# PROBABILISTIC GENOMES FOR GENETIC PROGRAMMING

Submitted to the Department of Computer Science of Amherst College

in partial fulfillment of the requirements

for the degree of Bachelor of Arts with honors

April 12, 2023

Author: David Dang

Advisor: Prof. Lee Spector

# Abstract

Genetic programming (GP) is a problem-solving method inspired by biological evolution that generates computer programs to solve problems using variation and selection. In most GP systems, the smallest amount of variation made to a program is the addition or removal of an instruction, which often causes a large impact on how the program behaves. In this thesis, we explore the idea of allowing even smaller changes to programs by associating these instructions with probabilities. As a result, rather than an instruction being added or deleted, its probability of being included in the program can be adjusted by a small amount. We implement this idea in the PushGP genetic programming system by developing a probabilistic genome representation. We show that, in some cases, the use of probabilistic genomes improves the solving power of a GP system.

# Acknowledgements

I would like to thank Professor Spector for his help on this thesis and for originating the idea of Probabilistic Plushy genomes. I was able to learn more about genetic programming and ultimately come up with the biased-perturbation mutation method through our engaging discussions. I enjoyed working together to develop the other parts of the thesis.

I greatly appreciate the help of Ryan Boldi for recommending me the benchmark problems I used in my experiments and the rest of the Push lab for suggesting new ideas for me to explore in my study of Probabilistic Plushies.

Finally, I want to thank my family and Amherst classmates for their support throughout this endeavor. They motivated me to continue to push through and complete this thesis. I am very fortunate to have these people in my life.

# Contents

# Chapter 1

# Introduction

## 1.1 Overview

Genetic programming (GP) is a population-based method inspired by biological evolution that generates computer programs to solve problems using evolutionary concepts such as variation and selection. For a GP system to run, a human user must input some problem, often specified in terms of a goal and a set of constraints, for the system to solve. Once the system receives a problem as input, it will begin the evolutionary process by initializing a random population of computer programs. After that, each computer program in the population is evaluated based on its ability to solve the specified problem according to the provided goal and set of constraints. If the GP system finds a program in the population that solves the problem from its evaluation process, then it will output that program for the human user. Otherwise, a select few of the computer programs that have the potential to solve the problem are chosen from the population to reproduce and create new computer programs. The system will group these child programs to form a new population and have them repeat the evolutionary process. This cycle continues until a GP system either finds a solution program or halts after meeting an iteration limit.

A common issue in genetic programming is the process of creating new programs from selected individuals in the population. More specifically, the variation methods a GP system uses on a computer program often create child computer programs that behave much differently from their parent program. In most GP systems, the smallest amount of variation to a program is an addition or removal of a gene instruction. Even this degree of change can greatly impact a program's behavior. As a result, if a computer program has certain semantics that helps it solve a given problem, there is a chance that new programs created from it will not possess this behavior. Because of this loss of behavioral continuity, GP systems can suffer in performance when solving problems, especially ones that require many iterations to evolve programs.

In this thesis, we explore the idea of allowing even smaller changes to programs by associating instructions with probabilities. Thus, rather than a computer program's instruction being added or deleted, the instruction's probability of being included in the program can be adjusted by a small amount. With this new variation method, a computer program can pass down aspects of its behavior to its child program, allowing the two programs to share similar semantics. We reason this effect of variation allows GP systems to maintain valuable behavioral traits within the population, improving their ability to search for programs that are capable of solving the given problem.

We implemented this idea in a PushGP genetic programming system by developing a probabilistic genome representation for computer programs that associated each instruction with a probability. These probabilistic genomes can use these probabilities to express their instructions and generate non-probabilistic genomes. Since we introduced a new genome into the system, we also had to create an error evaluation process for these genomes to help us better gauge their ability to solve the specified problem. We designed an evaluation process where a probabilistic genome expressed multiple non-probabilistic genomes. The method would then select the best non-probabilistic genome from the expressed genomes to represent the probabilistic genome's problem-solving capabilities. Additionally, we added other variation methods and constructed them to be compatible with probabilistic genomes. Finally, we included our new variation method that targets the probabilities of probabilistic genomes to maintain behavioral similarity between parent and child genomes.

We ran an initial experiment to analyze the GP system's performance using these settings. More specifically, we began with a preliminary experiment that measured the success rates of the GP system on float regression benchmark problems when given non-probabilistic run parameters versus probabilistic run parameters. Each of the run parameters used a different type of genome, error evaluation process, and set of variation methods. Our first set of runs helped to determine if the probabilistic settings could help the GP system perform better than itself when using the non-probabilities settings. After our initial testing, we repeated the non-probabilistic runs with the additional requirement of increasing the parameters in the non-probabilistic run parameters to equalize the total number of error evaluations to the same amount in the probabilistic run parameters.

We found from our first set of runs that the GP system had a higher rate of success when using the probabilistic run parameters than the non-probabilistic run parameters across the entire benchmark suite. According to our comparison of the success rates between the probabilistic and non-probabilistic runs, the probabilistic runs performed about the same or better than the non-probabilistic runs. After standardizing the total number of evaluations for the non-probabilistic settings, we observed that the non-probabilistic runs had better success rates than the probabilistic runs on many of the float regression problems. When totaling the number of successes across the benchmark suite for the probabilistic and non-probabilistic runs, however, we found that the success rate of the probabilistic runs was greater than the success rate of the non-probabilistic runs over all the benchmark problems. These preliminary results show that while probabilistic genomes can improve a GP system's success rate on a given problem, it is still unclear if we can generalize this increase in performance when we provide non-probabilistic and probabilistic runs with the same total number of error evaluations.

We then moved to study how the GP system with probabilistic settings would perform given different genetic sources. A genetic source is the set of instructions specified to the GP system that it uses to generate computer programs to solve a given problem. A GP system's performance is known to suffer when provided with a more general genetic source consisting of instructions for different data types. Given such genetic sources, we reason that a GP system with probabilistic settings could have a better performance on a benchmark suite than when using non-probabilistic settings. As such, we created an "everything but the kitchen-sink", or kitchen-sink, instruction set and a hand-tuned instruction set for our float regression benchmark problems. These genetic sources were inspired by Helmuth, Pantridge, Woolson, and Spector's paper on "*Genetic Source Sensitivity and Transfer Learning in Genetic Programming*". For each genetic source, we ran an experiment that compared the success rates of the GP system on benchmark problems using different non-probabilistic and probabilistic run parameters. We hypothesized that the probabilistic runs would have a higher success rate than the non-probabilistic runs on the kitchen-sink instruction set.

We found that the non-probabilistic runs had better success rates than the probabilistic runs on the hand-tuned genetic source. Our GP system had its best performance on the benchmark suite when we specified the non-probabilistic run parameters with additional error evaluations. Regarding our runs using the kitchen-sink instruction set, both non-probabilistic and probabilistic runs suffered in performance, with the non-probabilistic having the bigger dropoff in success rate. We found that certain probabilistic run parameters allowed our GP system to outperform itself using non-probabilistic settings on some of our benchmark problems.

In the remainder of this thesis, we will begin with an in-depth introduction to genetic programming. We will then further discuss the motivation of the thesis and provide a general description of our contributions to fulfill our research objectives. After that, we will transition to a brief literature review of previous work with probabilistic genomes. The next chapter then describes our software implementation and a detailed explanation of our probabilistic configurations to a PushGP genetic programming system. We will then explain the experiments we performed and our findings from the results we obtained from our experimentation. We conclude with a chapter on possible future work and a summary of the overall thesis.

## 1.2 What is Genetic Programming?

### 1.2.1 Automatic Programming

Automatic programming is a type of computer programming in which some system or software generates programs based on certain specifications defined by human programmers. To concisely put it, automatic programming produces code designed to write code. Automatic programming is a branch of artificial intelligence that provides many useful applications, including generative programming and source code generation.

An important utilization of automatic programming is its effectiveness in solving problems that do not have a known solution. Hand-written code is often a flawed approach to figuring out such problems due to the error-prone nature of programming and the lack of information human programmers require to develop a solution. With automatic programming, however, human programmers can specify goals and constraints at a higher level of abstraction that allows them to generate programs possessing their desired features. Automatic programming helps human programmers ignore implementation details by allowing human users to describe general guidelines of a problem so that the computer can find possible solutions for them.

One of the main approaches to automatic programming is genetic programming (GP), an evolutionary algorithm that generates computer programs capable of creating other programs using biological principles. Genetic programming draws inspiration from Darwinian evolution, where organisms survive, adapt, and evolve through natural selection. Genetic programming incorporates these principles into its algorithms to find solutions to problems that human programmers struggle to solve. The flexible and adaptive program structure of genetic programming allows such algorithms to solve problems from different domains. Genetic programming has been a successful automatic programming tool, contributing to many practical areas of computer science, such as code synthesis and automatic bug-fixing.

### 1.2.2 How Genetic Programming Works

Genetic programming algorithms are a type of evolutionary algorithm, meaning they utilize evolution as the main factor when deciding how to traverse the solution space. Unlike the solution space of other evolutionary algorithms, genetic programming algorithms aim to search for computer programs that can solve a given problem. A genetic programming algorithm begins by initializing a random population of computer programs. After that, the algorithm continuously improves the population by selecting the set of programs with the lowest errors in the population and reproducing child programs using variation methods. A genetic programming run is a success if it finds a computer program within the population that solves the given problem.

A genetic programming system starts by creating a defined number of randomly-generated candidate solutions, also known as “individuals”, that form a population. Each candidate solution is a computer program derived from a specified genetic source. A genetic source is a list of functions and constants a human programmer inputs into a GP system to allow it to generate computer programs that consist of these instructions. Often, human programmers will include instructions in the genetic source that are relevant to the problem they want to solve.

Once a GP system constructs an initial population, it will then evaluate each individual in the population to measure their errors. For the error evaluation phase to work, the human programmer must specify an objective function, training set, and testing set to determine the errors of each candidate solution. An individual's errors help to gauge how useful an individual will be in solving the given problem. A computer program with a low total error means that it is a good solution for the problem and is more likely to produce child programs that can lead to better solutions. Similar to the natural selection processes in biological evolution, the genetic programming system will select the individuals in the population with lower errors to reproduce in the variation phase.

Instead of total error, many resources on genetic programming often use "fitness" to denote a computer program's ability to solve a defined problem. Both of these values serve to help GP systems compare computer programs within the population and select the better ones to create child programs throughout the genetic programming run. In context to a specified problem, however, total error and fitness are opposites of each other in that one goes up while the other goes down. We will focus more on a computer program's errors as it is more relevant to our methods and experiments.

Once the candidate solutions in the population with lower errors are selected to reproduce, the variation phase begins, where the system generates new programs using crossover and mutation methods. These methods, commonly known as genetic operators, are analogous to the crossover and mutation mechanisms performed in biological evolution. In crossover, the GP algorithm will choose two individuals as the parents and combine some amount of each individual's genetic information to form a new candidate solution that represents the child of the parents. With mutation, an individual is chosen as the parent and has one or more of the individual's gene instructions perturbed to produce a child program. Although crossover is an important genetic operator to solve GP problems, our investigation will focus primarily on the mutation side of variation.

During the reproductive process, the genetic programming system performs a combination of these genetic operators on the chosen candidate solutions of the current population to create new individuals to form the new population. The idea behind this process is that individuals with low errors contain desirable genetic material. As a result, selecting these individuals to produce new programs allows them to pass these traits on to their child programs. Additionally, variations methods can introduce new genes to the child programs to create a more diverse set of candidate solutions. By forming a new population built from the errors of the previous generation and adding new instructions into the gene pool, the system can create individuals with lower errors than their parent programs.

After the genetic programming system finishes the reproductive process, the evolution cycle repeats, starting with the error evaluation of each candidate solution in the new population. GP systems continue this sequence of error evaluation, selection, and variation until there either exists a computer program with a total error that satisfies the human programmer's specifications or when the system reaches an iteration limit.

As such, the error evaluation stage helps a GP system determine a candidate solution's ability to solve the given problem and select the candidate solutions that contain valuable genetic material. The variation phase allows the system to explore the solution space by passing this genetic information onto child programs and diversifying these programs to form a new population. By repeatedly executing these stages, genetic programming systems can optimize the population to generate a computer program that solves the specified problem. This process of evolution is analogous to how heuristics are used in search algorithms, as the objective function represents the set of constraints to a problem, and the genetic programming systems help candidate solutions approach a global optimum that satisfies these constraints.

### 1.2.3 Genetic Representation of Computer Programs

Many genetic programming systems utilize an encoding process to represent computer programs to help with the evolutionary process. The goal behind this approach is to allow GP systems to maintain a computer program's actual behavior while also being able to manipulate these programs and generate new ones. As a result, many GP systems will often focus on evolving the genetic representation of a computer program rather than the program itself. Since a program's genetic representation is not executable, these systems use the encoding process to map them back to a program to measure its errors on a given problem. One example of an encoding scheme is in Tree-Based GP, where syntax trees, binary trees with their internal nodes as functions and their leaf nodes as variables and constants, represent computer programs.

### 1.2.4 Setup and Problem Specification

Formally, genetic programming is a systematic, domain-independent method that requires a high-level statement to initiate the process of generating candidate solutions (Langdon, Poli, McPhee, and Koza 2008). Before running a genetic programming system, a human user needs to define the following specifications for a given problem:

1. genetic source;
2. objective function and training set;
3. set of run parameters;
4. termination criterion.

#### Genetic Source

The genetic source or instruction set is the set of instructions that can appear in a computer program, which can consist of:

1. external inputs that a computer program uses as arguments;
2. zero-argument functions (e.g. a function that returns a random number);
3. multi-argument functions (e.g. a function that returns the sum of two numbers)
4. constants (e.g. numbers, strings, or boolean values) that are either pre-specified or randomly generated when a new computer program is constructed.

GP problems are often domain-specific and constrained, and as a result, human programmers must provide a genetic source to help GP systems search within the appropriate solution space. This instruction set represents the possible blocks of code that can appear in a computer program. Genetic programming systems use the instructions from the genetic source to create candidate solutions throughout the evolutionary process.

#### Objective Function And Training Set

The objective function and training set help assign a candidate solution with a set of errors when evaluated on the inputs of the training set. Often, a GP system determines a candidate solution's errors by assessing the individual's output and the objective function's output at each input in the training set. This evaluation process allows genetic programming systems to compare two individuals using their respective total errors. While the last two specifications define the search space, the objective function and training set allow a genetic programming system to determine which regions of the search space it should focus on investigating. As a result, these components implicitly define the GP system's desired goal throughout the search process and the constraints a candidate solution must satisfy. Human users may also specify a testing set to provide additional test cases when measuring the errors of a solution program.

**Set of Run Parameters**

The set of run parameters specifies additional aspects of the genetic programming run, including the population size, the probabilities of executing each genetic operator, and the maximum size of a computer program. A genetic programming system's ability to solve a given problem can vary depending on the provided set of run parameters. As a result, one must carefully consider the type of parameters and the amount used for each parameter to maximize the GP system's performance.

**Termination Criterion**

The termination criterion designates how the GP system will finish its run and output its results. Often, a genetic programming system will terminate if it has found a candidate solution with a total error that satisfies a defined error threshold. The system can also halt the genetic programming run if it does not find a solution program in the population after a specified maximum number of iterations or generations. Along with returning the computer program that solves the given problem, the GP system may also output additional information, such as the number of generations to find the solution or an individual's total error on the testing set.

## 1.3 Probabilistic Genomes for Genetic Programming

### 1.3.1 Motivation

During the variation phase, it is often the case that the child programs behave much differently than their parent programs. The smallest amount of variation to a program is an addition or deletion of a gene instruction, and even this change to a candidate solution can largely impact the behavior of a program. Because of this effect on a program's semantics, a parent program is frequently unable to pass down its characteristics to a child program. As a result, if a candidate solution's behavior appears to be useful in solving a given problem, there is a chance that new individuals created from it will not inherit these behavioral traits, causing a loss of problem-solving power to the next generation of programs.

In this thesis, we explore the idea of creating a smaller effect of change by associating instructions with probabilities. More specifically, we devise a new mutation method that targets an instruction's probability rather than the instruction itself. This variation method generates a child program that contains the same sequence of instructions as their parent with slight differences in their instructions' probabilities. As such, this genetic operator allows a parent program and its child program to share similar semantics and thus helps to move desirable features into the new population.

We develop a probabilistic genome representation of computer programs to implement such a variation method. This genome would assign each gene instruction with a number representing the probability of expressing the corresponding instruction into a non-probabilistic genome. As a result, this new genome representation can map to multiple computer programs and thus requires a new approach to determining its errors. We also implement our desired variation method that targets a genome's probabilities to generate child programs that behave similarly to their parent programs. Our investigation seeks to learn more about how a GP system will perform with these new configurations.

### 1.3.2 Probabilistic Genomes

Our new encoding scheme assigns each gene instruction to a number within the interval [0,1]. We can use these values to have a probabilistic genome generate a non-probabilistic genome. When aiming to express a non-probabilistic candidate solution, each number in the probabilistic genome represents the probability of adding the associated instruction into the non-probabilistic genome. Since a non-probabilistic genome has a deterministic mapping to a computer program, a probabilistic genome can express itself to many non-probabilistic genomes and, thus, map to more than one computer program. We created a probabilistic genome called the Probabilistic Plushy based on the non-probabilistic genome representation of Push programs called Plushies, which we further discuss in section 2.

### 1.3.3 Error Evaluation

Our error evaluation method relies on a probabilistic genome's ability to map to different computer programs to measure its errors. More specifically, given a probabilistic genome as input, we express the genome to a specified number of non-probabilistic genomes. After that, we perform the process used to calculate the errors for non-probabilistic genomes by mapping each one to a computer program and running them on the training set. Once we obtain the errors of each non-probabilistic genome, we sum each one to find the minimum total error within this list of numbers. We then assign the errors corresponding with the determined minimum total error to our probabilistic genome to have it represent the genome's errors.

The intuition behind this approach was to determine how to reasonably search the lowest errors from the set of non-probabilistic genomes expressed by a given probabilistic genome. By allowing a probabilistic genome to generate a few non-probabilistic genomes and translating these genomes into computer programs, we can obtain more information on which non-probabilistic genomes lead to lower errors. Additionally, since a probabilistic genome ultimately results in the creation of a non-probabilistic genome, it makes sense to associate the errors of this genome with the probabilistic genome.

Along with determining a probabilistic genome's set of errors, we associate a non-probabilistic genome with the probabilistic genome. More specifically, the non-probabilistic genome we assign is the one whose errors we designated as the errors of the probabilistic genome. We perform this additional association in the evaluation process to help our new variation method perturb the probabilities of a probabilistic genome.

### 1.3.4 Variation Methods

We created a new mutation method that targets an instruction's probability and results in a smaller effect of variation when producing a new probabilistic genome. This method, called biased-perturbation mutation (BPM), works by tweaking the probabilities of a probabilistic genome based on the product of a Gaussian noise factor and a specified standard deviation. The biased-perturbation mutation method utilizes the non-probabilistic genome associated with the probabilistic genome to decide how to change a gene instruction's probability. More specifically, the method will sweep through each gene instruction in the probabilistic genome and check if it appears in the non-probabilistic genome. If the probabilistic genome expressed the instruction, then the instruction's probability is more biased towards increasing. On the other hand, the instructions not expressed in the non-probabilistic genome are more likely to have their probabilities decrease. The resulting perturbation forms a new probabilistic genome consisting of the same instructions as their parent genome with slight differences in each instruction's probability.

We bias the amount of perturbation to an instruction's probability based on whether or not it was expressed in the associated non-probabilistic genome because we deem these instructions important when generating new probabilistic genomes with lower errors. Since the non-probabilistic genome we assign to the probabilistic genome has the lowest errors out of the set of expressed non-probabilistic genomes in the error evaluation process, we reason the instructions in the assigned genome can contribute to solving the given problem more than any other instruction in the probabilistic genome. As a result, we want to increase the likelihood of the probabilistic genome expressing these instructions while lowering the chance of expressing the instructions that do not appear in the associated non-probabilistic genome. This logic helps to generate new probabilistic genomes that are more biased in generating non-probabilistic genomes with the same instructions as the ones in the associated non-probabilistic genome.

By changing a genome's probabilities rather than its instructions, the biased-perturbation mutation method allows a child probabilistic genome to share similar semantics with its parent genome. Because of this smaller effect of variation, both genomes consist of the same sequence of instructions, allowing them to express near identical sets of non-probabilistic genomes and computer programs. In addition, the mutation method provides a probabilistic genome to pass down important behavioral traits to newly created probabilistic genomes, as the child genomes are more likely to express non-probabilistic genomes with lower total error. Over many generations, the biased-perturbation mutation method can provide probabilistic genomes with enough information for GP systems to output a program that solves the specified problem.

Besides the biased-perturbation mutation method, we also implemented other variation methods normally applied to non-probabilistic genomes into versions compatible with probabilistic genomes. The only change required for these methods was to attach a probability to any new instruction being added to a probabilistic genome so that the genome could express it into a non-probabilistic genome and allow its probability to be tweaked when using the biased-perturbation mutation method.

## 1.4 Research Objectives

The main research question under investigation is if probabilistic genomes can help a GP system perform better on problems it struggles to solve when using non-probabilistic genomes. These probabilistic genomes will allow us to create a genetic operator that manipulates a genome's actual semantics rather than its syntactic representation. By making small changes to the behavior of probabilistic genomes at each generation, we hypothesize that a GP system can interpret semantic relations that it overlooks using non-probabilistic genomes during evolution, thereby increasing the GP system's solving capabilities. Our investigation aims to compare a GP system's performance given different non-probabilistic and probabilistic settings based on objective measurements such as success rate.

Although there is a possibility that probabilistic genomes will help a GP system to solve previously difficult problems, it could be that they do not improve and even worsen the system's performance across all GP problems. As a result, another goal of this project would be to identify possible limitations in our experimentation and develop potential theoretical justifications from our empirical results. From our investigation, we would like to gain more insight into applying probabilistic genomes in a GP system and their effectiveness in solving problems.

Additionally, we will present future research questions according to our findings. Our project ties into current genetic programming research as most work focuses on developing new methods to increase the performance of existing GP systems on various problems. As such, we would like to discuss possible projects from our study of probabilistic genomes to help succeeding researchers explore more about these genotypes and genetic operators that allow parent and child genomes to maintain behavioral similarity.

## 1.5 Related Works

### 1.5.1 Probabilistic Incremental Program Evolution: Stochastic Search Through Program Space

Salustowicz and Schmidhuber's 1997 paper "*Probabilistic Incremental Program Evolution: Stochastic Search Through Program Space*" presents Probabilistic Incremental Program Evolution (PIPE), a novel technique for automatic program synthesis that combines a probability vector of program instructions to a tree-based representation of programs to help it generate computer programs. PIPE relies on an n-ary called a Probabilistic Prototype Tree (PPT) throughout the evolutionary process to help it traverse the search space and create program trees. Each node in a PPT contains a random constant and probability vector that associates a probability to each instruction in the specified genetic source.

To create solution programs, PIPE uses generation-based learning to evolve programs by improving their fitness values. PIPE begins by initializing a PPT and generating a population of programs using the probabilities from the PPT. After that, it measures the fitness of each program in the population and finds the program with the best fitness value. Next, the prototype tree's probabilities are modified such that the PPT's probability of generating the best program of the current generation increases. PIPE then performs mutation on the PPT by finding the nodes that helped to create the current best solution and mutating their probabilities by some probability. Finally, PIPE prunes the prototype at the end of each generation, and the evolutionary cycle repeats once more.

The researchers compared PIPE to Koza's GP variant on a function regression problem and the 6-bit parity problem. They found that PIPE performed better on the 6-bit parity problem than GP. Additionally, its best solutions to the function regression problem were better than GP's best solutions.

The work in this thesis focuses on evolving a population of probabilistic genomes in a genetic programming run. Additionally, we discuss a new error evaluation process for GP systems to gauge a probabilistic genome's solving power on a problem by evaluating the errors of the non-probabilistic genomes it can express from its probabilities.

### 1.5.2 Probabilistic Grammatical Evolution

In their 2021 paper "*Probabilistic Grammatical Evolution*", Megane, Lourenco, and Machado introduce Probabilistic Grammatical Evolution (PGE) that introduces a new approach to representing computer programs and a new mapping mechanism for Grammatical Evolution (GE), a GP variant. PGE utilizes a Probabilistic Context-Free Grammar (PCFG), a Context-Free grammar with the additional set that associates each production rule with a probability. The Probabilistic Grammatical Evolution uses the PCFG to map a genotype, a list of probabilities, to a computer program. PGE will then modify the probabilities of the PCFG according to the fittest individuals produced.

More specifically, PGE begins by initializing a PCFG where all its derivation rules in the grammar have the same chance of being selected. At each generation of the genetic programming run, the probabilities of the PCFG are updated based on the number of times each derivation rule was chosen by the best individual of the current generation or the best individual overall. A derivation rule's probability will increase if used to create one of the best individuals. Otherwise, its probability will decrease if not used in the individual's creation. This perturbation process allows PGE to generate high-fit individuals throughout the evolutionary process.

Their experimentation involved performing the PGE on two regression problems and comparing it with GE and Structure Grammatical Evolution (SGE). They found that PGE performed better than GE and had similar results to SGE on these problems.

In contrast to this work, our thesis utilizes a probabilistic genome that associates probabilities to each instruction in the genome rather than assigning probabilities to derivation steps. Our work also presents a new mutation method that increases the likelihood of perturbing a probabilistic genome's probability upward if the associated instruction helped generate the best non-probabilistic genome that the probabilistic genome could express.

# Chapter 2

# Design and Implementations

## 2.1 Software Implementation

### 2.1.1 Introduction to Propeller

Propeller is a Push-based genetic programming system written in the Clojure programming language (a JVM-based dynamic and functional dialect of Lisp). This GP system is often used in researching novel genetic programming techniques due to its flexible design that allows human users to specify a wide range of run parameters, such as population size and generation limit. Another benefit to Propeller's adaptability is that it allows us to easily integrate new genetic representations of computer programs and genetic operators into the system to help further investigate our study of probabilistic genomes.

The Propeller GP system solves various genetic programming problems by outputting computer programs created in the Push programming language. Propeller includes Clojure implementations of many Push instructions to create new Push programs. Additionally, Propeller utilizes a Clojure-based Push interpreter that can execute a Push program and thus help it to evaluate the errors of Push programs based on a set of inputs. We wish to reiterate that Propeller is written in Clojure, while the computer programs it evolves and ultimately generates to solve a given problem consist entirely of Push instructions.

The main component of Propeller is the genetic programming algorithm, as it simplifies the biological evolutionary process into its essential steps. Given a benchmark problem, Propeller will initialize a population of random candidate solutions and then determine how to measure the errors of each individual in the population. After that, the system will select the Push programs in the population with lower errors and choose which variation methods to use to create child programs. Propeller then decides which Push programs will form the new population for the next generation of the genetic programming run.

Additionally, Propeller provides an intuitive interface that allows a human programmer to specify benchmark problems and customize run parameters. Benchmark problems are the set of problems used to measure the performance of a genetic programming system. Propeller's design helps a human user to describe a benchmark problem and what type of computer program it should output based on a genetic source of Push instructions. Additionally, the system enables a human user to input an objective function and training set to measure a candidate solution's errors. Human programmers can also introduce new genetic programming methods into Propeller that are system-compatible with its genetic programming logic.

### 2.1.2 The Push Programming Language

Push is a programming language designed specifically for genetic programming. Push utilizes a stack-based execution architecture with a separate stack for each data type (e.g. integer, float, boolean, string, vectors of integers, vectors of booleans, etc). Furthermore, the programming language contains types and instructions that permit run-time manipulation and execution of code.

Push executes an instruction by popping inputs from the top of the designated data stacks and pushing output onto the appropriate-typed stack. Each Push instruction targets specified data stacks to obtain its inputs and to also insert its output. With this implementation, Push programs can even use a variety of control structures.

By combining this expressiveness with the only syntax rule for a Push program being balanced parentheses, Push allows Propeller to generate syntactically valid Push programs. Genetic programming systems often struggle evolving computer programs written in other languages due to their proneness to compile errors when modifying them without following strict syntax rules. Push, however, stresses execution safety, which results in any newly formed Push program being able to execute without errors. This defining feature of the programming language helps Propeller to freely combine different instructions and create Push programs that can evolve in the genetic programming run.

### 2.1.3 Plushy Genomes

Propeller uses Plushy genomes to represent Push programs. A Plushy is a linear data structure consisting of a sequence of literals and Push instructions. Because of a Plushy genome's simple and flexible structure, it is easy for the GP system to manipulate and generate new Plushies. Propeller primarily interacts with Plushy genomes for a majority of the evolutionary process. As a result, Propeller indirectly evolves Push programs by creating and continuously improving upon a population of Plushy genomes.

(5 x int_gt CLOSE exec_if x SKIP int_sqrt CLOSE x 0.13 2 int_mult)

Figure 2.1: A (Non-Probabilistic) Plushy Genome

Propeller only has direct interactions with Push programs during the evaluation process to measure the errors of a Plushy genome. The evaluation of a Plushy genome's quality of errors begins with Propeller performing a translation process to map the Plushy to its corresponding Push program. The GP system will then use inputs from the training set on the specified objective function and program to generate the expected and predicted output, respectively. The error on that given input is the absolute value of the difference between these two outputs. Propeller collects the set of errors for each input in the training set and assigns them to the Plushy genome. Since the Plushy genome is the genetic representation of a Push program, the genome's errors can be viewed as the translated program's errors.

(5 x int_gt CLOSE exec_if x SKIP int_sqrt CLOSE x 0.13 2 int_mult) → (5 x int_gt exec_if (3 x int_sub) (x 2 int_mult))

Figure 2.2: Translation from Plushy Genome to Push Program

For our investigation, we developed probabilistic genomes based on the design of Plushy genomes with the addition where each gene instruction is associated with some probability. With this new representation, we can develop a new mutation method that targets these probabilities rather than the Push instructions within a Plushy genome. We provide more details about this new genome representation of Push programs and variation methods in sections 2.3 and 2.5.1, respectively.

## 2.2 Genetic Operators

### 2.2.1 Selection

We use epsilon-lexicase selection to choose parent Plushies in the population to produce child Plushy genomes during the reproductive process. Epsilon-lexicase selection is a variant of lexicase selection where it uses test cases to filter the population so that the best-performing individuals remain in the selection pool. Lexicase selection repeats this process until a single candidate solution remains and is the individual selected to reproduce child programs. Epsilon-lexicase selection is different from lexicase selection because it relaxes the lexicase filtering step by removing individuals who fall outside of some epsilon of the best at each training case (La Cava, Spector, Danai 2016). As a result, this change in the filtering process softens the selective pressure from lexicase selection, allowing epsilon-lexicase selection to select individuals that are still doing well in specific cases.

The epsilon-lexicase selection algorithm proceeds as follows:

1. Initialize
    - a. Set *selection pool* to be the entire population of programs.
    - b. Set *cases* to be a list of all the test cases in the training set in random order.
2. Loop
    - c. Set *t* to be the first test case in cases.
    - d. Set *best* to the best error value of any individual in *selection pool* on *t*.
    - e. Calculate *epsilon* by performing the median absolute deviation of the population on *t*.
    - f. Filter *selection pool* to include only individuals within *epsilon* of *best*.
    - g. If *selection pool* contains just a single individual then return it.
    - h. If *cases* contain just a single test case then return a randomly selected individual from the pool.
    - i. Otherwise pop *t* from *cases* and go to loop.

We implemented epsilon-lexicase selection into Propeller to use in our experiments. Epsilon-lexicase is known for its effectiveness on symbolic regression problems as it can produce more accurate models than popular selection methods. We plan to use epsilon-lexicase selection as our benchmark suite consists entirely of float regression problems.

### 2.2.2 Variation

In addition to our new variation method that targets a probabilistic genome's probabilities rather than its instructions, we will also use another mutation operator called uniform mutation by addition and deletion (UMAD) in our genetic programming runs. In their 2018 paper, "*Program Syntheis using Uniform Mutation by Addition and Deletion*", Helmuth, McPhee, and Spector describe UMAD as a single-parent variation method that creates a child Plushy from a parent genome by performing two sweeps to add and delete gene instructions. The first sweep will add random genes before or after each existing gene in a given genome, with the second randomly deleting gene instruction from the resulting genome. The mutation operator performs the addition and deletion processes independently from each other, allowing it to be a less restricted form of mutation compared to traditional mutation methods.

UMAD allows a user to specify the rate of addition and the rate of deletion performed when creating a child Plushy from a parent Plushy. When the addition rate is higher than the deletion rate, the genome sizes will continue to increase throughout the evolutionary process. On the other hand, a larger deletion rate will cause the Plushy size to shrink after many generations. To implement a size-neutral UMAD, where the mutation method, on average, deletes the same amount of genes as it adds, a user can calculate the appropriate deletion rate based on the following equation that takes the addition rate as input: $\text{deletion rate} = \frac{\text{addition rate}}{1 + \text{addition rate}}$.

## 2.3 Probabilistic Plushy genomes

Probabilistic Plushy genomes are linear data structures that consist of Push instructions and constant values, similar to Non-Probabilistic Plushy genomes. The difference, however, is that each gene instruction in the Probabilistic Plushy genome is associated with a number within the interval [0,1]. As a result, each element in a Probabilistic Plushy is a tuple of a Push instruction and a corresponding float number. A Probabilistic Plushy can express a Non-Probabilistic Plushy genome by initially creating an empty Non-Probabilistic Plushy. After that, we iterate through each gene in the Probabilistic Plushy and use these float values as probabilities of adding the associated Push instruction into the Non-Probabilistic Plushy. As such, an expressed Non-Probabilistic Plushy consists of sub-sequences of the Push instructions from a

Probabilistic Plushy.

([5 0.32] [x 0.12] [int_gt 0.43] [CLOSE 0.23] [exec_if 0.76] [x 0.53] [SKIP 0.23] [int_sqrt 0.34] [CLOSE 0.32] [x 0.13] [2 0.67] [int_mult 0.93])

Figure 2.3: A Probabilistic Plushy Genome

The conversion from a Probabilistic Plushy genome to a Push program begins by expressing a Non-Probabilistic Plushy out of a Probabilistic Plushy. Once created, the process uses the operation discussed in section 2.1.3 to translate the Non-Probabilistic Plushy into a Push program. Since Probabilistic Plushy genomes generate Non-Probabilistic Plushies according to their probabilities, expressing the genome multiple times may form different Non-Probabilistic Plushies and thus translate to various Push programs. Therefore, unlike a Non-Probabilistic Plushy, a Probabilistic Plushy genome maps to more than one Push program.

([5 0.32] [x 0.12] [int_gt 0.43] [CLOSE 0.23] [exec_if 0.76] [x 0.53] [SKIP 0.23] [int_sqrt 0.34] [CLOSE 0.32] [x 0.13] [2 0.67] [int_mult 0.93])

↙ ↘

(5 exec_if int_sqrt x 2 int_mult)   (x int_gt CLOSE exec_if x int_mult)

Figure 2.4: Expressing Multiple Non-Probabilistic Plushies from a Probabilistic Plushy

## 2.4 Multiple Evaluations

Normally, Propeller will evaluate the errors of a Non-Probabilistic Plushy by translating it to a Push program and using inputs from a training set to measure the absolute value of the difference between the expected outputs and the program's predicted outputs. With Probabilistic Plushies, however, their instructions are not deterministic since each one has a chance of not being expressed. In other words, the errors of a Probabilistic Plushy are not constant because it can translate into multiple Non-Probabilistic Plushies, and thus many computer programs, with different errors. As a result, the current evaluation method used to calculate the errors of a Non-Probabilistic Plushy is not effective in determining the quality of errors of a Probabilistic Plushy.

The multiple evaluations method attempts to measure the errors of a Probabilistic Plushy by associating it with the errors of a Non-Probabilistc Plushy it can express. More specifically, the approach takes a Probabilistic Plushy as input and generates a specified number of Non-Probabilistic Plushies from its instructions. We then run Propeller's evaluation process on these expressed Non-Probabilistic Plushies by translating them into Push programs and calculating each one's respective errors. After that, we sum each Non-Probabilisitc Plushy's errors to obtain their total error and find the genome with the minimum total error from the group of expressed Plushies. The process then assigns the errors of the selected Plushy to represent the errors of the Probabilistic Plushy. The Non-Probabilistic Plushy with this minimal total error is also associated with the Probabilistic Plushy for later use during the variation phase. As such, we can judge how "good" a Probabilistic Plushy will be in solving a given problem according to its set of expressible Non-Probabilistic Plushies.

With this approach, selection requires no changes when Propeller uses Probabilistic Plushies, as each genome's errors are associated with the errors of an expressed Non-Probabilistic Plushy. Similar to how Propeller performs the selection process with Non-Probabilistic Plushies, the GP system chooses the Probabilistic Plushies in the population to move on to the variation phase based on their errors. In the next section, we will show how the multiple evaluations method helps to perturb the probabilities of a Probabilistic Plushy.

$$\begin{tikzcd}
 & \text{([5 0.32] [x 0.12] [int\_gt 0.43] [CLOSE 0.23] [exec\_if 0.76] [x 0.53] [SKIP 0.23] [int\_sqrt 0.34] [CLOSE 0.32] [x 0.13] [2 0.67] [int\_mult 0.93])} \arrow[dl] \arrow[dr] & \\
\text{(5 exec\_if int\_sqrt x 2 int\_mult)} \arrow[d] & & \text{(x int\_gt CLOSE exec\_if x int\_mult)} \arrow[d] \\
\text{errors 1} \arrow[dr] & & \text{errors 2} \arrow[dl] \\
 & \text{total error 1} < \text{total error 2} \arrow[d] & \\
 & \{\text{errors = errors 1, Associated Non-Probabilistic Plushy = (5 exec\_if int\_sqrt x 2 int\_mult)}\} &
\end{tikzcd}$$

Figure 2.5: Multiple Evaluations Method on a Probabilistic Plushy using 2 expressions

## 2.5 New Genetic Operators

### 2.5.1 UMAD

We implemented a version of UMAD that was compatible with Probabilistic Plushy genomes. Similar to the UMAD used on Non-Probabilistic Plushy genomes, our new UMAD performs uniform addition across a Probabilistic Plushy by inserting a new gene instruction before or after each element in the Probabilistic Plushy based on the UMAD rate. The difference is that the gene instruction is instead a Push instruction with an associated probability. Uniform deletion, the second phase of UMAD, requires no changes to be used in the updated version of UMAD since the type of gene instruction does not impact the function's deletion logic.

### 2.5.2 Biased-perturbation mutation

Originally, we created a mutation function that performed minor tweaks to the probabilities of a Probabilistic Plushy to maintain behavior similarity between parent and child Probabilistic Plushy genomes. Given a Probabilistic Plushy, the method perturbs each instruction's probability by adding the values with some "random" amount. This "random" number was the product of some specified standard deviation multiplied by a Gaussian noise factor of mean 0 and standard deviation 1. The function returns a child Probabilistic Plushy genome with slightly different probabilities from its parent's probabilities.

Since the parent and child Probabilistic Plushy genomes share the same sequence of instructions with slight variations in their probabilistic values, they often generate the same set of expressible Non-Probabilistic Plushies. As a result, these genomes can generate Non-Probabilistic Plushies that produce comparable outputs when evaluated on a training set, meaning that the parent and child genomes have similar behaviors. The slight semantic difference between the genomes is that one has a higher chance of creating a particular Non-Probabilistic Plushy from the set than the other. In traditional GP mutation methods, adding/deleting/changing instructions creates a big effect that causes the child to "act" differently than its parent. By just perturbing the probabilities of genomes rather than their instructions, we can produce a smaller effect of variation that allows parent and child candidate solutions to share behavioral traits.

Although the original idea helped parent and child genomes behave similarly, we wanted an approach that also aimed to adjust the probabilities based on the corresponding instruction's contribution in expressing a Non-Probabilistic Plushy with low errors. In other words, if we knew a Probabilistic Plushy could express some Non-Probabilistic Plushy with low total error, the child Probabilistic Plushy should be biased in expressing that particular Non-Probabilistic Plushy. The reasoning behind this approach was that a Probabilistic Plushy's instruction should be more likely to be expressed if it appears in Non-Probabilistic Plushies with low errors. On the other hand, if an instruction in a Probabilistic Plushy often appears in an expressed Non-Probabilistic Plushies with a high total error, then that instruction should have a decreased chance of being expressed in the mutated Probabilistic Plushy. Despite slightly decreasing the probabilities of instructions that do not seem relevant to the given problem, we still keep these instructions in the Probability Plushy in the chance that they can help generate a Non-Probabilistic Plushy with minimal total error.

The biased-perturbation mutation (BPM) method first extracts the information added from the multiple evaluation function to obtain the expressed Non-Probabilistic Plushy that represents the Probabilistic Plushy's errors. The method will then iterate each Push instruction in the Probabilistic Plushy and check if it is in the Non-Probabilistic Plushy. Depending on if the instruction appeared in the Non-Probabilistic Plushy, the method will add or subtract by a "random" amount. Once again, the "random" value is a product of some standard deviation argument multiplied by a Gaussian noise factor of mean 0 and standard deviation 1. This approach will cause the expressed instructions to be more likely or biased to have their probabilities increase than the ones that do not appear in the Non-Probabilistic Plushy. By applying this method over numerous generations, each instruction's probability will approach 0 or 1, with the sequence of instructions with high nonzero probabilities expressing to a Non-Probabilistic Plushy with low total error.

([5 0.32] [x 0.12] [int_gt 0.43] [CLOSE 0.23] [exec_if 0.76] [x 0.53] [SKIP 0.23] [int_sqrt 0.34] [CLOSE 0.32] [x 0.13] [2 0.67] [int_mult 0.93])

$\downarrow$

Associated Non-Probabilistic Plushy = (5 exec_if int_sqrt x 2 int_mult)

$\downarrow$

([5 0.33] [x 0.09] [int_gt 0.47] [CLOSE 0.18] [exec_if 0.79] [x 0.45] [SKIP 0.25] [int_sqrt 0.37] [CLOSE 0.25] [x 0.15] [2 0.62] [int_mult 0.95])

Figure 2.6: Biased-perturbation mutation performed on a Probabilistic Plushy Genome

## 2.6 Data Collection

### 2.6.1 Benchmark Problems

To measure Propeller's performance using Probabilistic Plushies and Non-Probabilistic Plushies, we selected seven float regression problems for testing. In a regression problem, the goal for GP is to create a computer program that most accurately fits a given data distribution. These problems require a training set of inputs and an objective function, often a mathematical expression, that takes these inputs to produce an output. Propeller uses these two components throughout the evolutionary process to create a program such that given a number from the training set as input, the predicted output from the program matches the expected output from the objective function. We then evaluate the program with a testing set of unseen data to determine if the program generalizes well to new data. If the program's predicted output is equivalent to the objective function's expected output using inputs from the testing set, then the program is a good fit for the data distribution.

The symbolic regression problems we used for experimentation were float regression problems, where the variable type of the inputs and outputs are float numbers. The training set of inputs we used for the benchmark problems were the numbers from -1.5 to 1.5 in steps of 0.1, totaling $(1.5 - (-1.5))/0.1 = 30$ inputs. With our testing set, we decided on the group of numbers from -1.75 to 1.75 in increments of 0.05, which results in $(1.75 - (-1.75))/0.05 = 70$ inputs.

We normally split the training and testing sets into two separate groups of data inputs. In our case, however, we used a training set that was a subset of our testing set, which may appear odd as we generally want our testing set to consist of inputs a program has not seen before. We believe the testing set we defined is appropriate for our experimentation because we still provide a candidate program with unseen data to evaluate on. Additionally, a program that has a high total error on the training set will also not perform with the testing set, meaning that the program will not generalize well to the data distribution.

As for the objective functions, we chose the mathematical formula $f(x) = (1 + x^3)^3 + 1$ and similar variants to it to represent the outputs of the training set. By similar variants, we mean a combination of additional terms and changes to the coefficients of the previously shown objective functions. Table 2.1 provides the list of objective functions used for our float regression problems.

| |
|---|
| $(1 + x^3)^3 + 1$ |
| $(1 + x + x^2)^3 + 4$ |
| $(1 + x^2)^3 + 1$ |
| $(1 + x^3)^4 + 1$ |
| $(2 + x^3)^3 + 1$ |
| $(1 + x + x^4)^3 + x$ |
| $(1 + x^3)^3 + x^2 + 3$ |

Table 2.1: Objective functions used in benchmark suite

We decided to use problems with similar objective functions rather than a more diverse set of symbolic regression problems or problems from different domains because it allowed us to observe subtle differences in Propeller's performance with Probabilistic Plushies versus using Non-Probabilistic Plushies. Another reason we chose these problems as our benchmark suite is that they allowed Propeller to output solutions within a reasonable period, allowing us to execute multiple runs of the same problem. Often, more difficult GP problems require several days for Propeller to produce a solution program, which limits the number of runs we can perform in our experimentation.

### 2.6.2 Experimental Design

#### Preliminary Experimentation

| | Non-Probabilistic | Probabilistic |
|---|---|---|
| Generations | 500 | 500 |
| Population | 500 | 500 |
| Number of Evaluations Per Plushy | 1 | 10 |
| Multiples Evaluations Std | - | 0.2 |
| Max initial Plushy Size | 100 | 100 |
| Solution Error Threshold | 0.1 | 0.1 |
| Step Limit | 200 | 200 |
| Parent-Selection | Epsilon-lexicase | Epsilon-lexicase |
| Variation | {100% UMAD} | {5% UMAD, 95% Biased-Perturbation Mutation} |
| UMAD addition rate | 0.1 | 0.1 |
| UMAD deletion rate | 0.082 | 0.082 |
| Elitism | False | False |

Table 2.2: Non-Probabilistic and Probabilistic Run Parameters

We will begin with a preliminary experiment that runs Propeller on the symbolic regression problems discussed in section 2.6.1 using non-probabilistic and probabilistic run parameters. Each configuration uses a particular type of Plushy genome, non-probabilistic or probabilistic, and other Propeller system parameters such as population size and generation limit. Additionally, we include specific genetic operators and error evaluation processes for each set of run parameters. Table 2.2 shows the non-probabilistic and probabilistic settings we used in our initial experimentation. Additionally, section 6.1 shows the instruction set we used for our preliminary experimentation.

Based on the run parameters presented in Table 2.2, one could argue about the difference in the total number of evaluations given to Probabilistic Plushies and Non-Probabilistic Plushies. During the error evaluation process of Probabilistic Plushies in the population, the system performs ten evaluations on a genome when determining the appropriate error value to assign it. On the other hand, each Non-Probabilistic Plushy only needs to be evaluated once when measuring its errors. As a result, the multiple evaluations method in the probabilistic run parameters causes Propeller to execute additional error evaluations when using Probabilistic Plushy genomes than Non-Probabilistic genomes, which results in an unfair comparison of the GP system's performances using the non-probabilistic and probabilistic settings.

| | Non-Probabilistic (generations) | Non-Probabilistic (population size) |
|---|---|---|
| Generations | **5000** | 500 |
| Population | 500 | **5000** |
| Number of Evaluations Per Plushy | 1 | 1 |
| Multiples Evaluations Std | - | - |
| Max initial Plushy Size | 100 | 100 |
| Solution Error Threshold | 0.1 | 0.1 |
| Step Limit | 200 | 200 |
| Parent-Selection | Epsilon-lexicase | Epsilon-lexicase |
| Variation | {100% UMAD} | {100% UMAD} |
| UMAD addition rate | 0.1 | 0.1 |
| UMAD deletion rate | 0.082 | 0.082 |
| Elitism | False | False |

Table 2.3: Non-Probabilistic Run Parameters (Equal Total Number of Error Evaluations)

In Table 2.2, the probabilistic settings allow Propeller to perform at most 10 error evaluations per candidate solution x 500 candidates solutions x 500 generations = 2,500,000 error evaluations. With the current non-probabilistic settings, there can be 1 error evaluation per candidate solution x 500 candidate solutions x 500 generations = 250,000 possible evaluations executed in a Propeller run. To equalize the total number of error evaluations in the probabilistic configuration, we created two additional non-probabilistic run parameters that either increased the generation limit or the population size by a factor of 10. Table 2.3 shows these new non-probabilistic settings we will include in our preliminary experimentation. We still plan to run Propeller with our initial non-probabilistic settings to determine if Propeller, using the probabilistic run parameters, can at least outperform itself given such settings despite this unfair advantage in the total number of evaluations.

For each of the seven float regression problems and configurations presented in this section, we perform 100 runs of Propeller on the given problem using one of the settings from Tables 2.2 and 2.3. In total, our experiment will consist of 7 problems x 100 runs per problem x 4 different run parameters = 2400 runs. We will perform these 2400 runs on the Amherst College High-Performance Computing System (HPC) as it can handle the computational-intensive runs from our Propeller experiments and allows us to collect necessary data. The system relies on the Slurm job management program to control the list of runs queued in the cluster. For each of the 2400 runs, we will have Propeller output a solution program to a given problem, the number of generations it needed to find the program, and the total error of the program when evaluated on a testing set.

We will use the success rate to gauge Propeller's performance using Non-Probabilistic Plushies and Probabilistic Plushies. Given a benchmark problem, the success rate is the number of successes out of the total number of times we run Propeller on it. We consider a Propeller run successful if the system finds a candidate solution in the population with a total error of at most 0.1 after executing the error evaluation process. Our termination criterion accepts a non-zero total error because we accounted that marginal discrepancies between an objective function and solution program may occur due to the nature of performing arithmetic with float numbers. Since all our experiments involve executing 100 runs of Propeller on a benchmark problem, the success rate will be the same as the number of successes out of the 100 runs. Although there are many other metrics we could have used, such as the average run time or the average number of generations of successful runs, the success rate metric helps us to directly compare Propeller's solving capabilities using Non-Probabilistic Plushies versus Probabilistic Plushies.

### Genetic Sources

After our preliminary runs, we wanted to investigate how Probabilistic Plushy genomes would impact Propeller's performance on different genetic sources. A GP system's success rate on a problem is known to suffer when given a more generic instruction set. The reason is that a computer program often contains "bad " instructions that do not contribute to or even decrease the program's solving power on a given problem. We hypothesize that Probabilistic Plushies can improve Propeller's performance on generic instruction sets due to its ability to express Non-Probabilistic Plushies. More specifically, unlike Non-Probabilistic Plushies, Probabilistic Plushies can effectively eliminate "bad" instructions within their structures by using their probabilities to generate Non-Probabilities that contain important instructions to solve a given problem. Combined with the biased-perturbation mutation method, Probabilistic Plushies can pass down these behavioral traits to new genomes, helping them to express Non-Probabilistic Plushies with such instructions.

We created two genetic sources that differ by their generality to float regression problems. The first instruction set, the hand-tuned genetic source, was tailored specifically for our benchmark suite, consisting of function input, float operations, and constants. For our second instruction set, we threw “everything but the kitchen sink” into the genetic source to make it much more generic than the hand-tuned genetic source. In other words, the “kitchen-sink” genetic source contains multiple data type instructions (e.g. string, boolean, integers, floats, etc), making it applicable to many problem domains. Section 6.2 lists the instructions included in the hand-tuned and kitchen-sink genetic sources.

| | Prob (5% UMAD, 95% BPM) | Prob (20% UMAD, 80% BPM) | Prob (30% UMAD, 70% BPM) | Prob (20% Crossover, 20% UMAD, 60% BPM) |
|---|---|---|---|---|
| Generations | 500 | 500 | 500 | 500 |
| Population | 500 | 500 | 500 | 500 |
| Number of Evaluations Per Plushy | 10 | 10 | 10 | 10 |
| Multiples Evaluations Std | 0.2 | 0.2 | 0.2 | 0.2 |
| Max initial Plushy Size | 100 | 100 | 100 | 100 |
| Solution Error Threshold | 0.1 | 0.1 | 0.1 | 0.1 |
| Step Limit | 200 | 200 | 200 | 200 |
| Parent-Selection | Epsilon-lexicase | Epsilon-lexicase | Epsilon-lexicase | Epsilon-lexicase |
| Variation | {**5% UMAD, 95% BPM**} | {**20% UMAD, 80% BPM**} | {**30% UMAD, 70% BPM**} | {**20% Crossover, 20% UMAD, 60% BPM**} |
| UMAD addition rate | 0.1 | 0.1 | 0.1 | 0.1 |
| UMAD deletion rate | 0.082 | 0.082 | 0.082 | 0.082 |
| Elitism | False | False | False | False |

Table 2.4: Probabilistic Run Parameters (Genetic Sources Experiment)

Along with new genetic sources, we also implemented additional probabilistic settings to observe how Propeller performs using different combinations of genetic operators to create new Probabilistic Plushies. We developed three more probabilistic configurations based on the probabilistic run parameters from preliminary experimentation. Each probabilistic configuration differs by the percentage of UMAD and the percentage of the biased-perturbation mutation used in a Propeller run and holds all other run parameters constant.

We decided to reconfigure another probabilistic setting to incorporate crossover into the genetic operators used on Probabilistic Plushies, which already included UMAD and the biased-perturbation mutation method. Our probabilistic settings until now have all mainly focused on the mutation side of the variation phase. We wanted to observe and find if Propeller would benefit the system’s performance on benchmark problems by introducing crossover into its run parameters when using Probabilistic Plushies. Table 2.4 shows the four new probabilistic run parameters and highlights the ratios of each variation method used in a given probabilistic setting.

The set of run parameters we will use for this experiment are the settings used in our initial testing, tables 2.2 and 2.3, and the run parameters presented in table 2.4. For each problem in our benchmark suite and configuration in our set of run parameters, we will perform 100 Propeller runs on a given problem using either the hand-tuned or kitchen-sink genetic source. This experiment results in conducting 7 problems x 100 runs per problem x 7 run parameters = 4900 runs. As mentioned in our last section, we will execute these runs on the HPC and have Propeller produce the same output. Additionally, we will use the success rate to gauge Propeller's performance on our benchmark suite.

# Chapter 3

# Results

## 3.1 Preliminary Experimentation

| Objective function | Non-Probabilistic Plushy | Non-Probabilistic Plushy (generation) | Non-Probabilistic (population size) | Probabilistic Plushy |
|---|---|---|---|---|
| $(1+x^3)^3+1$ | 24/24 | 43/43 | 50/50 | 85/85 |
| $(1+x+x^2)^3+4$ | 9/9 | 19/19 | 23/23 | 21/21 |
| $(1+x+x^4)^3+x$ | 0/0 | 0/0 | 0/0 | 0/0 |
| $(1+x^3)^3+x^2+3$ | 0/0 | 1/1 | 1/1 | 2/2 |
| $(1+x^2)^3+1$ | 81/81 | 100/100 | 100/100 | 91/91 |
| $(2+x^3)^3+1$ | 15/15 | 38/38 | 28/28 | 20/20 |
| $(1+x^3)^4+1$ | 5/5 | 48/48 | 32/32 | 40/40 |
| Total | 134/134 | 249/249 | 234/234 | 259/259 |

Table 3.1: Number of Successes out of 100 Independent Runs (training set/testing set)

Table 3.1 shows Propeller's success rates on each of the benchmark problems when using the set of run parameters discussed in section 2.6.2. The successes on the left of the "/" are the number of solution programs that are under the error threshold 0.1 when run on the training set. Since the testing set has more test cases than the training set, we calculated the testing error threshold using the training error threshold per individual training example to obtain 70 (testing cases) $\times \frac{0.1 \text{ (training error threshold)}}{30 \text{ (training cases)}} =$ 0.23 (testing error threshold). As a result, the values on the right of the "/" indicate the solutions within our determined error threshold when run on the testing set. Based on our observations of the number of successes on the testing set, Non-Probabilistic and Probabilistic Plushy genomes generalize well on unseen data.

| Objective function | Non-Probabilistic Plushy | Non-Probabilistic Plushy (generation) | Non-Probabilistic (population size) | Probabilistic Plushy |
|---|---|---|---|---|
| $(1+x^3)^3+1$ | 30.71 | 28.47 | 36.97 | 78.76 |
| $(1+x+x^2)^3+4$ | 28.26 | 23.4 | 28.54 | 79.35 |
| $(1+x+x^4)^3+x$ | 24.42 | 26.89 | 26.87 | 78.34 |
| $(1+x^3)^3+x^2+3$ | 30.53 | 27.17 | 29.66 | 80.66 |
| $(1+x^2)^3+1$ | 30.94 | 32.55 | 46.02 | 76.32 |
| $(2+x^3)^3+1$ | 34.49 | 29.24 | 32.58 | 82.02 |
| $(1+x^3)^4+1$ | 28.71 | 25.32 | 31.93 | 76.6 |

Table 3.2: Average Non-Probabilistic Plushy and Probabilistic Genome Size out of 100 Independent Runs

| Objective function | Successful Runs | Unsuccessful Runs |
|---|---|---|
| $(1+x^3)^3+1$ | 0.37 | 0.41 |
| $(1+x+x^2)^3+4$ | 0.37 | 0.40 |
| $(1+x+x^4)^3+x$ | - | 0.41 |
| $(1+x^3)^3+x^2+3$ | 0.43 | 0.39 |
| $(1+x^2)^3+1$ | 0.36 | 0.40 |
| $(2+x^3)^3+1$ | 0.41 | 0.40 |
| $(1+x^3)^4+1$ | 0.39 | 0.40 |

Table 3.3: Average Proportion of Probabilities Equal to 0

A noteworthy observation of Probabilistic Plushy genomes is that they helped Propeller perform better on the first benchmark problem than when the GP system used any other non-probabilistic configuration. When comparing Propeller's success rates using the probabilistic settings to the non-probabilistic run parameters in Table 2.2, we found that the probabilistic runs had a larger number of successes than the non-probabilistic runs for a majority of the benchmark suite. This initial comparison shows us that Probabilistic Plushy genomes can improve Propeller's performance on GP problems.

When equalizing the total number of error evaluations for the non-probabilistic and probabilistic runs, Propeller appears more effective in solving our benchmark problems using the non-probabilistic settings instead of the probabilistic run parameters. We observed that the non-probabilistic runs with additional generations and a bigger population size had a higher success rate than the probabilistic runs on most of our symbolic regression problems, with either, if not both, of the non-probabilistic settings resulting in Propeller having more success than when it uses the probabilistic settings. However, we found that Propeller performs better on the entire benchmark suite using Probabilistic Plushy genomes instead of

Non-Probabilistic Plushy genomes when totaling the number of successes for their respective columns. While probabilistic genomes can improve a GP system's success rate on a given problem, it is still unclear if we can generalize this increase in performance with error evaluations.

We wish to note that these non-probabilistic runs took much longer to finish than the probabilistic runs, particularly the non-probabilistic settings where we increased the population size. Although we do not have specific runtime metrics, both non-probabilistic configurations required Propeller to run for multiple days when attempting to find a solution program. These runtimes were much longer than the probabilistic runs, which only needed a few hours. As a result, we can claim that Probabilistic Plushy genomes are more practical when solving GP problems than Non-Probabilistic Plushy genomes with additional error evaluations due to their runtimes.

Table 3.2 displays the average genome size for each benchmark problem and configuration. We chose the solution Plushy in a successful run for our calculation of the average Plushy genome length. On the other hand, since the unsuccessful runs did not produce a solution program, we decided to use the genome in the population with the least total error at the final generation of the Propeller run when calculating the average genome size.

We observed that the average genome sizes were fairly consistent across all benchmark problems and run parameters. The average genome size for the Non-Probabilistic Plushy genomes appears to range around 20-30, with the highest average genome length being 46.02.

We found that the average genome size for Probabilistic Plushy genomes was much larger than Non-Probabilistic Plushy genomes. Similar to the non-probabilistic results, the average genome length also seems consistent across the benchmark suite. We believe that the average Probabilistic Plushy genome size is large due to the number of gene instructions with a probability equal to 0 in a Probabilistic Plushy genome.

Table 3.3 shows the average proportion of the probabilities of the Probabilistic Plushies equal to 0. The numbers on the left of "/" are the successful runs, while the values on the right are the unsuccessful runs. The '-' in the table serves to indicate that Propeller had no successful runs on a particular benchmark problem using specific run parameters. We see that about a third of a Probabilistic Plushy contains genes with a probability of 0, with higher percentages occurring in the unsuccessful runs. We reason that removing these genes can help reduce the average Plushy length for Probabilistic Plushies.

We wish to note that we are presenting unsimplified programs in our tables. Propeller is capable of reducing a Plushy genome into a more concise form through a simplification process, resulting in the genome's size decreasing. Since we did not perform the simplification of evolved genomes in our GP runs, the average sizes are much bigger than required. We suggest measuring the average size of simplified genomes in future experiments with Probabilistic Plushy genomes.

| Objective function | Successful Runs | Unsuccessful Runs |
|---|---|---|
| $(1+x^3)^3+1$ | 0.76 | 0.81 |
| $(1+x+x^2)^3+4$ | 0.76 | 0.80 |
| $(1+x+x^4)^3+x$ | - | 0.80 |
| $(1+x^3)^3+x^2+3$ | 0.77 | 0.79 |
| $(1+x^2)^3+1$ | 0.75 | 0.81 |
| $(2+x^3)^3+1$ | 0.78 | 0.80 |
| $(1+x^3)^4+1$ | 0.77 | 0.80 |

Table 3.4: Average Proportion of Deterministic Probabilities (0 or 1)

Table 3.4 presents the average proportion of deterministic probabilities, probabilities equal to 0 or 1, in a Probabilistic Plushy genome for successful and unsuccessful runs. Similar to how we calculated the average Plushy genome size, we used solution programs for the successful runs and the lowest total-error individual in the population at the final generation of a Propeller run for the unsuccessful runs for our analysis. Similar to table 3.3, the '-' in table 3.4 informs us that a benchmark problem had zero successful Propeller runs using a particular setting.

From our observations, many of the probabilities of Probabilistic Plushy genomes for both types of runs were equal to 0 or 1. In addition, these percentages were consistent across the benchmark problems. These results require further investigation to determine how changing the proportion of deterministic probabilities in Probabilistic Plushy genomes impacts Propeller's performance. We also noticed that the unsuccessful runs had a slightly higher average percentage of deterministic probabilities than the successful runs, but we believe this difference is trivial.

## 3.2 Genetic Sources

### 3.2.1 Hand-tuned

| Objective function | Non-Prob | Non-Prob (generations) | Non-Prob (population size) |
|---|---|---|---|
| $(1+x^3)^3+1$ | 49/49 | 84/84 | 67/67 |
| $(1+x+x^2)^3+4$ | 38/38 | 76/76 | 66/66 |
| $(1+x+x^4)^3+x$ | 65/65 | 90/90 | 82/82 |
| $(1+x^3)^3+x^2+3$ | 20/20 | 60/60 | 42/42 |
| $(1+x^2)^3+1$ | 100/100 | 100/100 | 100/100 |
| $(2+x^3)^3+1$ | 79/79 | 99/99 | 88/88 |
| $(1+x^3)^4+1$ | 92/92 | 99/99 | 98/98 |
| Total | 443/443 | 608/608 | 543/543 |

Table 3.5: Number of Successes out of 100 Independent Runs (training set/testing set)

| Objective function | Prob (5% UMAD, 95% BPM) | Prob (20% UMAD, 80% BPM) | Prob (30% UMAD, 70% BPM) | Prob (20% Crossover, 20% UMAD, 60% BPM) |
|---|---|---|---|---|
| $(1+x^3)^3+1$ | 34/34 | 46/46 | 53/53 | 55/55 |
| $(1+x+x^2)^3+4$ | 24/24 | 38/38 | 43/43 | 50/50 |
| $(1+x+x^4)^3+x$ | 20/20 | 40/40 | 44/44 | 58/58 |
| $(1+x^3)^3+x^2+3$ | 11/11 | 6/6 | 11/11 | 14/14 |
| $(1+x^2)^3+1$ | 71/71 | 85/85 | 91/91 | 89/89 |
| $(2+x^3)^3+1$ | 39/39 | 51/51 | 49/49 | 56/56 |
| $(1+x^3)^4+1$ | 67/67 | 86/86 | 87/87 | 90/90 |
| Total | 266/266 | 352/352 | 378/378 | 412/412 |

Table 3.6: Number of Successes out of 100 Independent Runs (training set/testing set)

Tables 3.5 and 3.6 present Propeller's success rates on the benchmark suite using non-probabilistic and probabilistic run parameters and the hand-tuned genetic source. The row header of table 3.6 displays the four probabilistic run parameters we used in our experiment that differ by the combination of genetic operators specified to Propeller. The first three run parameters use some percentage of UMAD and biased-perturbation mutation, while the last probabilistic setting introduces crossover into our Propeller runs with Probabilistic Plushy genomes. The numbers on the left and right of the "/" in these tables refer to the number of successes according to the training error threshold of 0.1 and the testing error threshold of 0.23, respectively. Once again, the non-probabilistic and probabilistic runs generalize well over unseen data.

With the hand-tuned instruction set, Propeller performs better using the non-probabilistic run parameters than the probabilistic configurations on many of the benchmark problems by a large amount. Both the non-probabilistic settings with additional error evaluations had higher successes than all of the probabilistic settings. The non-probabilistic runs where we increased the number of generations appeared to be the overall best run parameters for Propeller runs on the hand-tuned instruction set.

Based on table 3.6, it seems that Propeller had a higher number of successes on the first two benchmark problems using the Prob (30% UMAD, 70% BPM) and Prob (20% Crossover, 20% UMAD, 60% BPM) configurations than when running with the non-probabilistic parameters discussed in table 3.5. Additionally, for the probabilistic runs in table 3.7, we noticed that the success rates of the probabilistic runs trended upward for many of the benchmark problems as we increased the percentage of UMAD and decreased the percentage of the biased-perturbation mutation method. We reason that this surge in the number of successes is possibly due to UMAD diversifying the population more often, allowing Probabilistic Plushies to express new Non-Probabilistic Plushies it could not previously generate.

We also found that the probabilistic run parameters that included crossover in Propeller's genetic operators outperformed all other probabilistic settings across all benchmark problems. From our observation, we believe that Probabilistic Plushies create a relaxed version of crossover due to their ability to express Non-Probabilistic Plushies. We elaborate more about this topic in section 4.1.

| Objective function | Non-Prob | Non-Prob (generations) | Non-Prob (population size) |
|---|---|---|---|
| $(1+x^3)^3+1$ | 57.11 | 48.37 | 52.3 |
| $(1+x+x^2)^3+4$ | 54.06 | 48.39 | 51.61 |
| $(1+x+x^4)^3+x$ | 61.61 | 51.37 | 59.17 |
| $(1+x^3)^3+x^2+3$ | 49.53 | 51.51 | 54.95 |
| $(1+x^2)^3+1$ | 49.53 | 55.46 | 56.26 |
| $(2+x^3)^3+1$ | 61.26 | 58.68 | 50.65 |
| $(1+x^3)^4+1$ | 57.64 | 49.39 | 51.02 |

Table 3.7: Average Non-Probabilistic Plushy Size out of 100 Independent Runs

| Objective function | Prob (5% UMAD, 95% BPM) | Prob (20% UMAD, 80% BPM) | Prob (30% UMAD, 70% BPM) | Prob (20% Crossover, 20% UMAD, 60% BPM) |
|---|---|---|---|---|
| $(1+x^3)^3+1$ | 80.8 | 80.92 | 81.46 | 75.55 |
| $(1+x+x^2)^3+4$ | 80.01 | 80.61 | 85.48 | 78.34 |
| $(1+x+x^4)^3+x$ | 78.99 | 82.17 | 79.1 | 78.21 |
| $(1+x^3)^3+x^2+3$ | 81.31 | 81.8 | 83.43 | 79.14 |
| $(1+x^2)^3+1$ | 76.62 | 72.37 | 73.99 | 73.32 |
| $(2+x^3)^3+1$ | 79.98 | 79.01 | 82.34 | 75.37 |
| $(1+x^3)^4+1$ | 76.65 | 75.07 | 78.31 | 74.02 |

Table 3.8: Average Probabilistic Plushy Size out of 100 Independent Runs

Table 3.7 displays the average Non-Probabilistic Plushy genome size for each benchmark problem and non-probabilistic run parameters. Compared to the average Plushy length in our preliminary experiment, the change to the hand-tuned instruction set increased the size of the Non-Probabilistic Plushy genomes in these Propeller runs. In addition, the values in the table entries are consistent across each benchmark problem and when increasing the number of error evaluations.

| Objective function | Prob (5% UMAD, 95% BPM) | Prob (20% UMAD, 80% BPM) | Prob (30% UMAD, 70% BPM) | Prob (20% Crossover, 20% UMAD, 60% BPM) |
|---|---|---|---|---|
| $(1+x^3)^3+1$ | 0.38/0.38 | 0.34/0.37 | 0.35/0.37 | 0.35/0.36 |
| $(1+x+x^2)^3+4$ | 0.36/0.36 | 0.35/0.36 | 0.36/0.37 | 0.34/0.37 |
| $(1+x+x^4)^3+x$ | 0.34/0.38 | 0.38/0.38 | 0.36/0.36 | 0.37/0.37 |
| $(1+x^3)^3+x^2+3$ | 0.39/0.38 | 0.38/0.37 | 0.37/0.37 | 0.36/0.35 |
| $(1+x^2)^3+1$ | 0.37/0.39 | 0.34/0.37 | 0.36/0.40 | 0.34/0.40 |
| $(2+x^3)^3+1$ | 0.39/0.37 | 0.37/0.37 | 0.37/0.37 | 0.37/0.36 |
| $(1+x^3)^4+1$ | 0.37/0.36 | 0.36/0.36 | 0.34/0.36 | 0.36/0.35 |

Table 3.9: Average Proportion of Probabilities Equal to 0

When observing the average Probabilistic Plushy size in table 3.8, we saw that the Probabilistic Plushy genome lengths are still relatively high for each of the benchmark problems and run parameters. Similarly, we noticed that approximately a third of a Probability Plushy genome contained genes with probabilities equal to 0 across all benchmark problems and probabilistic run parameters. We also found that these sizes are consistent within their respective columns and across each objective function. Interestingly, the average Probabilistic Plushy length decreased across the benchmark suite once we introduced crossover into Propeller's set of mutation operators.

Table 3.10 shows the average proportion of deterministic probabilities for our probabilistic runs using the hand-tuned genetic source. Once again, the percentages on the left and right of the "/" refer to the successful and unsuccessful runs, respectively. Although the differences between the average proportions of deterministic probabilities between the successful and unsuccessful runs are small, it is noteworthy to state that the percentages of the successful runs are always lower than those of the unsuccessful runs. We require additional experiments to understand this apparent trend in table 3.10.

| Objective function | Prob (5% UMAD, 95% BPM) | Prob (20% UMAD, 80% BPM) | Prob (30% UMAD, 70% BPM) | Prob (20% Crossover, 20% UMAD, 60% BPM) |
|---|---|---|---|---|
| $(1+x^3)^3+1$ | 0.77/0.79 | 0.74/0.80 | 0.74/0.80 | 0.74/0.79 |
| $(1+x+x^2)^3+4$ | 0.77/0.78 | 0.75/0.80 | 0.75/0.80 | 0.74/0.81 |
| $(1+x+x^4)^3+x$ | 0.74/0.80 | 0.75/0.80 | 0.73/0.79 | 0.75/0.80 |
| $(1+x^3)^3+x^2+3$ | 0.78/0.79 | 0.78/0.79 | 0.77/0.80 | 0.77/0.80 |
| $(1+x^2)^3+1$ | 0.74/0.79 | 0.72/0.78 | 0.73/0.79 | 0.73/0.82 |
| $(2+x^3)^3+1$ | 0.77/0.80 | 0.74/0.81 | 0.75/0.80 | 0.75/0.80 |
| $(1+x^3)^4+1$ | 0.75/0.79 | 0.73/0.80 | 0.72/0.80 | 0.74/0.80 |

Table 3.10: Average Proportion of Deterministic Probabilities (0 or 1)

### 3.2.2 Kitchen-Sink

| Objective function | Non-Prob | Non-Prob (generations) | Non-Prob (population size) |
|---|---|---|---|
| $(1+x^3)^3+1$ | 5/5 | 13/13 | 9/9 |
| $(1+x+x^2)^3+4$ | 15/15 | 34/34 | 15/15 |
| $(1+x+x^4)^3+x$ | 0/0 | 2/2 | 2/2 |
| $(1+x^3)^3+x^2+3$ | 0/0 | 0/0 | 0/0 |
| $(1+x^2)^3+1$ | 49/49 | 79/79 | 79/79 |
| $(2+x^3)^3+1$ | 2/2 | 9/9 | 9/9 |
| $(1+x^3)^4+1$ | 7/7 | 13/13 | 20/20 |
| Total | 78/78 | 150/150 | 134/134 |

Table 3.11: Number of Successes out of 100 Independent Runs (training set/testing set)

Tables 3.11 and 3.12 present Propeller's number of successes on the benchmark suite using non-probabilistic and probabilistic run parameters and the kitchen-sink genetic source. Similar to our previous results, the non-probabilistic and probabilistic runs generalize well on the testing set. The first notable observation we found from these results is that the success rates drastically decreased using this instruction set in our Propeller runs instead of the hand-tuned genetic source. These findings make sense as the kitchen-sink

| Objective function | Prob (5% UMAD, 95% BPM) | Prob (20% UMAD, 80% BPM) | Prob (30% UMAD, 70% BPM) | Prob (20% Crossover, 20% UMAD, 60% BPM) |
|---|---|---|---|---|
| $(1+x^3)^3+1$ | 11/11 | 31/31 | 37/37 | 72/72 |
| $(1+x+x^2)^3+4$ | 9/9 | 18/18 | 21/21 | 30/30 |
| $(1+x+x^4)^3+x$ | 0/0 | 0/0 | 2/2 | 2/2 |
| $(1+x^3)^3+x^2+3$ | 0/0 | 1/1 | 0/0 | 3/3 |
| $(1+x^2)^3+1$ | 36/36 | 65/65 | 70/70 | 77/77 |
| $(2+x^3)^3+1$ | 0/0 | 4/4 | 6/6 | 25/23 |
| $(1+x^3)^4+1$ | 13/13 | 25/25 | 26/26 | 68/67 |
| Total | 69/69 | 144/144 | 181/181 | 277/274 |

Table 3.12: Number of Successes out of 100 Independent Runs (training set/testing set)

genetic source is much more generic than the hand-tuned instruction set, which causes Propeller's performance to suffer on our benchmark suite. We see that, however, the differences in the number of successes for the non-probabilistic runs are much greater than the probabilistic runs. Therefore, Propeller appears to have a higher dropoff in performance using Non-Probabilistic Plushies than Probabilistic Plushies as we increase the generality of the instructions sets we provide to the system.

Regarding the success rates of our probabilistic runs, we find that the number of successes trends upward for many of the benchmark problems as we increase the UMAD/biased-perturbation mutation ratio. We can also see that introducing crossover into our probabilistic runs results in Propeller performing much better on the first, sixth, and last benchmark problems than when it uses other probabilistic run parameters. This observation provides valuable information into the potential of combining crossover with Probabilistic Plushy genomes.

When comparing the number of successes of the probabilistic runs to the non-probabilistic runs, we can see in the table that Propeller performs better on the first and last benchmark problems when using "Prob(30% UMAD, 70% BPM)" or "Prob(20% Crossover, 20% UMAD, 60% BPM)" run parameters than with any of the non-probabilistic settings. As such, the Probabilistic Plushies genome can help improve the GP system's performance on multiple problems than when the GP system uses Non-Probabilistic Plushies and the number of total error evaluations is the same. In addition, these genomes can produce similar or better success rates than the non-probabilistic runs as we introduce more UMAD and crossover to our probabilistic runs.

From tables 3.13 and 3.14, we observe that running Propeller with the kitchen-sink genetic source causes the average genome size to increase for Non-Probabilistic and Probabilistic Plushies, with the average lengths of the Probabilistic Plushies still being higher than the Non-Probabilistic Plushies. Table 3.15 also shows about the same average proportion of probabilities equal to 0 as the values presented in table 3.9. A notable observation of the average genome sizes of Probabilistic Plushies is that this increase resulted in some of these values exceeding our specified initial maximum genome size of 100. As in our previous findings, the average genome lengths appear consistent across all benchmark problems and configurations in both tables.

| Objective function | Non-Prob | Non-Prob (generations) | Non-Prob (population size) |
|---|---|---|---|
| $(1+x^3)^3+1$ | 68.09 | 64.11 | 58.81 |
| $(1+x+x^2)^3+4$ | 54.84 | 50.02 | 48.38 |
| $(1+x+x^4)^3+x$ | 66.67 | 62.29 | 64.69 |
| $(1+x^3)^3+x^2+3$ | 61.26 | 52.73 | 49.94 |
| $(1+x^2)^3+1$ | 59.59 | 52.4 | 62.75 |
| $(2+x^3)^3+1$ | 65.38 | 59.62 | 63.95 |
| $(1+x^3)^4+1$ | 63.88 | 58.15 | 64.5 |

Table 3.13: Average Non-Probabilistic Plushy Size out of 100 Independent Runs

| Objective function | Prob (5% UMAD, 95% BPM) | Prob (20% UMAD, 80% BPM) | Prob (30% UMAD, 70% BPM) | Prob (20% Crossover, 20% UMAD, 60% BPM) |
|---|---|---|---|---|
| $(1+x^3)^3+1$ | 94.75 | 102.54 | 103.5 | 96.93 |
| $(1+x+x^2)^3+4$ | 97.68 | 100.38 | 100.78 | 86.85 |
| $(1+x+x^4)^3+x$ | 93.78 | 92.21 | 94.41 | 87.22 |
| $(1+x^3)^3+x^2+3$ | 96.91 | 106.66 | 108.51 | 103.1 |
| $(1+x^2)^3+1$ | 94.14 | 104.56 | 103.84 | 89.56 |
| $(2+x^3)^3+1$ | 96.64 | 104.65 | 105.98 | 90.12 |
| $(1+x^3)^4+1$ | 95.23 | 98.97 | 94.29 | 92.4 |

Table 3.14: Average Probabilistic Plushy Size out of 100 Independent Runs

Table 3.16 shows the average proportion of deterministic probabilities of the Probabilistic Plushy genomes using the kitchen-sink instruction set. We observed that these percentages between the successful and unsuccessful runs were similar across all benchmark problems and probabilistic settings, except for the table entries with zero successful runs. We also noticed the averages of the percentage of deterministic probabilities of the successful runs were always less than the proportions of the unsuccessful runs.

| Objective function | Prob (5% UMAD, 95% BPM) | Prob (20% UMAD, 80% BPM) | Prob (30% UMAD, 70% BPM) | Prob (20% Crossover, 20% UMAD, 60% BPM) |
|---|---|---|---|---|
| $(1+x^3)^3+1$ | 0.36/0.43 | 0.39/0.42 | 0.34/0.38 | 0.32/0.35 |
| $(1+x+x^2)^3+4$ | 0.41/0.42 | 0.35/0.40 | 0.35/0.40 | 0.33/0.37 |
| $(1+x+x^4)^3+x$ | -/0.46 | -/0.42 | 0.36/0.40 | 0.35/0.39 |
| $(1+x^3)^3+x^2+3$ | -/0.42 | -/0.40 | -/0.40 | 0.25/0.36 |
| $(1+x^2)^3+1$ | 0.36/0.40 | 0.35/0.41 | 0.33/0.38 | 0.32/0.40 |
| $(2+x^3)^3+1$ | -/0.43 | 0.35/0.42 | 0.40/0.40 | 0.33/0.37 |
| $(1+x^3)^4+1$ | 0.37/0.47 | 0.35/0.43 | 0.34/0.41 | 0.33/0.35 |

Table 3.15: Average Proportion of Probabilities Equal to 0

| Objective function | Prob (5% UMAD, 95% BPM) | Prob (20% UMAD, 80% BPM) | Prob (30% UMAD, 70% BPM) | Prob (20% Crossover, 20% UMAD, 60% BPM) |
|---|---|---|---|---|
| $(1+x^3)^3+1$ | 0.73/0.78 | 0.73/0.78 | 0.70/0.77 | 0.72/0.77 |
| $(1+x+x^2)^3+4$ | 0.78/0.78 | 0.72/0.78 | 0.74/0.77 | 0.73/0.78 |
| $(1+x+x^4)^3+x$ | -/0.79 | -/0.78 | 0.74/0.77 | 0.71/0.78 |
| $(1+x^3)^3+x^2+3$ | -/0.79 | -/0.79 | -/0.78 | 0.68/0.77 |
| $(1+x^2)^3+1$ | 0.74/0.77 | 0.72/0.78 | 0.68/0.77 | 0.69/0.78 |
| $(2+x^3)^3+1$ | -/0.78 | 0.72/0.79 | 0.76/0.78 | 0.74/0.78 |
| $(1+x^3)^4+1$ | 0.76/0.79 | 0.72/0.79 | 0.71/0.79 | 0.72/0.77 |

Table 3.16: Average Proportion of Deterministic Probabilities (0 or 1)

Although statistical hypothesis tests may help us to interpret the significance of the result, their interpretation in the context of these experiments may not be trivial, and was beyond the scope of this project. We recommend future studies of Probabilistic Plushy genomes utilize significance tests to compare Propeller's performances on benchmark problems using Probabilistic Plushies and Non-Probabilistic Plushies.

# Chapter 4

# Limitations and Future Work

## 4.1 Crossover

Since our investigation primarily focused on creating child Probabilistic Plushy genomes using biased-perturbation mutation, we were limited in the number of runs we could perform using crossover. We found from our experiment involving different genetic sources that adding crossover to Propeller's set of genetic operators for Probabilistic Plushies resulted in the GP system having the best number of successes out of the other probabilistic run parameters. With Non-Probabilistic Plushies, crossover often does not contribute to creating lower error child genomes, as newly formed Plushies can consist of incompatible sequences of instructions due to being formed by the halves of their parent genomes. As such, we hypothesize that Probabilistic Plushy genomes can relax the effects of crossover by their ability to express into Non-Probabilistic Plushies using their probabilities to "eliminate" such sequences, allowing the genetic operator to be effective in the evolutionary process. Thus, one future work would be to experiment with different crossover/UMAD/biased-perturbation mutation ratios to gauge Propeller's performance with crossover.

## 4.2 Hyperparameter Tuning

One limitation of our investigation was the lack of hyperparameter tuning when running Propeller with Probabilistic Plushies. With the multiple evaluations method, we decided that the number of Non-Probabilistic Plushies a Probabilistic Plushy would express was ten because we wanted more than one expression to occur during the error evaluation process. In addition, we chose our percentage of biased-perturbation mutation to always be much higher than our percentage of UMAD for convenience, as we wanted Propeller to create child Probabilistic Plushies using our new mutation method for a majority of the evolutionary process. We specified a standard deviation of 0.2 our the amount of perturbation we perform on each of a Probabilistic Plushy genome's probabilities.

On a similar note, we specified that the maximum initial genome length for a Probabilistic Plushy was 100. Our results suggest that these genomes may need to begin with a bigger size to prevent these genomes from expressing extremely small sized Non-Probabilistic Plushies. We briefly explored alternative values for each of these hyperparameters but did not know if the hyperparameters we chose were close to optimal. As a result, the lack of hyperparameter tuning could have hindered Propeller's performance on the benchmark suite when using Probabilistic Plushies.

As such, one future work would be finding the optimal values for these parameters that help Propeller to increase its success rate on GP problems when using Probabilistic Plushy genomes as the candidate solutions. A possible experiment would be to create a set of probabilistic run parameters with different numbers for a specific parameter while keeping all other parameters fixed. We could then perform 100 independent Propeller runs on a benchmark suite using each probabilistic setting and analyze the GP system's number of successes on each benchmark problem.

## 4.3 Benchmark Problems

Another possible limitation was the benchmark problems we used in our experimentation, as our benchmark suite only consisted of floating-point symbolic regression problems. Although our experiments with these problems provided more information about Propeller's performance with Probabilistic Plushies, the lack of GP problems from other domains prevents us from further analyzing the solving capabilities of Probabilistic Plushies. Additionally, our limited set of problems hinders us in identifying ones that Propeller struggles to solve with Non-Probabilistic Plushies.

One future work would be experimenting with Probabilistic Plushies on problems that utilize real-world data, as these problems are challenging and important for many applications. It may be that using

Probabilistic Plushies could help Propeller solve such problems. A possible experimental setup could be creating new float regression problems using data sets from publicly available repositories. This extension could help continue our investigation of Probabilistic Plushies and their impact on Propeller's success rate on other GP problems.

# Chapter 5

# Conclusions

This thesis presents the Probabilistic Plushy, a probabilistic genome representation of programs that can express a random set of instructions to produce Non-Probabilistic Plushy genomes. Probabilistic Plushies allow for the implementation of genetic operators that make small changes to a Probabilistic Plushy's probabilities. As a result of these genetic operators, new Probabilistic Plushy can inherit the behaviors from their parent Probabilistic Plushy. From our preliminary experimentation, Probabilistic Plushy genomes can help improve Propeller's performance on some float regression problems and produce similar success rates to GP runs using Non-Probabilistic Plushies. When experimenting with different genetic sources, we found that Propeller has better success rates using Probabilistic Plushies than Non-Probabilistic Plushies when using crossover, UMAD, and biased-perturbation mutation. We also discuss the limitations of our experiments and suggest future work built on this project that can contribute to exploring more of the solving capabilities of probabilistic genomes.

# Chapter 6

# Appendix: Genetic Sources

## 6.1 Preliminary Experimentation

```
(def instructions
  (list :in1
        :float_add
        :float_subtract
        :float_mult
        :float_quot
        :float_eq
        :exec_dup
        :exec_if
        `close
        0.0
        1.0))
```

## 6.2 Genetic Sources

### 6.2.1 Hand-tuned

```
(def hand-tuned
  (list :in1
        :float_add
        :float_subtract
        :float_mult
        :float_quot
        :float_dup
        0.0
        1.0))
```

### 6.2.2 Kitchen-Sink

```
(def kitchen-sink
  (list 0.0
        1.0
        :print_newline
        :integer_subtract
        :integer_inc
        :boolean_stack_depth
        :vector_integer_eq
        :boolean_pop
        :string_from_char
        :vector_string_shove
        :vector_float_yank_dup
        :exec_yank_dup
        :vector_integer_shove
        :integer_yank_dup
        :string_flush
```

```
:boolean_swap
:exec_shove
:vector_boolean_yank
:exec_y
:boolean_yank
:integer_eq
:string_butlast
:string_conj_char
:vector_float_last
:string_substr
:integer_mult
:in1
:vector_string_dup_times
:vector_integer_dup
:boolean_or
:boolean_empty
:vector_string_print
:vector_boolean_swap
:char_dup_items
:vector_float_pushall
:char_is_whitespace
:vector_string_replacefirst
:string_first
:vector_boolean_first
:string_indexof_char
:vector_float_replace
:integer_from_string
:char_from_integer
```

```
:vector_integer_emptyvector
:vector_string_eq
:exec_dup_items
:vector_float_butlast
:boolean_dup_items
:exec_empty
:string_shove
:vector_boolean_pushall
:exec_rot
:vector_string_concat
:vector_float_indexof
:vector_string_subvec
:vector_integer_swap
:char_pop
:exec_dup
:vector_integer_butlast
:vector_float_rest
:vector_string_flush
:boolean_from_float
:float_sin
:boolean_flush
:char_is_digit
:float_lte
:vector_integer_empty
:code_print
:vector_string_stack_depth
:string_reverse
:exec_k
```

```
:vector_integer_yank
:float_from_integer
:char_rot
:vector_integer_dup_times
:char_print
:vector_integer_stack_depth
:vector_boolean_concat
:boolean_xor
:integer_gte
:vector_float_shove
:vector_integer_take
:boolean_dup_times
:string_replace_first
:vector_integer_yank_dup
:boolean_shove
:float_lt
:vector_string_dup
:vector_string_occurrencesof
:vector_integer_replace
:vector_float_reverse
:float_mod
:vector_float_subvec
:string_last
:boolean_print
:boolean_rot
:vector_string_rest
:integer_quot
:vector_float_remove
```

```
:integer_from_float
:integer_lte
:vector_integer_rot
:integer_mod
:string_concat
:vector_string_butlast
:vector_float_emptyvector
:vector_string_yank_dup
:integer_rot
:float_yank_dup
:vector_string_rot
:vector_string_take
:vector_float_dup_items
:integer_add
:vector_integer_occurrencesof
:integer_shove
:string_dup_times
:char_swap
:integer_max
:vector_integer_flush
:vector_integer_subvec
:vector_boolean_indexof
:vector_float_pop
:char_dup_times
:vector_string_remove
:vector_integer_contains
:code_append
:vector_float_eq
```

```
:vector_integer_conj
:string_eq
:integer_stack_depth
:float_max
:vector_boolean_set
:vector_float_conj
:float_dup_items
:string_take
:char_stack_depth
:vector_integer_replacefirst
:float_stack_depth
:integer_dup_times
:float_gt
:boolean_dup
:float_from_boolean
:vector_float_replacefirst
:vector_boolean_conj
:exec_dup_times
:vector_boolean_dup
:vector_integer_indexof
:vector_string_swap
:exec_eq
:string_empty_string
:string_swap
:integer_yank
:exec_while
:float_empty
:vector_boolean_print
```

```
:integer_min
:exec_swap
:vector_string_yank
:string_stack_depth
:string_replace_char
:char_all_from_string
:vector_integer_rest
:vector_boolean_length
:char_yank
:vector_float_empty
:string_pop
:float_eq
:integer_dup_items
:vector_boolean_empty
:vector_string_last
:string_nth
:vector_string_pop
:vector_integer_nth
:vector_integer_dup_items
:exec_if
:char_shove
:vector_boolean_remove
:vector_integer_remove
:boolean_invert_first_then_and
:string_print
:integer_from_boolean
:char_yank_dup
:vector_string_first
```

```
:boolean_from_integer
:string_set_char
:vector_integer_last
:char_is_letter
:vector_integer_concat
:integer_print
:boolean_eq
:float_gte
:string_occurencesof_char
:string_replace_first_char
:float_print
:integer_flush
:float_shove
:string_replace
:char_dup
:float_pop
:char_eq
:vector_float_nth
:vector_string_conj
:integer_gt
:vector_float_dup_times
:float_subtract
:vector_integer_length
:vector_float_set
:vector_string_indexof
:vector_boolean_rest
:vector_boolean_shove
:float_min
```

```
:boolean_not
:float_mult
:float_from_string
:vector_boolean_dup_items
:vector_integer_pop
:vector_boolean_last
:float_dec
:vector_float_contains
:string_empty
:char_empty
:exec_pop
:vector_integer_set
:vector_float_rot
:string_yank_dup
:string_remove_char
:vector_string_replace
:vector_float_first
:char_flush
:vector_float_occurrencesof
:vector_string_emptyvector
:float_add
:exec_s
:float_dup
:vector_string_nth
:vector_integer_reverse
:vector_integer_print
:char_from_float
:integer_lt
```

```
:vector_boolean_eq
:vector_boolean_dup_times
:string_contains_char
:string_yank
:vector_boolean_rot
:float_swap
:vector_string_pushall
:vector_string_set
:vector_boolean_flush
:vector_boolean_stack_depth
:vector_integer_pushall
:vector_boolean_reverse
:integer_swap
:string_split
:vector_boolean_contains
:string_from_boolean
:vector_float_dup
:vector_boolean_replace
:vector_string_dup_items
:integer_dup
:vector_boolean_nth
:vector_string_length
:string_rest
:float_tan
:string_rot
:exec_yank
:string_parse_to_chars
:integer_pop
```

```
:integer_empty
:vector_float_flush
:vector_float_yank
:exec_print
:float_dup_times
:float_inc
:vector_float_length
:integer_dec
:string_contains
:vector_float_concat
:vector_float_stack_depth
:vector_integer_first
:vector_float_print
:float_rot
:vector_string_contains
:vector_boolean_occurrencesof
:string_dup_items
:vector_string_reverse
:exec_stack_depth
:float_flush
:boolean_and
:vector_boolean_butlast
:string_length
:float_cos
:string_from_integer
:exec_flush
:vector_string_empty
:exec_when
```

```
:vector_float_swap
:vector_boolean_pop
:float_quot
:vector_boolean_take
:vector_float_take
:boolean_invert_second_then_and
:vector_boolean_subvec
:float_yank
:vector_boolean_emptyvector
:vector_boolean_replacefirst
:string_from_float
:vector_boolean_yank_dup
:string_dup
:boolean_yank_dup))
```

# Bibliography

[1] William La Cava, Lee Spector, and Kourosh Danai. Epsilon-lexicase selection for regression. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. ACM, jul 2016.

[2] Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. Program synthesis using uniform mutation by addition and deletion. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, page 1127–1134, New York, NY, USA, 2018. Association for Computing Machinery.

[3] Thomas Helmuth, Edward Pantridge, Grace Woolson, and Lee Spector. Genetic Source Sensitivity and Transfer Learning in Genetic Programming. volume ALIFE 2020: The 2020 Conference on Artificial Life of *ALIFE 2022: The 2022 Conference on Artificial Life*, pages 303–311, 07 2020.

[4] William B. Langdon, Riccardo Poli, Nicholas F. McPhee, and John R. Koza. *Genetic Programming: An Introduction and Tutorial, with a Survey of Techniques and Applications*, pages 927–1028. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[5] Jessica Megane, Nuno Lourenco, and Penousal Machado. Probabilistic grammatical evolution. In Ting Hu, Nuno Lourenco, and Eric Medvet, editors, *EuroGP 2021: Proceedings of the 24th European Conference on Genetic Programming*, volume 12691 of *LNCS*, pages 198–213, Virtual Event, 7-9 April 2021. Springer Verlag.

[6] Edward Pantridge, Thomas Helmuth, and Lee Spector. *Comparison of Linear Genome Representations for Software Synthesis*, pages 255–274. Springer International Publishing, Cham, 2020.

[7] R. P. Salustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):123–141, 1997.