved individuals from this GP run is the same as the algorithm used by creatures of the real world, the sorts of solutions found might provide inspiration about the cognitive capacities needed to efficiently forage.

9.2 Problem description

The goal is to evolve a program that cuts all the grass in a simulated lawn world that has enough similarity to the real world to make the problem interesting, difficult, *and* useful as a tool to think about the nature of intelligent and/or fit behavior.

Every agent is tested in a randomly created grid world. The grid is a 10x10 square, with three types of terrain: uncut grass, cut grass, and rocks. When the mower enters a square it cuts the grass, if any, automatically. The mower cannot, however, move into a square with rocks, though there is no penalty for attempting (the agent just does not move).

The layout of the world is created at the beginning of the GP run, with 10 rocks placed randomly within the grid. At the beginning of each fitness test, the cut grass is returned to the uncut state, but the rocks remain in the same place. To promote generalization, the mower's starting location is set randomly at the start of each fitness test. Fitness is computed by summing the number of uncut squares of grass at the end of a fixed number of virtual CPU cycles (200, in the first few runs). To increase the reliability of fitness ratings, and to guard against lucky initial starting locations, multiple fitness trials (two, in the first few runs) are run and averaged together to find the fitness of an individual.

9.3 The genetic makeup of an individual

An individual is a linear, variable length list of virtual CPU opcodes. During each CPU tick a single instruction is executed, and the results, if any, are stored in a register specified by the opcode. Some of the opcodes instruct the agent with an action to perform, such as turning to the left, or moving forward.

In this set of runs a 4 Register Virtual CPU was used, with random immediate values ranging between -2 and +6.

Basic Instructions

Opcode	(R = register, I = Immediate value)
Set R I	Set the contents of register R to the immediate value I.
Set R1 R2	Set the contents of R1 equal to the contents of R2
Multiply R1 R2	Multiply the contents of R1 and R2, and store the result in R1
Add R1 R2	Add the contents of R1 and R2, and store the result in R1
Subtract R1 R2	Subtract the contents of R2 from R1, and store the result in R1
Divided R1 R2	Divide the contents of R1 by R2, and store the result in R1 (or zero, if R2=0)
LessThanOrEqual R , I	If the value in R is less than I, execute the next instruction, otherwise skip to the next one

Domain Specific Instructions

Instruction	Description
GoFoward	Move the agent forward one unit on the grid in the direction the agent is currently pointing.
GoLeft	Shift the direction the agent is currently pointing to the left (i.e., north becomes west).
GoRight	Shift the direction the agent is currently pointing to the right

	(i.e., south becomes west).
SenseAhead	Retrieve an integer representation of the grid in the direction the agent is facing (0 = cut grass, 1 = grass, 2 = rocks).

9.4 The mechanics of evolution

Evolution progresses via the standard LJGP tournament system. Parent programs are selected by drawing several members of the population at random, and selecting the best of that sample as the parent. A form of genetic transformation is selected at random from the list below, and applied to the parent program(s). The resulting child is then tested for fitness and inserted back into the population, replacing the worst member of a small, randomly selected sample of programs. These mechanics insure that even the less fit members of the population have a chance to reproduce, so that local minima do not trap the entire population in a sub-par solution.

Mutation	Random point mutation
PointInsertion	Random insertion of a single instruction
PointDelete	Random deletion of a instruction
OnePointCrossover	Produces a child with everything to the left of a random point in the first genome, and everything to the right of a random point in the second genome.

9.5 Pilot runs of the lawnmower problem

These parameters were used initially and continued to be used unchanged until specifically noted in the text.

Population Size	5000
Initial Genome Size	20 randomly selected instructions, evenly weighted from the list above
Selection Tournament Size	4 (the number of randomly selected programs from which the best will be designated a parent)
Replacement Tournament Size	4 (the number of randomly selected programs, from which the worst will be replaced by a new child).
Run length	1000 generations (a generation is defined N(=population size) cycles of selection, mutation, replacement.

9.5.1 The first run (trial 1)

In the first trial agents were placed in the world and tested twice, each time starting in a random location. For each test they were given 200 cycles in which they could move, turn, sense, or do math. After 1000 generations (1000 * 5000 tournaments), each run was halted. The number of uncut areas of the grid was used as fitness.

The hope was that the agents would learn to use their sensor to their advantage, and react to their environment, rather than moving randomly. I've defined reactive use as making use of a value derived from a **SenseAhead** command within the space of two move forward instructions, and any number of other instructions. Within these constraints the information returned by the **SenseAhead** command still tells the agent something about the world. Once it moves any farther, however, the data returned no longer represents anything about the current part of the world the agent is in.

A typical program from the first run (italic lines have no effect on behavior and could functionally be replaced by **noop**s). This program was given a fitness of 17

```
Go Forward
Multiply r0 r1
Go Forward
Go Forward
Go Forward
Go Forward
Turn Right
Sense Ahead
Go Forward
Go Forward
Set r1 r1
Turn Left
Go Forward
Go Forward
Turn Left
```

The result, out of 76 runs, was that none of the individuals made any reactive use of their sensors. That is to say, the best individuals of all 76 runs did not base any of their behavior upon the values returned by the sense ahead within a short enough time for the values to be applicable. Indeed, out of 76 best-of-run individuals, 28 made no use of the **SenseAhead** instruction at all. More over, the non-sensing programs were more fit than the other individuals, (average fitness of 12.5, as opposed to 13.6). On consideration, this is not too surprising - if sensing is not affecting behavior in useful ways, it just wastes instructions that could be better used producing a longer random walk.

In summary, no intelligent, reactive controllers evolved. Even the most fit individuals were non-reactive. And even the most fit individuals wasted instructions on mathematical calculations that did not affect behavior, as in the example program above. One of two possible causes are suggested by this:

1. The fitness metric is too noisy. Each new program is tested twice, which could mean that the most fit-rated individuals are those that had lucky runs, rather than more successful programs. A program that is slightly better than average might not distinguish itself against all the lesser agents who happened to have lucky runs.

2. The GP run is not given enough generations to find a more efficient program, or another parameter of the run needs to be changed, such as the percentage of crossover to mutation.

After this one run, an interesting code representation issue becomes apparent. Getting two instructions together is hard. That is to say, when two instructions must exist in a fixed order to be useful (such as a **SenseAhead**, followed by **Is LTE?**), a lot of randomly created and permutated programs will only have those instructions in the wrong order or too far apart. Or even will have them in the right order, but use different registers so that one does not communicate with the other. This is one clear tradeoff between a linear, register-based system, and a tree system. In a multiregister linear program, it is easy for a program to store a value for later use, something a tree system cannot do without variables. It is also very easy, however, to store a useful value and then and never make use of it later on.

Reducing the number of registers to the bare minimum would help, but is a tradeoff. One goal of registers is to allow the storage of useful data for later use, and with fewer registers, less data can be stored. Another possible solution is to make larger instructions that perform the duties that currently require multiple instructions. This is certainly the methodology that Koza used in GP I, where most functions are very high level. However, as the function set becomes less and less general, and more tailored, the number of unique possible solutions shrinks. In this domain, for instance, it might be nice to make conditional choices based upon knowledge of your own past behavior, and not just based on knowledge of your sensors (which are, after all, quite limited). If the conditional instruction only considered the status of the grid in front of the agent, such higher level cognitive processing would be impossible.

Another issue that highly tailored instruction sets bypass are constant values that are rarely in the right range. It is easy to specify the appropriate range of values that might be assigned to a given instruction when that instruction has no side effects. However, as soon as combinations of instructions interact there is no clear and easy way to keep constants in the right range between instructions.

For instance, the **SenseAhead** function returns a number between 0 and 4, depending on what type of terrain is in front of the agent. A conditional that makes use of this input should compare those values to constants that fall within that range so that behavior varies based upon changes in the environment. If the constant does not split the set of possible inputs, then the conditional is nothing but a constant goto. **If LTE? n, 100** always evaluates to true, for all **n** < 100 (e.g., all **n** returned by **SenseAhead**).

Now, in this example, one simple solution would be to limit the constants used for LTE to the range -1 to 3 (the minimal range that bounds the values returned by **SenseAhead**). That, however, limits the types/ranges of conditionals that the instruction set can create for internal control structures not reacting directly to sensor values. The goal of using basic instructions over tailored instructions is to allow for creativity beyond what the programmer specifically imagines. If conditionals are limited to a range such that they only are useful reasoning about with values from a sensor, however, then there is no reason not to just combine the two opcodes.

For now, however, there is no clear way to address this problem other than to hope that evolution can overcome it with enough generations and a large enough population size.

9.5.2 Trial 2

Trial 2 addresses the issue of a possibly noisy fitness function from among the hypothesized flaws in the previous run. Instead of just testing each new agent with two randomly selected initial starting points, the number of runs was increased to ten. All other parameters were unchanged. Out of 42 unique runs of 1000 generations, the average fitness (14.6) was worse as compared to the first run (13.3). This is almost certainly because of the much more stringent fitness test, however, since mowing ability must be maintained over ten trials, instead of just two. The real question of interest was not the raw fitness values, but the nature of the programs evolved.

Again, none of the evolved controllers made intelligent use of the ability to sense (using the same metric of "intelligence" defined during the discussion of Trial 1). Notably, the average fitness of the programs that use sensing (14.4) is almost the same as those that do not (15). In **all** cases, the conditionals / sensor pairs could have been replaced by groups of no-op instructions without changing the functional behavior of the agents. Again, this suggests that fitness is noisy, since a truly fit program should have no wasted opcodes.

Another possibility, though, is that not enough cycles of evolution were run to find the optimum solution (be it a random walk or not). For instance, one agent, whose fitness was 8 (almost the best of the run), had a length of 53, 9 instructions of which had no effect (i.e., were replaceable with no-ops). If the GP system were doing its job, those useless instructions should have been removed in favor of making the random walk longer.

9.5.3 Trial 3: ten tests, more generations (5000)

In trial 3 everything was kept as in trial 2, except the number of generations was increased from 1000 to 5000 in the hope that the controllers would evolve more fit solutions (longer random walks, or intelligent reactive behavior). The average fitness after 77 runs was 13.4 (as compared to 14.6 for trial 2). While this is an improvement, the programs produced are not that different from those evolved in the previous trial. With one exception detailed below, none of them use sensors reactively, and, in all cases, the evolved program code contains instructions that could functionally be replaced by no-ops.

The intelligent program?

Out of 77 best-of-run programs, the one at right was the only one to make reactive use of its sensors. Out of a 45 instruction program, however, only lines 16-18 are reactive, meaning that the amount of improvement in fitness they could impart is low.

Also note that this program, which is more fit than the average program, still includes many functionally non-active instructions (shown in italics).

One possible issue with LJGP that this program brings up is the performance of the crossover operator:

Perhaps the lack of propagation of lines 16 & 17 is a sign of failure of the crossover operator to copy this useful fragment and insert it elsewhere.

Fitness: 11.8

1. Go Forward
2. Turn Right
3. Go Forward
4. Go Forward
5. Go Forward
6. Go Forward
7. Go Forward
8. *Is LT/E? r1 5*
9. *Set r1 r3*
10. Go Forward
11. Go Forward
12. *Is LT/E? r2 -2*
13. *Set r0 0*
14. Turn Left
15. Go Forward
16. Sense Ahead
17. Is LT/E? r0 0
18. Go Forward
19. *Subtract r1 r3*
20. Turn Left
21. Go Forward
22. Go Forward
23. Go Forward
24. *Set r1 2*
25. Go Forward
26. Go Forward
27. Go Forward
28. *Set r1 -1*
29. *Is LT/E? r0 3*
30. *Sense Ahead*
31. Turn Left
32. *Subtract r2 r0*
33. Go Forward
34. Go Forward
35. Go Forward
36. Go Forward
37. *Is LT/E? r1 4*
38. Turn Right
39. Turn Left
40. *Add r0 r1*
41. Go Forward
42. Go Forward
43. Go Forward
44. Turn Right

9.5.4 Trial 4: better crossover

In this trial all settings were the same as in trial 3, but with an added form of crossover that used two points instead of one. With this version, a fragment is cut out of a individual and replaced by a similarly excised fragment from another parent. The weight given to crossover also increased, since each type of crossover now had a 1/5 chance of being selected (2/5 total), as compared to the 1/4 weight given one point crossover used in earlier trials.

The result of this change was that in 185 runs, the average fitness reached 6.9, which was a big improvement over the previous trial (14.6). The 85 agents that used the **SenseAhead** averaged 6.7 fitness, compared to 7.1 fitness for the agents that did not use **SenseAhead**. In addition, the 30 agents that had the most instances of **SenseAhead** in their programs had an average fitness of 5.9, suggesting now that sensing was useful.

Actually, however, this last result is a byproduct of the fact that larger programs were usually more fit than average (the 30 largest program had an average fitness of 4.8), and that the largest programs usually had the most instances of the **SenseAhead** instruction. Indeed, the 30 largest programs and the 30 fittest programs contained a lot of overlap. The two sets were not identical, however, - the 30 most fit programs had an even better average fitness of 3.2.

Interestingly enough, non-behavior causing instructions continued to be prevalent. The 30 agents that had the most instances of useless instructions (by percent) averaged a fitness of 6.8, whereas the more "efficient" programs averaged 8.55! How could this be? It appears that having many instances of useless instructions is a byproduct of having a big program. Since being big is correlated with the best fitness, having lots of useless instructions is also correlated, even though those instructions do not contribute to fitness.

So why do the large programs have higher fitness? Perhaps because their behavior is more random. Short programs loop back to the start when they reach the last instruction in their code. This means that short programs are likely to cause the agent to enter a cyclic pattern. Cyclic movement usually mows the same terrain repetitively, without any escape into unmoved areas of the grid. This characteristic might explain the patterns of large programs (which do not loop) doing better: they are less likely to enter repetitive movement patterns.

It is worth pointing out that when the four very best programs are examined, the number of wasted instructions between them is fairly low, (only 15 out of 200, in three cases, and 0 out of 75 in one other). So perhaps wasted instructions do impair fitness, but so little that it does not pressure the whole population into solutions without them.

The conclusion, from all these trials is that while the parameters of evolution have not yet been tuned such that every run finds the most efficient random behaving program, tweaking settings has very little likelihood of success if the goal is evolving reactive behavior instead of random behavior. Consider:

1. 200 cycles probably does not leave enough time to think and move
2. In over 300 runs not a single best of run individual made even slightly consistent reactive use of its sensors, supporting conclusion 1.

Since the goal of these pilot studies (beyond experimenting with LJGP) is to evolve intelligent (that is, reactive) behavior, the next trials use a constrained version of the lawnmower problem in an attempt to make reactive behavior more possible and more rewarding than random movement.

9.6 Grazer pilot study

The lesson from the mowing/lawnmower task was that random behavior was sufficient to satisfy the requirements of the task. The results from those trials are interesting material for analysis in several domains:

- philosophical - the nature of intelligent behavior (random walks are effective, and are an easily evolved foraging strategy).
- functional fixedness - the experimenter did not foresee the efficiency of random behavior for this task because most computer science algorithms do not rely on randomness.
- task analysis - unbiased exploration of the solution space was made possible by the use of GP, producing results a human could have easily duplicated by explicit programming, but probably would not have.

It was, however, disappointing from the viewpoint of dissecting the inner cognitive capacities utilized by the evolved intelligence. The nature of the task, therefore, was changed slightly with the hope that a new constraint would make intelligent behavior more likely to evolve.

In the new task (called grazing), everything is as before, except that the mower takes on a biologically inspired limitation. Instead of all actions having the same metabolic cost, movement now has a significant higher expense to the agent. This expense is manifested simply as a limit of N moves per fitness test - any more and the agent is considered to have run out of energy. This means each agent now faces two limits - time (number of opcodes executed per run), and movement (the number of movement opcodes executed) within that span of time. The number of movement opcodes allowed to execute is lower than the limit on the total number of opcodes (movement or not) each agent can execute. This means that the agent can spend some time doing calculations, and still be able to move as many times as an agent who only includes movement instructions.

The other change, based upon the results from the mowing problem, was to increase the number of opcodes executed per run (i.e., to allow more time for thought/calculations). The sum of these changes was that agents now execute up to 400 opcodes per test, but only the first 150 instances of the **GoForward** instructions encountered. To keep the wall-clock time used per run reasonable, the number of fitness tests per fitness evaluation was reduced from 10 to 5.

9.6.1 Grazing trial 1

No significantly intelligent (reactive) solution evolved. The average fitness was 5.0 (out of 133 runs), as compared to 6.9 for the last run of the mower problem. These fitnesses values cannot really be compared, however, because the constraints of the problem have changed (as has the number of tests taken to measure fitness) even if the same metric (squares mowed) remains. Nonetheless, it is surprising that the average fitness is better on this problem, since it seems that the grazing problem is harder than the mowing problem. Perhaps this change results from the reduced number of fitness cases per fitness test. Consider the performance of the best of the best-of-run individuals from this trial:

Best of run individual ranking	Size of programs	Times the sense command was used	Useless instructions	Fitness rating (average # of blocks uncut per trial)
1	116	7	50	0
2	77	0	35	0
3	138	9	53	0.4
4	101	5	43	0.4
5	44	1	18	1
6	401	32	157	1
7	72	2	24	1
8	101	6	36	1

While there is no penalty for spending some time sensing (since only 150 move instructions can be issued out of 400 total per trial) it is notable that the individual #2 uses no sensing commands at all. Even if an argument could be made that planning is impossible in this domain of limited sensing, it is clear that attempting to move into an inaccessible square is never a fit strategy. Yet one of the two perfectly fit individuals did not even attempt to avoid this. Since a purely random, non-reactive agent can achieve the best fitness the metric can assign, it is clear that the evolutionary conditions are not appropriate for producing generally successful agents.

It now seems likely that the fitness of the best individuals results from lucky combinations of 5 fitness tests at a time, and that perhaps decreasing the number of tests to 10 was too severe.

Other notable trends from this table are:

- The number of useless instructions is now very high. Of course, the fitness metric does not punish individuals with useless instructions as long as 150 move instructions are executed, so it is unsurprising that the resulting individuals are even less concise than those from the mowing problem.
- The number of sensing instructions does not correlate well with fitness, but only with program length.

All the data suggests that the fitness metric needs to be more rigorous.

9.6.2 Grazing trial 1 part 2

The next trial attempted to determine how much random variation was added to the fitness when the number of trials was reduced to 5 per fitness test.

In this run, all parameters were kept the same as before. At the end of each run however, the best-of-run individual was evaluated again for a more accurate fitness test, where everything was held constant as before, but the fitness metric sampled 10 starting locations instead of 5. This end-of-run fitness will be called the rigorous fitness test.

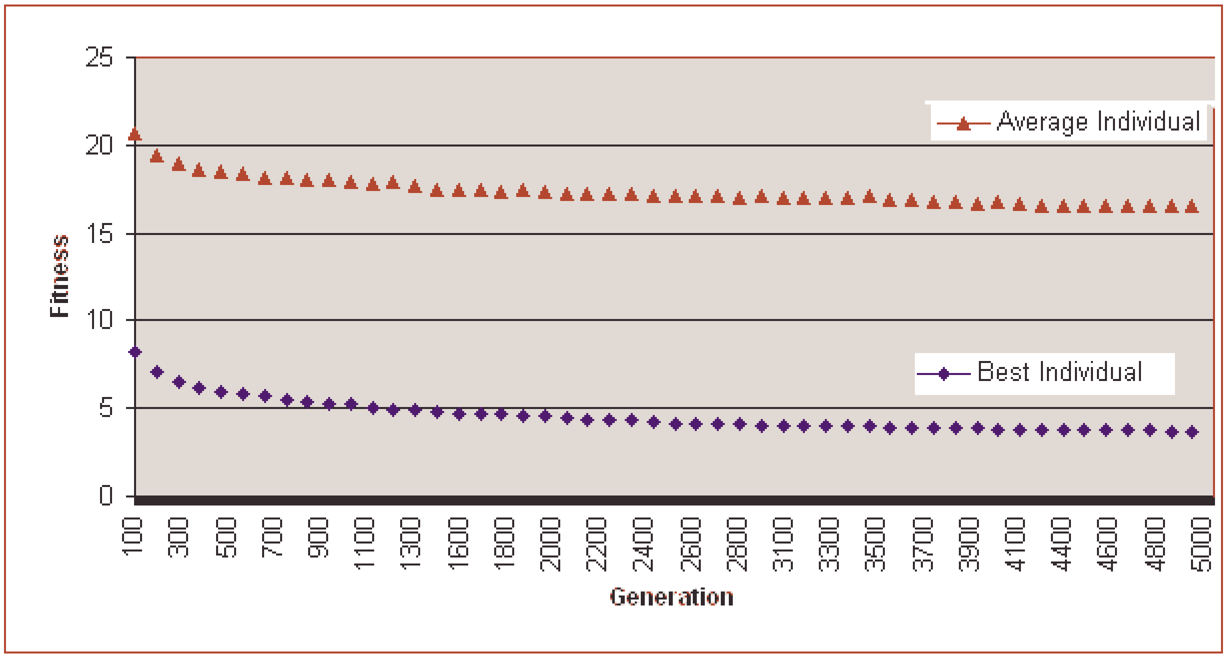
The result, out of 126 trials, was a normal fitness average of 4.7 (similar to the last run), but the rigorous fitness was 10.5 (an average change of 5.8 between normal fitness and rigorous fitness). Clearly, the better fitness of the first grazing trial compared to the last mowing trial was the result of luck.

When re-evaluated with 10 tests, the fitness was even worse than that found when running the mowing problem. These two collections of fitnesses may not be comparable, however, since the best-of-run individuals (and their fitness) come for a large set of fitness tests, leaving more room for an exceptionally lucky test producing the best fitness value seen throughout the run. The rigorous fitness, since only applied once, probably displays regression to the mean.

9.6.3 Grazing trial 1 part 3

It seems possible that fitness metric noise is the cause of the lack of evolution, but other factors could contribute. For instance, fit individuals might not be propagating within the population. In this trial all parameters were kept the same as above, but multiple recordings of fitness were taken as evolution progressed. The values recorded were from the standard fitness test, since the goal was to evaluate the selection system's dynamics.

The following chart shows how fitness changed, on average, from the start of a trial to the end. Data shown are averaged values from 32 unique runs.



Notable trends:

- Most of the changed happens within the first 1000 generations.
- The difference between the average individual and the best individual moves in lockstep over time.

Thus it appears that while better individuals do propagate, the rate is slow enough that many low fitness individuals continue to survive. Whether this is beneficial (preserving diversity and impeding premature convergence), or not (slowing progress) is unclear at this time. Since the fitness trend is clearly downward (and this is true for the entire population as well as the most fit individual at each time step), it seems likely that the evolutionary pressure is sufficient, for now. Once the fitness metric is stronger, this issue will be worth more fine-tuning.

9.6.4 Grazing trial 2

In this run, the question of interest was how strengthening the fitness metric would change the evolutionary end product. Therefore, run parameters were as before with only two changes. First, the standard fitness metric samples fitness 20 times per new individual, each time starting the agent at a new location in the world. Since this means the normal number of fitness tests is more than the rigorous fitness metric used before (10), the rigorous metric now samples fitness 100 times. Unfortunately, this means this run also cannot be measured against pervious runs. Clearly, if the rigorous fitness metric concept is to be useful for comparison between runs, the nature of the metric must be decided early and held constant. The easiest way to achieve this, therefore, is to test an individual in the most demanding fashion possible when the rigorous metric is called, within reasonable time constraints.

In this case, the most demanding, but reasonably fast metric was to run the individual from 100 random starting places, the same as the number of squares in the grid. Perhaps an even better system would be to insure that the agent actually started once in every grid space, rather than relying on the uniqueness of the random generator to approximate that result. This was not considered, however, at the time the rigorous metric was designed. Out of the desire to keep the rigorous metric constant between this and further runs, it was not changed hence.

The result of strengthening the fitness metric was that out of 106 runs, the average fitness of the best-of-run individual (as measured during evolution, with 20 fitness trials) was 5.7. The average fitness as recalculated rigorously (100 fitness trials) for the best-of-run individuals was 8.4 (average difference between metrics, 2.7). There are two notable trends:

1. The average fitness in this test as measured by the standard fitness test (20 trials) is worse than for the first run of the grazer problem, (fitness trials = 5). This is not surprising since more intensive testing should eliminate more lucky runs.
2. For this run the average rigorous fitness (8.4) is better than the rigorous test in the first grazer run (10.5), even though this "rigorous" measure of fitness used more tests (100 verse 10). This is in contrast to trend #1. The agents evolved in this run are better at more rigorous fitness tests than those of the first run, even when the tests are much (10x) more rigorous. Conclusion: the selective pressure of just 5 tests per fitness was allowing too much room for lucky tests, and weakened the ability of evolution. Thus, the rigorous metric has exposed that changing the number of fitness tests used during evolution does have an effect on the true fitness of the individuals evolved.
3. 20 tests still does not capture the fitness of an individual in an unbiased form. In all cases, the rigorous fitness test found that the agent was actually worse than the quicker test reported.

The agents produced in this trial of the grazer problem are better grazers, but at what cost? It takes 4 times as long to complete a single run. The first trial (parts 1,2 and 3), converged upon 0 fitness in much shorter time. Those agents with 0 fitness, however, appeared to have good fitness as a product of luck, so running the system 4 times as long would probably just have produced four times more lucky individuals.

Unexamined up until now, however, is whether the best of the agents evolved in the second run display any better design. Are they also just a product of luck?

A survey of the all the programs evolved by Trial 2 shows very little productive use of sensing is made. Only six out of the 106 runs, in fact, made reactive use of SenseAhead, and those made only sparing use even then. Those programs also did not have particularly better than average fitness. Again, it seems that random behavior is most fit for this domain. Does this vary between metrics? The following table ranks the top 10% of the best-of-run individuals for further consideration of this issue:

Fitness as sorted by rigorous metric

Size of programs	times the sense command was used	useless instructions	fitness rating, standard	**fitness rating, rigorous**	delta (rigorous - standard)
401	22	166	0	**0.38**	0.38
144	14	49	0.45	**1.13**	0.68
142	11	51	1.75	**1.97**	0.22
83	0	35	1.1	**2.15**	1.05
155	7	61	1.05	**2.61**	1.56
350	37	123	2.15	**2.86**	0.71
401	20	176	1.3	**3.02**	1.72
305	16	147	2.55	**3.16**	0.61
131	8	54	1.4	**3.27**	1.87
74	5	26	1.55	**3.29**	1.74

Fitness as sorted by standard metric

size of programs	times the sense command was used	useless instructions	**fitness rating, standard**	fitness rating, rigorous	Delta (rigorous - standard)
401	22	166	**0**	0.38	0.38
144	14	49	**0.45**	1.13	0.68
155	7	61	**1.05**	2.61	1.56
83	0	35	**1.1**	2.15	1.05
401	20	176	**1.3**	3.02	1.72
131	8	54	**1.4**	3.27	1.87
74	5	26	**1.55**	3.29	1.74
123	6	53	**1.65**	4.35	2.7
142	11	51	**1.75**	1.97	0.22
285	12	122	**1.95**	3.6	1.65

Background shading signifies the instance of identical individuals in both tables.

The first notable trend is that by both the rigorous metric and standard metric, the top 10% of the trial is roughly the same in terms of membership. The exact order changes, in a few cases, but not for the two most fit members. Also notable is that the number of useless instructions seems to have little effect on the fitness, rigorous test or no, and that in most cases, useless instructions make up a full third of the instruction set.

The results seem to indicate that, even when faced with a more rigorous fitness metric, the best behavior is one that is fully random, and that it is not hard to achieve such behavior. Oddly enough, most fit programs are very large (more so than average), suggesting that having more instructions makes the agent perform more randomly. Why, then, the dead code? Could not the agent behave even more randomly if all instructions produced overt behavior? Perhaps the code deemed "useless" because it does not directly react to the sensed world, actually exists to make behavior more random.

This could be tested with the creation of a post processor, which contains one genetic operator: point deletion. With a population of one, and a hill climbing selection system, this processor will delete instructions randomly, and will keep the result only if it is as fit, or more than the previous individual. Only those instructions that contribute to fitness will remain after such a post-processor is applied.

9.6.5 The surprise

In preparation for building the tool described above, a tool was built to re-evaluate programs from source code. When the most fit programs found by the last run were re-evaluated without any changes, their performance was much, much worse.

The only difference between evaluations stems from the world agents were evaluated in: for each run of the problem, a new layout of the grid was randomly created and then used for fitness tests for the rest of the run. So, when the source code was re-evaluated, the agents were tested for the first time in a world they did not evolve in.

The change in performance of these supposedly fit programs was dramatic. See the table at right for a comparison of the top ten grazers from the last run and the their re-evaluated fitness (plus, in bold, three not-so-good grazers from the run, for comparison).

rigorous fitness rating from last run	rigorous fitness rating, on second evaluation
0.38	32
1.13	33
1.97	32
2.15	33
2.61	35
2.86	33
3.02	31
3.16	38
3.27	38
3.29	28
14.8	**32**
15.7	**46**
18.88	**43**

Clearly, the fitness metric as used in the last run was not promoting generalized grazers. Rather, evolution found grazers tuned to their environment, in such a way that even though they started from 100 random locations, the path they followed reached a fixed point early on, and from there followed a path specifically designed to canvas the grid without needing any sort of reactive use of sensors.

When evaluated in a new world, this strategy provided no help, and movement became a random walk. It is notable that even significantly worse agents, as judged by the fitness assessed in trial 2, are relatively similar in fitness to the best of the agents of trial 2, when re-evaluated in a random new world. The conclusion is that the fitness metric used up to this point has not selected for general-purpose grazers, and when tested in a new world, the general grazing ability does not vary much among individuals.

9.6.6 The third grazer trial

This trial evolves its population in several different grid worlds, rather than in just one.

The normal metric fitness was changed to use 10 randomly created gridworlds, with fitness tests rotated between grids. Each test ran the agent 20 times, twice each in the different grids, starting each time at a random location. Rigorous fitness testeds each agent ten times in each grid, starting in new random locations each time (total tests, 100). The maximum length of each agent was shrunk to 200, to make the resulting code easier to analyze. No other changes were made.

After running 27 trials, the average fitness was 15.5, and the average rigorous fitness was 19.6. While the rigorous fitness was much worse than that evolved during the pervious run, the question of interest now is how well this number represents the general fitness of the grazers beyond the worlds they evolve in. The results showed a better correlation, but still some difference between worlds in which the agents evolved and entirely new worlds

Fitness for the five most fit (sorted by rigorous fitness evaluated on the same set of worlds that code was evolved under):

Size	Sensing opcodes	Useless opcodes	Normal fitness	**Rigorous fitness**	Re-evaled rigorous fit
83	7	30	14.3	**17.4**	26
101	5	39	13.1	**17.4**	24
80	1	31	14.6	**17.5**	24
101	4	38	14.95	**18.1**	23
79	5	27	13.95	**18.3**	23

Fitness for the least most fit:

Size	Sensing opcodes	Useless opcodes	Normal fitness	**Rigorous fitness**	Re-evaled rigorous fit
92	6	37	17.35	**21.81**	24
77	4	30	17.95	**21.81**	30
99	3	40	16.15	**22.35**	28
85	7	28	17.65	**22.38**	29
116	8	49	17.1	**22.39**	30

The notable trend is that the rigorous fitness becomes significantly worse when re-evaluated with a newly generated set of 10 worlds. Yet again, it appears that the fitness metric did not produce generalized grazers. The average revaluated fitness in these tests, however, was better than that from individuals evolved in earlier runs without rotating grid worlds, though not by much.

It still seems, then, that even with changing grid worlds during the run, that the population is evolving to fit the landscapes. When the landscapes are generated again at random, performance drops.

Nonetheless, it seems odd that 10 randomly generated grid worlds would all share enough characteristics that a grazer could over-adapt to their collective layout. It seems that the only way to test this theory would be to try a run with even more grid worlds, and even more fitness tests. Given enough tests in different worlds, it should be impossible for a program to adapt the specific layout of each. This question will be left for future research to address.

9.7 Conclusion

In this narrative account, LJGP has been applied to several different variations of two related behavioral simulations. The trials conducted were all preliminary explorations meant to explore what kinds of questions might be addressable with grid world simulations of foraging and mowing. It was hypothesized that navigation of a grid world would present issues from the real world, including efficient discovery of geographical layout, and strategies for reducing wasteful backtracking.

Where it was hypothesized that evolution would find behavioral programs that reacted to their environment, it was found that it was usually much more effective to ignore sensors. Where it was hypothesized that the layouts of the worlds simulated were varied enough that any behavior evolved would be generally applicable to other layouts, instead evolution found methods to fit the environments perfectly, at the expense of any sort of general ability for other layouts.

While the programs evolved did not perform computationally complex tasks, the behaviors they produced were fit to the environments they evolved in, emphasizing that genetic programming is an effective technique for analyzing the complexity of problems. It especially underlines the fact that toy problems are often susceptible to simple solutions that do not involve anything like the amount of complexity a researcher expects.

While these are interesting questions and issues for further research, these runs were also conducted to test the LJGP system. The package has now successfully completed over 1000 runs and is thoroughly stable in many different Java environments. Furthermore, by running extensive trials with LJGP, the need for features like network backups, rigorous fitness, and conversions of vCPU programs to and from source code was discovered. With these features implemented, LJGP will be more useful for future research projects.

Conclusion

Conclusion

Manually writing software is a difficult task. It requires careful attention to detail and a full understanding of each problem to be solved. This project has focused on genetic programming, a field created to make computers write software automatically.

This thesis was designed to provide a wide exposure to the field of genetic programming, with two related projects.

The LJGP project engaged implementation issues and theoretical considerations for contemporary genetic programming by building a fully functionally genetic programming system from scratch.

The PushGP project focused on extending the boundaries of current research on evolving modular programs. Evolving modularity is a particularly important topic in genetic programming because most interesting, hard problems have regularities that make them easier to solve by splitting them into smaller, simpler problems, which may occur multiple times, but only have to be solved once.

The LJGP Project

The goal of the LJGP project was to implement a portable, Java-based genetic programming environment that evolves linear programs. The system built satisfied these goals and provides the following specific features:

- Portability between multiple platforms and Java environments.
- Easy execution on host machines without requiring any installation.
- Easy distribution of runs to, and collection of results from, geographically distant hosts.
- High stability (tested via successful completion of 1000 runs and many thousands of computing hours).
- Demonstrated ability to evolve maximally short programs that effectively make use of registers to store and reuse partial calculations.

For future work, it would be worthwhile to add the following features to LJGP:

- More example problems encoded in LJGP to allow quick testing of new genetic programming ideas and methods in multiple domains.
- A larger set of virtual CPU instructions to allow more flexibility in the types of programs evolution can create.

- A mechanism for allowing LJGP to evolve modular solutions, via either a system based on automatically defined functions (ADFs) or by adding a set of functions that allow arbitrary conditional jumps within the code.

PushGP

The work done with PushGP focused on techniques that encourage genetic programming to evolve more modular, complex, and functionally expressive code. PushGP allows modularity by virtue of using the Push language, which supports modularity by providing a code stack where program code can be interactively created and executed.

The research in chapters 4, 5 and 6 demonstrated that PushGP, like GP with automatically defined functions, can evolve modular programs that decompose problems into easier to solve sub-problems. Unlike with ADFs, however, this was done without requiring the GP system to externally impose structures that create modularity. Furthermore, the results demonstrated that PushGP's method of evolving modularity allows the programmer to use it to create unconstrained, novel solutions.

PushGP's modularity was explored using problems taken from Koza's Genetic Programming II. These problems have been shown to have modular solutions when Koza used ADFs. The following four results were found when testing PushGP's ability to also evolve modularity for these problems:

- In most cases where modularity was allowed via code manipulation, PushGP found a modular way to exploit the decomposition of the problem. It did this without any explicit selective pressure towards modularity. It appears, then, that modularity evolved because it suited the problem search space.

- In most cases, PushGP evolved modularity via repeatedly executing code that had been written by evolution. PushGP, therefore, allows functions to be created directly by evolution, much like ADFs do.

- In one case, PushGP evolved modularity that modified the code that it executed on the fly to create a recursive function. This demonstrates that PushGP allows recursive functions (unlike typical ADFs) and can take advantage of the ability to dynamically modify the code it executes.

- For symbolic regression, PushGP demonstrated that the stack allowed reuse of partial calculations (another form of modularity) in a way that tree-based genetic programming could not.

The point of allowing evolution to create modular programs is it can make solving problems quicker and easier. It is important, therefore, to ask how well PushGP performed compared to more traditional genetic programming systems with ADFs. It was found that:

- PushGP was able to find solutions for all of the problems taken from Genetic Programming II, either by using default settings for all parameters, or after tuning some parameters (maximum program size, instruction set, etc.) to the problem.

- PushGP performance varied in respect to ADFs. Disappointingly, however, PushGP did not show any absolute improvements in performance.

- Results with even-six-parity and even-four parity suggests that the modularity that PushGP evolves allows it to scale well to harder problems in the same way that ADFs do. On parity at least, it appears possible that PushGP scales even better than ADFs do, and that PushGP will have a performance advantage on sufficiently hard parity problems.

Research was also conducted on the effectiveness of the genetic operators that PushGP uses. It was found that

- PushGP's default crossover and mutation operators both tended to create larger programs over time. When the individuals in the population neared the size limit, many of the new children produced had to be discarded because they were too big. This would start to happen fairly early in runs, and thereafter, evolution would progress with abnormally high levels of duplication.

- The inner nodes of programs did not see as much modification from genetic operators as did the leaf nodes. This happened because there was an equal chance of selecting any node in a given program, and there were many more leafs than inner nodes on average.

- Several new crossover and mutation operators that addressed these issues did not perform any better on a test symbolic regression problem. In most cases, in fact, they did worse.

- The only configuration that did improve performance on the symbolic regression problem used PushGP's default mutation operator and no crossover. Though not ideal, it appears that the default operators are sufficient for most tasks.

PushGP was also applied to two problems not yet solved by other genetic programming systems. These problems were particularly interesting because there was no clear way to solve them using Koza-style GP with ADFs. It was clear, however, that successful programs could be written in the Push language. The question was if PushGP could evolve those programs. Initial research found:

- For the even-n-parity problem, PushGP was able to produce a zero error individual for the training set of 3, 4 and 6 even-parity fitness cases. The program found failed to generalize, however, to the even parity problem for an unbounded number of inputs. This result appears to stem from the fitness metric and how the problem was specified, and not any inherent limit in PushGP. With some tweaking, this problem should be solvable.

- For the factorial problem, PushGP was unable to match the training set. This appears to be a result of factorial being a GP-hard problem, in that initially good solutions found are not near in the search space to the true solution. This makes the problem particularly hard to solve for any genetic programming system. Nonetheless, several ideas were proposed that might allow PushGP to find a solution.

The experiments with PushGP documented in this thesis are some of the first conducted with it. There are many details still unexplored. Planned future areas of investigation include:

- More experiments with even-n-parity and factorial problems using new fitness functions and PushGP parameters.

- More experiments with symbolic regression to address why, when the default PushGP configuration is used, it is difficult to solve the sextic polynomial problem from Genetic Programming II.

- Determining what combinations of PushGP instructions are sufficient to allow evolution of modular solutions.

- Determining whether PushGP continues to scale well compared to ADFs when applied to larger even-parity problems.

- The application of PushGP to other interesting problem domains, where modularity and multiple data types are required to decompose problems into forms that genetic programming can solve.

Appendix A. Computational effort – lisp code

The following common Lisp code calculates computational effort as defined by Koza in GP I and GP II (Koza, 1992; Koza 1994).

```lisp
(defun number<= (lst value)
  "return the number of elements of a list which are less than value"
  (let ((r 0))
      (dotimes (i (length lst))
        (if (<= (nth i lst) value)
          (setq r (1+ r))))
r))

(defun round-up (n)
  "Round up to the nearest whole integer"
  (multiple-value-bind (whole part) (truncate n)
    (if (> part 0)
      (1+ whole)
      whole)))

;;-------------------------------------------------------
;; Notes
;; A success-list contains an unordered list of generations upon
;; which the target solution was found.

(defun calc-probability (success-list run-size generation)
  "P(M,i)"
  (/ (number<= success-list generation) run-size))

(defun print-probability-range (success-list run-size generation)
  "Pretty print P(M,i)"
  (dotimes (i generation)
    (format t "~%~3,'0d: ~4,2F" i (calc-probability success-list run-
size i))))

(defun probability-range (success-list run-size generation)
  "collect P(M,i)"
  (let ((out nil))
  (dotimes (i generation)
    (push (calc-probability success-list run-size i) out))
  out))


(defun runs-to-success (success-list run-size generation probability)
  "R(M,i,z)"
  (round-up (/ (log (- 1 probability) 10)
               (log (- 1 (calc-probability success-list run-size
generation)) 10))))

(defun effort (success-list run-size generation probability population-
size)
  (if (= 0 (calc-probability success-list run-size generation))
    -1
    (* population-size
       (1+ generation)
     (runs-to-success success-list run-size generation probability))))
```

```
(defun print-effort-range (success-list run-size generation probability
population-size)
  "print I(M,i,z)"
  (dotimes (i generation)
    (format t "~%~3,'0d: ~A" i  (round-up (effort success-list run-size
i probability population-size))))
  (format t "~%Lowest Computational Effort: ~A" (first (sort
                                (remove -1 (effort-range success-list
run-size generation probability population-size))
                                #'<))))

(defun effort-range (success-list run-size generation probability
population-size)
  (let ((out nil))
  (dotimes (i generation)
    (push (round-up (effort success-list run-size i probability
population-size)) out))
  out))

(defun interactive ()
  (let ((success-list nil )
        (run-size 0)
        (generation 0)
        (probability 0)
        (population-size 0))

    (format t "~%Enter number of runs: ")
    (setq run-size (read))

    (format t "~%Enter max number of generations: ")
    (setq generation (read))

    (format t "~%Enter required probability of success (ie .99): ")
    (setq probability (read))

    (format t "~%Enter population size: ")
    (setq population-size (read))

    (format t "~%Enter generations at which success was declared, -1 to
end")
      (loop
        (format t "~%-1 to end: ")
        (let ((input (read)))
          (if (= input -1)
            (return nil))
          (push input success-list)))

      (format t "~% Probability of success at each generation:")
      (print-probability-range success-list run-size generation)

      (format t "~% Probability of success at each generation:")
      (print-effort-range success-list run-size generation probability
population-size)
))
“evaluate (interactive) to start the end-user interface for calculating
computational effort”
```

Appendix B. Genetic programming systems in Java

LJGP is not the first genetic programming system in JAVA, however, it offers unique features that the others do not. This section will describe the other GP systems.

ECJ

Sean Luke's Tree Based GP System. Actively developed and quite large (1.3 megs of code). It comes with five example GP problems, including implementations of Koza's ant, lawnmower, and regression systems. http://www.cs.umd.edu/projects/plus/ec/ecj/

GPsys

Adil Qureshi's Tree based GP system. Actively developed, and fairly complete. Uses JAVA 2 (collection classes mostly) so it is not portable across older JREs.
http://www.cs.ucl.ac.uk/staff/A.Qureshi/gpsys_doc.html

JGProg

Open Source, Tree based GP system. Actively developed, and fairly complete. Provides a system for run distribution across machines. Architecture not as flexible as ECJ, and GPsys, but is easier to learn because of its lack of flexibility.
http://jgprog.sourceforge.net/

DGP

Tree based GP system by Fuey Sian Chong (Chong & Langdon, 1999), built to experiment with distribute runs between machines with Java using a central server. Encodes the artificial ant problem.
ftp://ftp.cs.bham.ac.uk/pub/authors/W.B.Langdon/gp-code/DGP/

Appendix C. LJGP/JAVA-VM benchmarks

No attempt is made here to compare performance of LJGP to other GP algorithms, since the No Free Lunch theory calls into question all but the most considered attempts at such comparisons. Instead, the question addressed here is Java VM performance: which Java systems perform best in terms of walk clock time when running LJGP?

This benchmark performs symbolic regression to find the function $\mathbf{x + x^2 + x^3 + x^4}$ given 10 data points (x = 0...9), with an operator pool of addition, subtraction, multiplication, division, and assignment, on a 4 register virtual CPU. The population size is 1000, and the genetic modifiers include single point crossover, single point insertion and deletion, and single point mutation. Normally this system can find the function named above in less than a second, but careful selection of the right random seeds produces a very poor initial population, requiring many generations to complete. The cycles expended, therefore, are representative of completing a single GP run where the fitness evaluation is relatively easy, but many generations must be completed.

The integer trials use a vCPU with integer registers, the Float trial uses floating point registers.

Understanding the numbers

Values reported are normalized arbitrarily to the speed of the slowest machine tested, running the fasted VM available for that platform. That machine scores an value of 1 on both the integer test, and float test, which correspond to 134 seconds to complete the first test, and 242 to complete the float test. A value of 2.0 means that the computer performed twice as fast as the base system.

Hardware	JRE	Integer Score	Float Score
Dual 433Mhz Celeron, 256MB, Win2k	Symantec Java 4.00.010(x) (from Visual Cafe 4.0a)	4.3	4.5
Dual 433Mhz Celeron, 256MB, Win2k	Sun JDK 1.3 Applet Viewer	2.3	2.4
Dual 433Mhz Celeron, 256MB, Win2k	Internet Explorer 5.5	1.9	2.8
Dual 433Mhz Celeron, 256MB, Win2k	Internet Explorer 5.0	1.9	2.8
Dual 433Mhz Celeron, 256MB, Win2k	Netscape 4.75 (Java 1.1.5)	1.4	0.5
Athlon 550Mhz 128MB, Win98	Internet Explorer 5.0	2.4	3.6
PII 400Mhz, 128MB, Win95	Internet Explorer 5.5	1.8	2.5
G3 iMac 400Mhz, 128MB, OS 9	Internet Explorer 5.0	1.6	1.6
G3 iMac 333Mhz, 192MB, OS 9	Internet Explorer 5.0	1.2	1.2
G3 iBook 300Mhz, 96MB, OS 9	Internet Explorer 5.0	1.3	1.2
G3 iMac 266Mhz, 96MB, OS 9	Internet Explorer 5.0	1.0	1.0

Notes. Other than for the Dual Celeron, all values are ordered from fastest to slowest. The values for the Celeron are kept together to aid in comparing different JREs on the same machine. During runs on the Dual Celeron, the second CPU sees only a few percentage points of use, so the values should compare closely to a single Celeron 433 machine.

Bibliography

Anderson, B, P. Svensson, M. Nordahl, and P. Nordin. 2000. *On-line Evolution of Control for a Four-Legged Robot Using Genetic Programming* In: *Real World Applications of Evolutionary Computing*(S. Cagnoni, et al., Eds), pp 319-326. Springer, Berlin, Germany.

Angeline, P. 1997. *Subtree crossover: Building block engine or macromutation?* In: *Genetic Programming 1997: Proceeding of the Second Annual Conference, July 13-16, 1997* (J. Koza, K. Deb, M. Dorigo, D. Fogel, M. Garzon, H. Iba, and R, Riolo, Eds.). pp 9-17. Morgan Kaufmann, San Francisco, CA.

Banzhaf, W., P. Nordin, R. Keller, and F. Francone. 1998. *Genetic Programming: An Introduction.* Morgan Kaufmann, San Francisco, CA.

Bruce, W. 1997. *The Lawnmower Problem Revisited: Stack-Based Genetic Programming and Automatically Defined Functions.* In: *Genetic Programming 1997: Proceeding of the Second Annual Conference, July 13-16, 1997* (J. Koza, K. Deb, M. Dorigo, D. Fogel, M. Garzon, H. Iba, and R, Riolo, Eds.). pp 52-57. Morgan Kaufmann, San Francisco, CA.

Chong, F. and W. Langdon 1999. *Java based Distributed Genetic Programming on the Internet.* In: *Proceedings of the Genetic and Evolutionary Computation Conference* (Wolfgang, B., J. Daida, A. Eiben, M.. Garzon, V. Honavar, M. Jakiela, and R. Smith, Eds.) p 1229. Morgan Kaufmann, San Francisco, CA.

Fogarty, T. (1989). *Varying the probability of mutation in the genetic algorithm*. In: *Proceedings of the Third International Conference on Genetic Algorithms* (J. Schaffer, Ed.), pp 104-109. Morgan Kaufmann, San Francisco, CA.

Karlsson, R., P. Nodin, and M. Nordahl. 2000. *Sound Localization for a Humanoid Robot by Means of Genetic Programming.* In: *Real World Applications of Evolutionary Computing*(S. Cagnoni, et al., Eds), pp 65-76. Springer, Berlin, Germany.

Koza, J. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press.

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

Koza, J., F. Bennett, A. Andre, and M. Keane. 1999. *Genetic programming III : Darwinian Invention and Problem Solving.* Morgan Kaufmann, San Francisco, CA.

Koza, J. 2000. *Human-Competitive Machine Intelligence.* Available at http://www.genetic-programming.com/humancompetitive.html

Lang, K. 1995 *Hill climbing beats genetic search on a Boolean circuit synthesis of Koza's.* In: *Proceedings of the Twelfth International Conference on Machine Learning*, Tahoe City, CA. Morgan Kaufmann, San Francisco, CA.

Langdon, W., T. Soule, R. Poli, and J. Foster. 1999. *The Evolution of Size and Shape*. In: *Advances in Genetic Programming volume 3*. (L. Spector, W. Langdon, U. O'Reilly, and P. Angeline, eds.) Pp 163-190. MIT Press, Cambridge, MA.

Luke, S., C. Hohn, J. Farris, G. Jackson, and J. Hendler. 1998. *Co-evolving Soccer Softbot Team Coordination with Genetic Programming*. In: *RoboCup-97: Robot Soccer World Cup I (Lecture Notes in Artificial Intelligence No. 1395) (*H. Kitano, ed).Pp 398-411. Springer-Verlag. Berlin, Germany.

Nordin, P. 1994. *A compiling genetic programming system that directly manipulates the machine code*. In: *Advances in Genetic Programming* (K. Kinnear, ed). Pp 311-331. MIT Press, Cambridge, MA.

Nordin, P. and J. Nordin. 1997. *Evolutionary Program Induction of Binary Machine Code and its Application*. Krehl-Verlag, Munster, Germany.

Spector, L. 2001. *Autoconstructive Evolution: Push, PushGP, and Pushpop*. In: *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001* (L. Spector, E. Goodman, A. Wu, W. Langdon, H.. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. Garzon, and E. Burke, Eds). Morgan Kaufmann, San Francisco, CA.

Tomassini, M. 1999. *Parallel and Distributed Evolutionary Algorithms: A Review*. In: *Evolutionary Algorithms in Engineering and Computer Science* (K. Miettinen, M. Makela, P. Neittaanmaki, and J Periaux, Eds). Pp 113-131. John Wiley & Sons, LTD, New York.

Wolpert, D, and W. Macready, 1997. *No Free Lunch Theorems for Optimization*. IEEE Transactions on Evolutionary Optimizations, vol. 1, no. 1, pp 67-82.

Zhao, K. and J. Wang. 2000. *Multi-robot Cooperation and Competition with Genetic Programming* In: *EuroGP 2000 Proceedings.* Pp. 50- 359. Springer-Verlag. Berlin, Germany.