Extensions to Behavioral Genetic Programming

by

Steven B. Fine

S.B., Massachusetts Institute of Technology (2016)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2017

Author .....
Department of Electrical Engineering and Computer Science
February 3, 2017

Certified by .....
Una-May O'Reilly
Principal Research Scientist, MIT CSAIL
Thesis Supervisor

Accepted by .....
Christopher J. Terman
Chairman, Master of Engineering Thesis Committee

Extensions to Behavioral Genetic Programming

by

Steven B. Fine

Submitted to the Department of Electrical Engineering and Computer Science
on February 3, 2017, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this work I introduce genetic programming [5] as a general technique to produce programs with arbitrary behavior. I discuss genetic programming and its application the task of symbolic regression. I introduce behavioral genetic programming [6] as an extension to genetic programming and explore various extensions to it. The codebase that I build is made sufficiently flexible to easily accommodate future adaptions to the behavioral genetic programming methodology. I test the performance of the implementation of behavioral genetic programming along with several extensions.

Thesis Supervisor: Una-May O'Reilly
Title: Principal Research Scientist, MIT CSAIL

Acknowledgments

I would like to thank Una-May O'Reilly for all of her guidance throughout this research project. Not only did she help me find a project that interested me, but our discussions were instrumental for the progression of my research. I would also like to thank Krzysztof Krawiec for our discussion on possible extensions to behavioral genetic programming, and for answering my questions about the original implementation. Finally, I would like to thank Erik Hemberg for all of the help he provided me in getting my experiments up and running on the CSAIL cloud, and for our discussions about my experimental results.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Genetic programming (GP) [5] is a subfield of Artificial Intelligence, in which the principles of evolution are algorithmically translated in order to produce programs with desired functionality. Each such program is defined by a set of genes, which can be mutated and swapped in a manner similar to reproduction in biological organisms. A typical genetic programming algorithm will start with a population of initial programs, which are randomly generated. Each successive iteration, a population of new programs is generated from the old population. Finally, the best programs from the two populations are selected, and the process is repeated until a sufficiently good program is found.

The power of this technique is that the programmer does not need to guess at the structure of the desired program. All the programmer needs is a means of constructing new programs from old programs, and a fitness function by which to compare the performance of one program to another. On the surface, this seems like a very promising method to produce arbitrarily complex programs, as long as the programmer has an understanding of the desired output. However, in practice, the landscape of programs is enormous, and often times there are many local optima, which prevent evolving programs from achieving the desired functionality.

There are a variety of techniques and adaptations to the basic genetic programming methodology that attempt to exploit features of the evolutionary process to arrive at optimal solutions more quickly. One such technique, designed by Krawiec

et al. [6] attempts to utilize information about how to identify useful subprograms. These subprograms are components of programs in the population that will help drive the evolutionary process, even if they are a part of a program that may not perform well on the specified task. This technique is termed *behavioral genetic programming* (BGP). The vision of this work is that BGP is a paradigm rich with possible extensions to explore, many of which could give deeper insight into genetic programming methods.

This work focuses on replicating the results first achieved by Krawiec et al., and exploring various extensions to the BGP paradigm. This work proceeds as follows. Chapter 2 discusses the basics of genetic programming, and the key ideas behind BGP. Chapter 3 discusses my implementation of BGP with the added ability of running many different configurations that were not explored by Krawiec et al. Chapter 4 discusses the experiments that are performed, and Chapter 5 discusses my contributions and possible paths for future work.

Chapter 2

Related Work

2.1 Basics of Genetic Programming

2.1.1 Program Representations

As is stated in Chapter 1, Genetic programming is a general technique that can be applied to almost any task domain. However, this work predominantly focuses on the task of symbolic regression. Symbolic regression aims to find a mathematical expression that fits a given data set's training and out of sample test examples. Therefore the programs in which we are interested define mathematical expressions by combining a set of primitive operations.

Much of the time in genetic programming random segments of code are being removed, inserted, and swapped between different programs. Therefore, it is important to have a program representation that is resilient to these types of operations. As a result, it can be useful to avoid loops, and other types of statements that could result in programs that do not terminate. For this reason, often a tree structure is employed, where the internal nodes of the trees are functions, with the children as the arguments, and the leaves of the trees are terminals, such as variables and constants.

For the task of symbolic regression, the functions can be any set of primitive mathematical functions, while the terminals will typically be the features of the data set.

The tree in Figure 2-1 represents a simple program, which computes the mathematical function $f(x) = \ln(x1) + (x1 - x2)$. Given a larger set of mathematical operations, it is possible to create arbitrarily complex mathematical expressions.

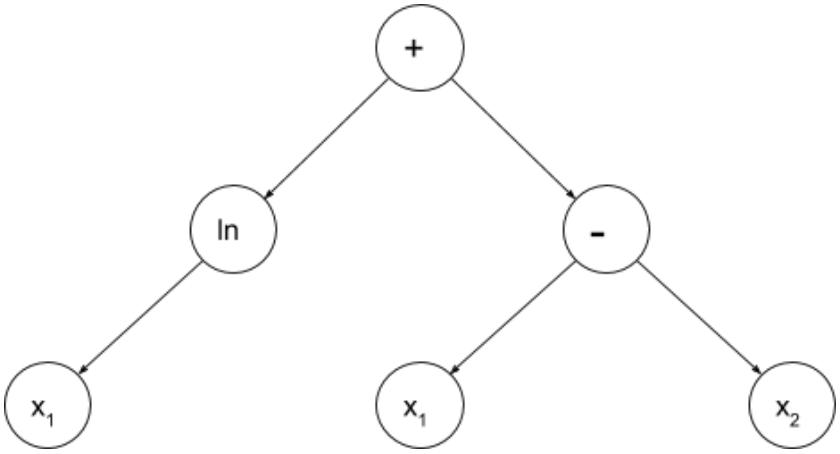

Figure 2-1: Sample tree representing a program.

2.1.2 Generating New Programs

One of the core aspects of genetic programming is how to generate a new population of programs from an old population. There are two primary operations used to generate new programs: the mutation operation and the crossover operation.

Mutation

The mutation operation generates a single new program from a single old program. First a mutation point is selected in the old program. This can be selected uniformly at random, or biased towards certain parts of the tree. Then the subtree located at the mutation point is removed, and a new subtree is generated in a similar way to how the first population of programs was generated.

Crossover

The crossover operation generates two new programs from two old programs. First a crossover point is selected in each old program. This can be done in any of the same

ways that a mutation point is selected. Then the subtrees located at the two crossover points in the two old programs are swapped, creating two new program trees.

2.1.3 Choosing Programs to Survive

After a new population of programs is generated, only half of the programs from the combined new and old populations are kept for the next generation of the evolutionary process. If there is a single fitness function that is being optimized, this process is very straightforward. Simply keep the programs that perform best on the fitness function. However, often there are multiple fitness functions that are used to drive the evolutionary process. For example, in the case of symbolic regression, one fitness function could be the absolute error for predicting the dependent variable, while the other fitness function could be the program size. When there are multiple fitness functions, often one program will have a better fitness score than another program for one fitness function, but not the remaining fitness functions. In this case a different method must be used to determine which programs should be kept. The method used for the purpose of this work is discussed in section 3.2.4.

2.1.4 Termination

The run of a genetic program terminates after one of several conditions is met. The following are common termination conditions:

1. An optimal program is found.
2. A maximum number of generations is reached.
3. The run has exceeded the maximum allocated amount of time.

2.2 Behavioral Genetic Programming

One extension to the genetic programming paradigm is behavioral genetic programming (BGP) [6]. As was mentioned in Chapter 1, BGP attempts to identify useful subprograms that can then be used to enhance the evolutionary process.

2.2.1 Trace

In most genetic programming algorithms, the search is driven by the output of the fitness functions alone. Only programs that perform well on the fitness functions are kept, while the rest are rejected. However, every subtree that makes up an individual program has a distinct numerical output for each fitness case in any given data set.

Let us consider the program in Figure 2-1, and the sample data set in Table 2.1. Conventional genetic programming will only consider the outputs for each data point: $\ln(3) + 1$, $\ln(5) + 2$, and compare those values to the desired output. However, each subtree has its own output, which is ignored by the genetic programming process.

x_1	x_2	y
3	2	3
5	3	4

Table 2.1: Sample data set.

The collection of the outputs on each subtree for all of the data points is called the trace. For the program in Figure 2-1 and the fitness cases in Table 2.1, the trace is shown in Table 2.2.

s_1	s_2	s_3	s_4	s_5	s_6
3	ln(3)	3	2	1	ln(3) + 1
5	ln(5)	5	3	2	ln(5) + 2

Table 2.2: The trace of the program from Figure 2-1 for the data set in Table 2.1.

The trace is a matrix where the number of rows is equal to the number of data points, and the number of columns is equal to the number of subtrees in a given program. There is no set order of the columns, although here they are presented in the depth first traversal order of the subtrees that produce the values. The trace captures a full snapshot of all of the intermediate states of the program evaluation.

2.2.2 Model

The key idea behind Behavioral Programming is to exploit the trace to identify useful subtrees. Ordinary crossover does not incorporate any information about the quality of the subtrees that are being swapped. However, the trace opens up many possibilities to explore the quality of subtrees. In Behavioral Programming, the trace is used to train a machine learning model to predict the desired output for each fitness case. The model is then used for two purposes. The first is to create additional fitness measures by which to evaluate a given program. The second is to identify useful subtrees based on the composition of this model, which are then placed in an archive. The content of the archive is then used for the supply the subtrees that are used in the crossover operation, instead of taking arbitrary subtrees from other programs in the population.

Chapter 3

Implementation

The following section details my implementation of behavioral genetic programming. It follows the details presented by Krawiec et al. [6], with several adaptions in order to explore several variants of BGP.

3.1 Codebase

The codebase that I use for this project was first built for FlexGP [9] and then extended for the implementation of Multiple Regression Genetic Programming (MRGP) [1]. It is written in the Java programming language. With simple modifications the codebase could run conventional GP for symbolic regression tasks. One of the main contributions of my work is the way in which I extended the codebase. Many of the abstractions that I introduced allow the GP process to be extended in unforeseen ways.

3.2 Genetic Programming Run

During the execution of a genetic programming algorithm, there are three steps associated with any given generation. The steps are as follows:

- **Initialization/Reproduction**: Generate a new population of programs. In the first round of evolution, the programs are generated from scratch. In all

subsequent rounds the programs are generated from the programs in the old population.

- **Evaluation**: Evaluate all of the programs in the new population.
- **Survival**: Select which programs in the combined population will be kept for the next round of evolutionary computation.

3.2.1 Initialization

When the GP run is initialized, the first step is to create an initial population of programs. The number of programs in each generation is specified by the user at the start of the run. The programs are generated by randomly choosing functions and terminals as children of the parent nodes, until every leaf in the tree is a terminal. A maximum depth is specified, and the trees are generated such that the trees vary from having a depth of one to the maximum depth.

3.2.2 Reproduction

In all subsequent generations, new programs are generated from the population of programs belonging to the previous generation. One of the modifications that I made was the introduction of a reproduction operator. Previously the mutation operator and crossover operator were hardcoded as the only possible operators used to generate new children. The reproduction operator introduces a method to add children to the new population, which can have many different implementations. For the purposes of this work, the implementation of the reproduction operator that is used allows mutation, crossover, and archive-based crossover, each with an associated probability of being used, each time new programs are to be generated. This implementation enables the use of conventional GP by setting the archive-based crossover probability to zero.

Selection of Programs for Reproduction

During reproduction, new programs are generated until the population size is reached. To generate a new program, an old program (or in case of crossover, a pair of old programs) is selected to have one of the reproduction operators applied. The process by which a program is selected is called *tournament selection*. A tournament size is specified in the parameters of the GP run. For a given tournament size n, n individuals are drawn uniformly at random from the population, and then compete to determine which individual will have the reproduction operator applied. The winner of the competition is simply the program with the highest rank based on the NSGA-II algorithm [2]. The NSGA-II algorithm is discussed in Section 3.2.4.

Reproduction Operators

The mutation and crossover operations were discussed in Section 2.1.2. In the implementation of behavioral genetic programming by Krawiec et al., the mutation and crossover points for a given program tree are selected by first choosing the depth of the point uniformly at random from 1 to the depth of the tree, then choosing a node uniformly at random from the given depth. The third operation, archive-based crossover, works identically to the mutation operator, with the exception of how the new subtrees are generated. In ordinary mutation, the subtrees are generated in the same manner that each member of the initial population of programs is generated. However, in archive-based crossover, the subtrees are drawn from an archive, which maintains a weighted distribution over which subtrees will be selected. The archive is discussed further in section 3.2.3.

3.2.3 Evaluation

Once a new population of programs is generated, all of the new programs must be evaluated on all of the fitness functions that are being used for the genetic programming run. Additionally, in the case of behavioral genetic programming, the archive is populated based on a machine learning model that is generated from the trace of each

program. This section details the process by which these steps are accomplished.

Fitness Function Evaluation

Another abstraction that I introduced into the codebase is that of a fitness function evaluator. In conventional genetic programming, the program fitness functions can be evaluated in any order, and all that is needed is the numerical output for each fitness function for each program. In behavioral genetic programming there are two classes of fitness functions. The functions in the first class only require the programs themselves to be evaluated. The fitness functions in conventional genetic programming belong to this class. The functions in the second class require a machine learning model built on the trace of each program to be evaluated. For each possible configuration of BGP, a different fitness function evaluator is used, which takes care of the proper order for the fitness functions to be evaluated, and the generation of the model. The different configurations are discussed in Section 4.1 and detailed in Appendix D.

Conventional Genetic Programming Fitness Functions

In conventional genetic programming, there are typically two fitness functions used to evaluate a program. The first is the program error f, which is a measure of how close the program output on each data point is to the desired output. The form of the program error fitness used by Krawiec et al. is given by Equation 3.1, where $\hat{y}$ is the output of the program, y is the desired output, and d_m denotes the Manhattan distance between the two arguments. The second fitness function is typically a measure of program size s, where smaller programs are considered more fit than larger programs. The form of the program size fitness used by Krawiec et al. is given by Equation 3.2, where $|p|$ is the number of nodes in the tree that defines the program.

$$f = 1 - \frac{1}{1 + d_m(\hat{y}, y)} \tag{3.1}$$

$$s = 1 - \frac{1}{|p|} \tag{3.2}$$

Behavioral Genetic Programming Fitness Functions

In behavioral genetic programming there are four fitness functions. Two are the conventional fitness functions discussed in Section 3.2.3. The remaining fitness functions that are used are the model complexity c, given by Equation 3.3, and the model error e, given by Equation 3.4, where M is the output of the machine learning model when it is evaluated on the trace of the program, and $|M|$ is the size of the model.

One of the most expensive operations in the execution of a genetic program is the evaluation of each program on all of the data points. This step is required for both calculating the program error fitness, and generating the trace of the program. Therefore, in behavioral genetic programming, the trace for a given program is generated while the program error fitness value is calculated.

$$c = 1 - \frac{1}{|M|} \tag{3.3}$$

$$e = 1 - \frac{1}{1 + d_m(M, y)} \tag{3.4}$$

Model

Once the trace for a given program in the population has been collected the machine learning model can be built. The purpose of the model is two-fold. The first is that it introduces additional fitness measures for each program (given by Equations 3.3 and 3.4). Second, it guides the process of populating the archive.

The algorithm that was used for the model in the original implementation of behavioral genetic programming, by Krawiec et al. was REPTree. [4] REPTree is a decision tree that can be used for both classification and regression tasks. For each program in the population, a different model is built on the program trace. In my implementation, one can use different models by writing different implementations for the model interface.

Archive

In behavioral genetic programming, an archive is used from which to draw subtrees that are used for archive-based crossover. The archive is given a maximum capacity, which in the original implementation of Behavioral Programming by Krawiec et al. was set to 50. After each round of fitness function evaluations, the archive is repopulated in the following way. First the candidate subtrees are selected based on whether or not their column in the trace was used in the machine learning model. Each subtree is assigned a weight given by Equation 3.5, where e is the model error from Equation 3.4 and $|U|$ is the number of distinct columns of the trace used in the model. Note that all subtrees that are used in a certain model are given the same weight in the archive. Each generation, after the model has been generated, the candidate subtrees are combined with the pre existing contents of the archive (with the exception of the first generation, as the archive would be empty). If the number of candidate subtrees combined with the previous contents of the archive is less than the archive capacity, then all of the subtrees are added to the archive. If the number of candidate subtrees combined with the previous contents of the archive is greater than the capacity of the archive, then subtrees are drawn without replacement with probability proportional to their assigned weights. Algorithm 1 illustrates this procedure. For my implementation I use a modification [8] of the algorithm introduced by Efraimidis and Spirakis [3] to efficiently draw from a weighted distribution without replacement (Note that Algorithm 1 does not include the details of this procedure). When subtrees are drawn from the archive for archive-based crossover, they are drawn from the weighted distribution with replacement.

$$w = \frac{1}{(1+e)|U|} \tag{3.5}$$

One significant implementation detail is that the archive cannot have two subtrees which both have the same output on all of the data points (termed the same *semantics*). In both the original implementation and my own, if two subtrees have the same semantics, then only the subtree with fewer nodes is kept. Subtrees are duplicated

Algorithm 1 Populate Archive

```
1: procedure PopulateArchive
2:     for subtree in archive do
3:         w ← the weight of subtree
4:         add (subtree,w) to candidateSubtrees
5:     clear archive
6:     for program in population do
7:         T ← collectTrace(program)
8:         M ← buildModel(T)
9:         U ← subtrees included in M
10:        w ← 1/((1 + e)|U|)
11:        for subtree in U do
12:            add (subtree,w) to candidateSubtrees
13:    if |candidateSubtrees| ≤ ARCHIVE_CAPACITY then
14:        add all candidateSubtrees to archive
15:    else
16:        while archive.size < ARCHIVE_CAPACITY do
17:            subtree ← draw from candidateSubtrees
18:            w ← the weight of subtree
19:            remove (subtree,w) from candidateSubtrees
20:            add (subtree,w) to archive
```

both when they are added to the archive, and drawn from the archive, to ensure that no two programs have a reference to the same subtree.

3.2.4 Survival

Once all of the fitness functions have been evaluated, and the archive repopulated, only half of the programs from the combination of the old and new generation may be kept. Given that in conventional genetic programming and behavioral genetic programming there are multiple fitness functions, we cannot simply keep the programs that perform best on the fitness functions. Many programs may perform well on one metric, at the expense of another. Therefore the NSGA-II algorithm [2] is used to rank all of the programs in the combined population, and only the highest ranking half of the combined population is kept for the next generation.

The NSGA-II algorithm first classifies each program into a series of fronts. The first front (termed *pareto front*) is defined by all of the programs for which no other

program in the population performs better on at least one fitness function, and as well or better on the remaining fitness functions. Each successive front is defined by first removing the last front from the population, and recomputing the next front on the remaining programs in the same way that the pareto front was computed. Every program in a given front outranks all programs in all subsequent fronts, and is outranked by all programs in all previous fronts.

Within a given front each program is ranked by its crowding distance. To compute the crowding distance of an individual program, first all fitness measures are normalized (Note that in BGP we use fitness functions which are already normalized). Then for each fitness measure, the difference between the fitness of the program with the next highest fitness and the next lowest fitness is computed. The crowding distance for an individual program is given by the average difference across all fitness measures. If for a given fitness measure an individual program has either the highest or lowest fitness value in the population, it receives a crowding distance of infinity. Programs with higher crowding distances outrank programs with lower crowding distances. The reason for the crowding distance ranking is to increase the diversity of the population. Programs with higher crowding distances are more dissimilar to other programs in the population.

3.3 Extensions to Behavioral Genetic Programming

One of the core goals of this work is to explore different extensions and alternative implementations of behavioral genetic programming. For the purpose of this work I explore several variants of the BGP machine learning model, which are detailed below.

3.3.1 Full Population Model

One of the main ideas that I explore is how the model would perform if instead of training one model on the trace of each program in the population, I train one model for the entire population on the combined traces of all of the programs. This

effectively creates a single trace matrix with the same number of rows but many more columns. The combined trace matrix represents an extremely high dimensional problem, with uninvestigated correlation of its features.

The idea behind training a model for each individual in the population is twofold. The first is that the accuracy of the model gives insight into how much information the subtrees in a program encode about the desired output. The second is that knowing which subtrees are used in the model gives insight into which subtrees are more valuable than others. What it does not tell us is which subtrees will work well with other subtrees in the population.

The motivation behind combining all of the program traces to train a single machine learning model is to hopefully identify groups of subtrees coming from different programs in the population that work well together. With this method one can populate the archive in the same way as in ordinary BGP, however, one cannot use the additional fitness measures, because each program in the population does not have a distinct model error and model complexity associated with it.

3.3.2 Lasso Model

Another implementation that I explore uses Least Absolute Shrinkage and Selection Operator (Lasso) as the machine learning model. Instead of training REPTree on the trace of each program in the population, I run the Lasso regression method. For each feature in the data set on which Lasso is run, it assigns a real valued weight. The predicted output is given by the inner product of the features in the data set, and the weights assigned by Lasso, plus a real valued offset. One benefit to Lasso is that it will perform feature selection by assigning some feature weights a value of 0. In this model, I use the same formula for model error, and I define the size of the model $|M|$ as the number of features with non-zero coefficients. Only subtrees with corresponding features that are given non-zero weights by Lasso are included in the archive. I use the absolute value of the weights assigned by Lasso as the weight for each respective subtree in the archive.

3.3.3 Scikit Learn Model

Another implementation of the model interface that I employ uses the Python Scikit Learn DecisionTreeRegressor class for regression. This model works identically to the model that uses REPTree with the exception of calling an alternative implementation of a decision tree.

3.3.4 Randomized Model

Finally, I use an implementation that is identical to the model used in REPTree, however, for each subtree used in the resulting REPTree decision tree, before being placed in the archive, it is replaced by a subtree drawn uniformly at random from the subtrees in the program. The purpose of this implementation is not to see whether or not this model performs better than the model that uses the subtrees from REPTree. Rather, it is used to illuminate how much is gained from populating the archive with the trees that are used by REPTree.

Chapter 4

Experiments

4.1 Setup

In order to test the performance of different behavioral genetic programming models, I run 16 distinct configurations of BGP on the same 17 data sets that are used for the task of symbolic regression by Krawiec et al. The basis of the data sets is taken from a paper entitled *Genetic Programming Needs Better Benchmarks* by McDermott et al. [7] The complete specification of all 17 data sets is presented in Appendix A.

The basis of the 16 BGP configurations that I use, comes from the 3 BGP configurations used by Krawiec et al.:

1. **BP2A** uses only the program error fitness function and the program size fitness function (Equations 3.1 and 3.2), but replaces crossover with archive-based crossover.
2. **BP4** uses all four BGP fitness functions (Equations 3.1, 3.2, 3.3, and 3.4), but performs ordinary crossover.
3. **BP4A** uses all four BGP fitness functions, and replaces crossover with archive-based crossover.

Note that in all three variants, the same model is constructed. The only distinction is for what the model is used. In addition to running the three configurations above using the REPTree model (as is done by Krawiec et al.), I run each of the three

configurations using each of the models specified in Section 3.3. I also explore several other parameter configurations. I explore the use of a larger archive, and the use of different mutation and archive-based crossover rates. Specifically, in several runs I use a mutation rate of 0.05, and a archive-based crossover rate of 0.95. The reason for this last variant is that GP crossover creates two new programs each time it is called. However, archive-based crossover only creates one new program. These altered operation rates closely simulate the creation of two new programs each time archive-based crossover is called.

For a baseline comparison, I run conventional GP with the two conventional GP fitness functions described in Section 3.2.3. For all of the runs, I use the same parameters that are specified by Krawiec et al. A complete list of the parameters that are fixed for every run can be found in Appendix C. The following is a complete list of the 17 configurations (16 BGP, 1 conventional GP) that I run. Their exact specifications are detailed in Appendix D.

1. GP
2. BP2A - REPTree
3. BP2A - Full Pop
4. BP2A - Lasso
5. BP2A - Scikit Learn
6. BP2A - Randomized
7. BP2A - Larger Archive
8. BP2A - Different Rates
9. BP4 - REPTree
10. BP4 - Lasso
11. BP4 - Scikit Learn
12. BP4A - REPTree
13. BP4A - Lasso
14. BP4A - Scikit Learn
15. BP4A - Randomized
16. BP4A - Larger Archive
17. BP4A - Different Rates

For the majority of the configurations, I perform 30 runs on each data set. However, all of the Scikit Learn configurations take much longer to run because they require calling Python from within Java. Therefore, they are run 17 times each. Ad-

ditionally, BP2A - Lasso, and BP4A - Lasso, both did not run to completion. As a result, they are omitted from the discussion. They are addressed in Section 5.2.

4.2 Results

Appendix B contains tables for the average and standard deviation of program fitness, program size, and program runtime, for the best of run programs for each configuration and data set. A best of run program is the program with the lowest program error generated in a given run. Additionally, there is a table with the percentage of runs that produced a perfect individual for each configuration and dataset. Below I discuss the results in aggregate and compare them to those of Krawiec et al.

Table 4.1 contains the average rank for each configuration across all data sets for a variety of metrics.

Average Rank for BGP with REPTree

Krawiec et al found that the three BGP configurations that they run rank as follows for the given metrics:

- **Program fitness**: BP4A, BP2A, BP4, GP
- **Number of perfect programs found**: BP4A, BP2A, BP4, GP
- **Program size**: GP, BP4, BP2A, BP4A
- **Program runtime**: GP, BP4, BP4A, BP2A

It is important to note that the above ranks by Krawiec et al. are calculated based on the performance of each configuration across multiple task domains: boolean, categorical, and regression, while I only implement BGP for regression. For program runtime, both Krawiec et al., and I find that GP is by far the fastest, and using an archive is slower than not. Considering program fitness, and the number of perfect programs found, both Krawiec et al., and I find that the BGP paradigm of replacing crossover with archive-based crossover is beneficial. However, my results suggest that

when considering program fitness, the added fitness functions hurt the evolutionary process.

	Average Fitness Rank		Average Rank For Finding Perfect Programs	
1	BP2A - Larger Archive	3.59	BP2A - Full Pop	1.12
2	BP2A - REPTree	3.76	BP4A - Larger Archive	1.71
3	BP2A - Different Rates	3.82	BP4A - Different Rates	2.24
4	BP2A - Scikit Learn	6.18	BP4A - Scikit Learn	2.47
5	BP4A - Scikit Learn	6.29	BP2A - Randomized	2.65
6	BP4 - Scikit Learn	6.53	BP4A - REPTree	2.65
7	BP2A - Randomized	6.65	BP2A - Larger Archive	2.76
8	GP	7.0	BP4A - Randomized	3.18
9	BP4A - Different Rates	9.82	BP4 - REPTree	3.24
10	BP4 - Lasso	9.88	BP2A - Different Rates	3.53
11	BP4A - REPTree	10.12	BP4 - Lasso	3.53
12	BP4A - Larger Archive	10.94	BP2A - REPTree	3.59
13	BP2A - Full Pop	11.35	BP2A - Scikit Learn	3.59
14	BP4 - REPTree	11.76	GP	4.59
15	BP4A - Randomized	12.29	BP4 - Scikit Learn	5.59
	Average Size Rank		Average Runtime Rank	
1	BP4 - Scikit Learn	2.82	GP	1.0
2	GP	3.0	BP2A - Larger Archive	2.24
3	BP2A - Randomized	5.82	BP4 - REPTree	4.76
4	BP4 - Lasso	6.29	BP4A - Different Rates	5.0
5	BP2A - Scikit Learn	6.59	BP2A - Randomized	5.71
6	BP4 - REPTree	7.18	BP4A - Randomized	6.18
7	BP4A - Scikit Learn	7.41	BP2A - REPTree	6.47
8	BP2A - Different Rates	7.71	BP2A - Different Rates	6.47
9	BP2A - Larger Archive	8.29	BP4A - Larger Archive	7.35
10	BP2A - REPTree	9.35	BP4A - REPTree	9.88
11	BP4A - Larger Archive	9.53	BP2A - Full Pop	10.94
12	BP4A - Randomized	9.88	BP4 - Lasso	12.35
13	BP4A - Different Rates	10.35	BP4 - Scikit Learn	12.94
14	BP4A - REPTree	10.88	BP4A - Scikit Learn	14.06
15	BP2A - Full Pop	14.82	BP2A - Scikit Learn	14.65

Table 4.1: Average rank of each configuration across all data sets.

Full Population Model

It seems that running the model on each program trace is almost always better than running the model on the combined trace of the entire population. In the full population case, each generation, the archive is only populated with subtrees taken from a single model, which might result in a less diverse archive. Further work is needed

to understand precisely why this model performs less well.

Lasso Model

The most significant feature of BGP that this configuration brings to light is that it is important to use a highly robust machine learning model for BGP. Even for BP4 - Lasso, which did run to completion, the standard deviation of the runtimes are by far the largest (see Table B.6). The results also suggest, that for BP4, Lasso may provide better additional fitness measures than REPTree.

Scikit Learn Model

For the Scikit Learn model, it seems that the resulting fitness is neither conclusively better nor worse than using REPTree. However, the runtimes of all of the Scikit Learn Model configurations were notably the longest. This is primarily due to the overhead of running a model that is written in the Python programming language, and calling it from Java.

Randomized Model

The randomized model consistently performed worse than the model that used the trees generated by REPTree. This gives confidence to the claim that the subtrees generated by REPTree are more beneficial for driving the evolutionary process.

Larger Archive

The results suggest that using a larger archive does not substantially help nor harm the resulting fitness of the generated programs.

Different Rates

It appears that for both relevant BGP configurations, using a higher archive-based crossover rate, does not have a substantial effect on the resulting program fitness for a given run. This seems to invalidate the possibility that BGP only performed better

than GP because the effective reproduction operator probabilities were substantially different from conventional GP.

Chapter 5

Conclusion

5.1 Contributions

My primary contributions are threefold.

1. Provide support for the claims of BGP by Krawiec et al.
2. Create a BGP implementation that is easily extendable for future work related to BGP.
3. Explore numerous extensions to and features of the BGP methodology.

5.2 Future Work

A lot of what this work brings to light is particular paths to extend the concepts and understanding of BGP. Below I detail several avenues to explore.

Alternate Models

The primary avenue that this work opens up is the possibility of exploring different models to use in BGP. It would be interesting to see if using a machine learning model whose purpose is more inline with what BGP asks for would benefit the evolutionary process. For example, instead of building an entire machine learning model on the trace, one could use a feature selection technique, or measure the statistical correlation

between the columns. The output would provide material with which to populate the archive. However, this would not provide additional fitness measures.

Lasso

Lasso brought several interesting features of BGP to light. It is important to have a machine learning model that is robust against the pathological inputs that can be generated by a genetic programming algorithm. It would be interesting to explore why Lasso in general, or at least the implementation that I use has significant trouble for certain inputs. Additionally, it is interesting that even though running Lasso is the runtime bottle neck, the configurations that took substantially too long to run were BP2A, and BP4A, both of which use an archive. Understanding precisely why the configurations with an archive produce worse inputs to Lasso could be insightful.

Mixing Traces

It is unclear exactly why the BGP model that uses the combined traces of all of the programs in the population performed less well than running the model on each program trace independently. It is possible that the idea has merit, but the particulars were not a good fit for BGP. In particular, in each generation only a single machine learning model is built. Therefore, all of the selected trees put into the archive in a single generation have the same weight.

An alternate implementation would be to draw random subsets from the combined trace of the programs in the population, and build a model on each. This would create many candidate subtrees with different weights, and possibly a more robust archive, if it can be populated with subtrees that are frequently selected by the machine learning model.

Subtrees with the Same Semantics

As is mentioned in Section 3.2.3 if two subtrees have identical columns in the program trace (i.e. identical *semantics*), only the smaller subtree is kept. This introduces a bias that is not necessarily beneficial to the evolutionary process. It would be interesting

to explore how common subtrees with identical semantics are, and if choosing the smaller tree is the better choice.

Combining Reproduction Operators

It would be interesting to see if combining mutation, crossover, and archive-based crossover could enable better performance than using only two of the reproduction operators.

Model Evaluation

In BGP, after the model is built on the trace of each program, it is evaluated on the trace. The result of its evaluation is then used as the output of one of the fitness functions for the program. It would be interesting to explore evaluating each model on a test set, instead of the trace. Perhaps this would yield a more useful fitness function.

Statistical Significance of the Results

Finally, in order to better understand the significance of the results presented in this work, it would be useful to determine for which configurations the difference in performance is statistically significant.

Appendix A

Data Sets

All of the data sets used are defined such that the dependent variable is the output of a particular mathematical function for a given set of inputs. They are taken from a paper entitled *Genetic Programming Needs Better Benchmarks* by McDermott et al. [7] All of the inputs are taken to form a grid on some interval. Let $E[a, b, c]$ denote c samples equally spaced in the interval $[a, b]$. (Note that McDermott et al. defines $E[a, b, c]$ slightly differently.) Below is a list of all of the data sets that are used:

1. **Keijzer1**: $0.3x \sin(2\pi x)$; $x \in E[-1, 1, 20]$
2. **Keijzer11**: $xy + \sin((x-1)(y-1))$; $x, y \in E[-3, 3, 5]$
3. **Keijzer12**: $x^4 - x^3 + \frac{y^2}{2} - y$; $x, y \in E[-3, 3, 5]$
4. **Keijzer13**: $6 \sin(x) \cos(y)$; $x, y \in E[-3, 3, 5]$
5. **Keijzer14**: $\frac{8}{2+x^2+y^2}$; $x, y \in E[-3, 3, 5]$
6. **Keijzer15**: $\frac{x^3}{5} - \frac{y^3}{2} - y - x$; $x, y \in E[-3, 3, 5]$
7. **Keijzer4**: $x^3 e^{-x} \cos(x) \sin(x)(\sin^2(x) \cos(x) - 1)$; $x \in E[0, 10, 20]$
8. **Keijzer5**: $\frac{3xz}{(x-10)y^2}$; $x, y \in E[-1, 1, 4]; z \in E[1, 2, 4]$
9. **Nguyen10**: $2 \sin(x) \cos(y)$; $x, y \in E[0, 1, 5]$
10. **Nguyen12**: $x^4 - x^3 + \frac{y^2}{2} - y$; $x, y \in E[0, 1, 5]$
11. **Nguyen3**: $x^5 + x^4 + x^3 + x^2 + x$; $x \in E[-1, 1, 20]$
12. **Nguyen4**: $x^6 + x^5 + x^4 + x^3 + x^2 + x$; $x \in E[-1, 1, 20]$
13. **Nguyen5**: $\sin(x^2) \cos(x) - 1$; $x \in E[-1, 1, 20]$

14. **Nguyen6**: $\sin(x) + \sin(x + x^2)$; $x \in E[-1, 1, 20]$
15. **Nguyen7**: $\ln(x + 1) + \ln(x^2 + 1)$; $x \in E[0, 2, 20]$
16. **Nguyen9**: $\sin(x) + \sin(y^2)$; $x, y \in E[0, 1, 5]$
17. **Sext**: $x^6 - 2x^4 + x^2$; $x \in E[-1, 1, 20]$

Appendix B

Results

		Keij1	Keij11	Keij12	Keij13	Keij14	Keij15	Keij4	Keij5	Nguy10	Nguy12	Nguy3	Nguy4	Nguy5	Nguy6	Nguy7	Nguy9	Sext
GP		0.338	0.858	0.97	0.562	0.798	**0.87**	0.6	0.989	0.106	0.38	0.181	**0.247**	0.108	0.017	0.114	0.1	0.102
BP2A	REPTree	**0.243**	0.776	0.972	0.393	0.723	0.883	0.384	0.975	0.11	**0.343**	0.196	0.265	0.037	0.091	0.122	0.068	0.052
	Full Pop	0.272	0.864	0.982	0.565	0.809	0.947	0.397	0.977	0.304	0.393	0.376	0.372	0.081	0.277	0.179	0.214	0.129
	Scikit Learn	0.327	0.769	**0.966**	0.481	0.726	0.907	0.468	0.977	0.199	0.379	0.2	0.285	0.04	0.119	0.127	0.075	0.054
	Randomized	0.289	0.788	0.973	0.462	0.816	0.875	0.504	0.979	0.223	0.385	0.227	0.277	**0.021**	0.044	0.144	0.047	0.075
	Larger Archive	0.286	**0.566**	0.97	0.388	0.742	0.877	0.397	0.977	0.146	0.344	0.192	0.257	0.029	0.112	0.127	0.059	**0.051**
	Different Rates	0.272	0.694	0.976	**0.326**	0.773	0.882	**0.37**	**0.974**	0.165	0.361	0.202	0.284	0.031	0.071	**0.103**	0.042	0.054
BP4	REPTree	0.359	0.852	0.982	0.817	0.872	0.922	0.522	0.993	0.309	0.388	0.193	0.33	0.103	0.133	0.117	0.165	0.127
	Lasso	0.324	0.833	0.985	0.669	0.811	0.916	0.511	0.987	**0.105**	0.365	0.292	0.399	0.13	0.08	0.182	0.103	0.092
	Scikit Learn	0.357	0.684	0.968	0.548	0.776	0.887	0.513	0.991	0.144	0.36	0.266	0.288	0.126	**0.0**	0.104	**0.04**	0.083
BP4A	REPTree	0.319	0.804	0.981	0.765	0.821	0.919	0.505	0.991	0.209	0.386	0.22	0.328	0.088	0.117	0.128	0.194	0.1
	Scikit Learn	0.261	0.811	0.973	0.507	**0.691**	0.94	0.471	0.981	0.264	0.379	0.219	0.273	0.034	0.088	0.115	0.065	0.056
	Randomized	0.338	0.844	0.984	0.705	0.844	0.939	0.56	0.989	0.338	0.414	0.197	0.364	0.104	0.137	0.138	0.145	0.118
	Larger Archive	0.321	0.858	0.982	0.738	0.788	0.922	0.529	0.99	0.271	0.367	0.222	0.348	0.137	0.125	0.136	0.118	0.096
	Different Rates	0.32	0.815	0.98	0.779	0.802	0.913	0.525	0.991	0.226	0.375	**0.145**	0.303	0.112	0.134	0.12	0.215	0.102

Table B.1: Average program error for best of run programs.

		Keij1	Keij11	Keij12	Keij13	Keij14	Keij15	Keij4	Keij5	Nguy10	Nguy12	Nguy3	Nguy4	Nguy5	Nguy6	Nguy7	Nguy9	Sext
GP		44.83	52.4	**41.23**	31.8	27.03	**50.4**	**48.77**	43.93	16.7	**22.8**	24.03	**27.87**	19.53	10.53	28.27	11.7	31.9
BP2A	REPTree	45.63	45.07	64.0	46.83	27.87	62.8	61.73	63.47	29.93	39.27	43.13	49.47	25.73	27.03	36.07	17.73	33.2
	Full Pop	88.0	70.07	102.13	50.6	55.9	115.3	101.63	76.57	53.5	67.97	61.47	77.4	36.73	50.2	64.4	56.83	76.7
	Scikit Learn	**32.24**	35.59	52.82	35.41	32.35	66.29	52.12	58.59	34.82	33.24	43.0	35.82	24.88	28.24	32.82	11.47	33.53
	Randomized	36.33	36.47	48.0	35.53	32.2	56.77	60.57	39.97	31.13	32.57	40.1	38.33	22.8	21.4	31.87	13.9	41.07
	Larger Archive	44.6	37.63	57.2	36.67	**26.87**	54.5	73.63	50.97	32.8	38.8	39.63	50.13	21.97	29.2	33.77	30.53	34.73
	Different Rates	36.47	37.47	57.9	48.43	29.0	64.13	67.57	58.27	31.3	34.27	37.17	43.57	23.9	22.73	32.87	11.93	39.9
BP4	REPTree	36.0	45.03	52.5	37.07	40.7	59.67	62.43	59.87	31.1	32.53	**21.77**	36.93	22.5	22.6	32.9	21.0	41.77
	Lasso	34.73	50.97	63.93	39.23	36.47	56.77	60.93	84.53	**16.13**	35.9	27.23	32.6	20.27	16.53	28.47	13.87	34.1
	Scikit Learn	34.82	**29.18**	54.06	**26.24**	28.71	51.29	65.0	**39.29**	20.59	32.12	26.18	31.76	20.82	**9.0**	**26.0**	**11.35**	**29.71**
BP4A	REPTree	61.93	50.3	89.23	50.1	32.13	67.13	83.07	53.93	31.97	40.37	40.6	48.13	22.2	31.23	33.23	26.73	43.53
	Scikit Learn	45.35	39.06	53.0	31.94	34.06	73.06	48.88	58.12	39.41	31.41	37.41	43.29	24.47	29.0	30.65	16.47	32.94
	Randomized	70.23	53.4	69.17	35.63	34.0	68.27	66.8	56.9	49.07	36.73	37.63	43.83	18.0	29.27	29.97	20.67	53.57
	Larger Archive	55.37	42.93	83.8	52.6	28.8	64.6	64.8	60.07	41.73	42.5	36.37	49.17	16.47	35.03	30.47	20.7	36.17
	Different Rates	42.97	43.17	95.73	66.1	32.17	68.73	97.33	65.13	35.33	36.7	31.67	43.27	**16.37**	31.2	34.1	27.43	52.63

Table B.2: Average program size for best of run programs.

		Keij1	Keij11	Keij12	Keij13	Keij14	Keij15	Keij4	Keij5	Nguy10	Nguy12	Nguy3	Nguy4	Nguy5	Nguy6	Nguy7	Nguy9	Sext
GP		**9.39**	**7.76**	**7.53**	**7.46**	**6.34**	**7.7**	**8.92**	**7.66**	**6.8**	**6.74**	**8.15**	**8.15**	**7.73**	**8.27**	**8.26**	**6.81**	**8.69**
BP2A	REPTree	21.11	20.43	19.36	20.47	15.36	21.04	23.37	38.18	19.34	17.66	20.45	20.93	19.89	20.47	19.32	20.09	21.03
	Full Pop	34.56	26.46	26.06	25.66	22.42	29.0	34.2	42.75	25.2	28.12	32.53	33.36	32.25	33.49	32.89	25.33	33.03
	Scikit Learn	1609.69	1777.26	1757.03	1686.68	1704.65	1768.26	1769.71	1772.31	1746.19	1755.9	1753.05	1752.44	1777.4	1609.44	1755.66	1599.99	1777.08
	Randomized	21.99	19.58	17.89	18.12	14.38	20.95	23.03	33.97	18.35	17.09	21.1	20.33	21.33	20.97	20.25	18.82	21.9
	Larger Archive	17.03	15.88	14.92	15.25	15.75	16.3	18.82	28.03	15.2	14.01	15.79	15.99	15.37	15.82	15.14	16.15	16.64
	Different Rates	20.33	21.61	19.53	21.15	14.51	22.18	23.44	42.84	20.43	17.87	19.79	19.77	18.69	18.99	18.5	21.55	20.47
BP4	REPTree	19.8	17.9	18.61	20.13	15.19	18.98	21.67	37.59	18.59	19.6	20.08	18.37	17.67	20.44	19.46	21.12	20.88
	Lasso	428.52	82.52	88.5	1715.48	325.04	54.02	2850.23	234.47	45.0	111.22	62.42	71.2	57.85	47.13	35.99	45.53	576.13
	Scikit Learn	1593.13	1601.44	1597.46	1606.18	1642.73	1618.15	1608.75	1605.08	1595.05	1609.67	1603.42	1608.03	1603.09	1591.73	1591.2	1604.62	1594.9
BP4A	REPTree	23.67	22.05	21.35	22.68	16.98	22.39	25.59	42.0	21.66	22.0	22.61	22.6	21.56	23.88	21.63	23.57	23.81
	Scikit Learn	1675.06	1687.09	1666.51	1684.53	1702.18	1658.17	1686.16	1672.19	1661.06	1680.43	1676.39	1665.83	1675.35	1682.23	1671.73	1690.09	1668.25
	Randomized	21.96	19.18	19.07	19.46	16.43	19.84	22.25	35.63	19.48	19.21	20.5	19.56	19.49	21.34	19.18	20.61	22.06
	Larger Archive	21.68	19.84	19.82	20.41	17.35	20.51	23.39	35.57	19.81	19.51	20.42	20.54	19.31	21.5	19.49	21.59	21.9
	Different Rates	19.64	18.82	19.78	20.52	16.04	20.02	21.53	39.21	19.14	18.83	18.35	18.59	17.14	18.96	17.2	22.13	20.14

Table B.3: Average runtime in seconds.

		Keij1	Keij11	Keij12	Keij13	Keij14	Keij15	Keij4	Keij5	Nguy10	Nguy12	Nguy3	Nguy4	Nguy5	Nguy6	Nguy7	Nguy9	Sext
GP		0.116	0.155	0.032	0.36	0.077	0.071	0.155	0.005	0.15	0.039	0.163	0.133	0.14	0.07	0.073	0.211	0.079
BP2A	REPTree	0.106	0.242	0.012	0.31	0.061	0.064	0.126	0.018	0.131	0.043	0.108	0.089	0.032	0.107	0.073	0.091	**0.018**
	Full Pop	0.091	**0.093**	0.01	0.333	0.071	**0.019**	0.12	0.016	0.141	0.066	0.143	0.097	0.082	0.14	0.126	0.182	0.086
	Scikit Learn	0.084	0.274	0.033	0.357	0.106	0.042	0.184	0.024	0.144	0.03	0.104	**0.061**	0.043	0.13	0.071	0.148	0.037
	Randomized	0.118	0.271	0.023	0.365	0.099	0.087	0.178	0.016	0.13	**0.028**	0.111	0.094	**0.008**	0.071	0.065	0.112	0.077
	Larger Archive	0.107	0.399	0.017	0.338	0.089	0.084	0.128	0.022	0.138	0.052	0.108	0.097	0.027	0.099	0.065	0.091	0.025
	Different Rates	0.111	0.333	0.014	0.295	0.077	0.05	0.123	0.021	0.156	0.04	0.11	0.088	0.026	0.08	0.059	**0.073**	0.028
BP4	REPTree	0.062	0.118	0.013	0.193	**0.04**	0.039	0.144	0.005	0.128	0.047	0.19	0.149	0.087	0.136	0.076	0.174	0.072
	Lasso	0.082	0.217	0.006	0.305	0.077	0.061	0.127	0.014	0.148	0.054	0.145	0.103	0.101	0.143	0.085	0.162	0.051
	Scikit Learn	**0.041**	0.356	0.02	0.412	0.066	0.048	0.15	0.005	0.167	0.042	0.161	0.11	0.135	**0.0**	0.07	0.096	0.043
BP4A	REPTree	0.087	0.231	0.01	**0.165**	0.066	0.036	0.119	0.008	0.165	0.034	0.108	0.099	0.066	0.113	0.06	0.174	0.052
	Scikit Learn	0.121	0.191	0.014	0.381	0.077	0.03	0.162	0.015	0.156	0.038	0.119	0.093	0.022	0.094	**0.059**	0.133	0.019
	Randomized	0.088	0.177	**0.006**	0.251	0.055	0.029	0.119	0.007	**0.12**	0.038	**0.103**	0.137	0.098	0.13	0.088	0.169	0.071
	Larger Archive	0.1	0.146	0.012	0.175	0.062	0.042	**0.114**	0.006	0.143	0.063	0.119	0.116	0.103	0.102	0.076	0.15	0.048
	Different Rates	0.079	0.203	0.008	0.178	0.066	0.044	0.138	**0.003**	0.154	0.038	0.113	0.102	0.089	0.134	0.061	0.183	0.058

Table B.4: Standard deviation of program error for best of run programs.

		Keij1	Keij11	Keij12	Keij13	Keij14	Keij15	Keij4	Keij5	Nguy10	Nguy12	Nguy3	Nguy4	Nguy5	Nguy6	Nguy7	Nguy9	Sext
GP		23.836	31.9	**17.99**	15.709	12.823	21.296	44.228	26.52	13.503	**6.901**	12.454	**8.655**	8.597	4.863	7.33	10.571	18.063
BP2A	REPTree	20.034	19.152	28.776	24.376	**8.523**	24.437	32.567	33.433	19.176	14.861	19.052	22.007	8.481	17.186	19.084	13.945	12.973
	Full Pop	60.071	29.06	93.357	28.114	26.739	58.357	43.987	42.429	36.081	43.072	26.065	32.648	26.125	18.721	43.892	35.947	59.331
	Scikit Learn	**9.434**	14.32	23.365	14.492	11.225	24.374	**23.392**	27.231	19.989	14.489	18.14	24.818	7.91	16.112	14.197	12.306	11.932
	Randomized	14.912	12.876	23.897	15.75	11.6	24.958	37.929	25.57	12.746	11.488	18.922	14.704	11.321	16.122	14.125	12.869	24.503
	Larger Archive	33.811	20.009	31.578	24.281	8.686	20.306	24.68	27.946	22.33	15.587	23.791	29.917	6.135	18.369	13.691	20.467	11.24
	Different Rates	18.016	17.905	25.438	37.239	10.106	22.753	40.731	35.773	17.508	16.935	17.846	20.513	7.059	12.928	12.543	**7.793**	16.644
BP4	REPTree	17.631	31.965	26.529	21.627	29.371	25.234	62.81	39.799	17.887	22.235	**10.125**	11.448	12.008	14.753	21.973	24.28	25.337
	Lasso	19.482	29.946	32.804	24.2	21.48	24.776	67.692	67.565	**11.566**	15.129	10.449	15.239	9.194	12.868	22.684	9.58	25.509
	Scikit Learn	18.822	**7.906**	18.552	**6.907**	17.398	**15.888**	65.088	**20.219**	14.709	7.692	14.916	16.311	7.778	**0.0**	**6.937**	13.754	**8.93**
BP4A	REPTree	129.111	27.786	54.411	30.277	20.175	29.212	42.227	38.899	25.492	17.227	23.807	23.523	14.822	18.16	14.042	23.694	28.724
	Scikit Learn	23.8	17.134	24.9	12.436	11.904	39.492	31.717	47.524	19.635	12.381	13.439	20.795	7.022	20.722	13.061	19.327	12.563
	Randomized	51.385	35.059	44.184	25.193	24.931	41.312	35.23	44.464	44.082	18.081	17.325	13.297	**5.385**	18.613	14.063	19.719	39.541
	Larger Archive	37.992	26.554	44.508	33.6	13.965	29.554	32.115	45.038	27.465	18.599	17.316	27.39	10.016	16.447	12.989	18.163	15.644
	Different Rates	26.565	19.868	57.523	52.438	15.269	29.461	71.036	43.696	24.241	16.265	13.595	20.525	6.661	17.194	13.504	22.915	50.013

Table B.5: Standard deviation of program size for best of run programs.

		Keij1	Keij11	Keij12	Keij13	Keij14	Keij15	Keij4	Keij5	Nguy10	Nguy12	Nguy3	Nguy4	Nguy5	Nguy6	Nguy7	Nguy9	Sext
GP		2.546	0.861	**0.644**	**1.03**	**0.544**	**0.791**	1.642	**1.086**	0.955	**0.5**	**0.469**	**0.323**	**0.496**	**0.394**	**0.601**	**0.504**	1.088
BP2A	REPTree	1.261	1.782	2.56	2.182	0.841	2.295	1.742	7.382	1.1	1.442	1.161	1.325	0.855	0.945	0.855	1.234	1.125
	Full Pop	3.071	3.257	4.284	2.796	2.555	4.886	2.559	4.849	3.139	5.65	2.463	2.814	3.229	2.596	3.152	2.445	2.95
	Scikit Learn	41.523	450.092	397.491	283.709	271.904	422.125	389.593	398.268	393.374	392.588	394.007	404.207	443.902	41.234	406.231	22.835	434.365
	Randomized	1.08	1.472	2.001	1.331	0.914	2.091	2.934	6.304	**0.905**	1.19	1.069	0.987	1.109	0.94	0.859	1.034	2.472
	Larger Archive	2.233	1.179	1.624	1.399	1.009	1.627	1.907	4.595	1.142	1.321	0.95	1.123	0.684	0.827	0.884	1.165	**0.899**
	Different Rates	1.84	1.715	2.41	2.165	0.972	1.972	3.243	6.61	1.479	1.906	1.399	1.366	0.706	0.642	1.029	1.454	1.546
BP4	REPTree	**0.782**	0.979	1.249	4.005	1.448	1.167	2.565	3.113	0.925	1.438	0.708	0.591	0.538	0.841	1.633	2.35	1.537
	Lasso	74.725	19.534	44.328	3743.796	122.409	10.526	4310.414	85.3	10.16	18.171	6.482	9.859	8.585	5.559	4.887	6.18	1310.848
	Scikit Learn	32.654	71.251	37.349	56.105	75.409	64.537	61.468	44.062	33.641	33.006	35.994	40.057	58.308	42.858	38.57	55.505	48.323
BP4A	REPTree	2.863	1.169	1.558	2.062	0.781	1.619	1.886	4.415	1.022	1.49	1.537	1.385	0.91	1.415	0.835	1.897	1.397
	Scikit Learn	204.655	203.663	192.772	198.467	191.264	178.164	208.586	204.726	184.522	192.713	186.326	197.194	201.328	204.377	203.64	205.414	163.618
	Randomized	1.446	0.938	1.655	2.295	1.017	1.906	1.662	3.201	1.348	1.061	1.349	0.733	0.73	1.104	0.671	1.411	1.432
	Larger Archive	1.259	1.387	1.387	2.115	0.842	1.556	**1.509**	2.864	1.613	1.129	1.151	1.206	0.83	1.091	1.056	1.627	1.099
	Different Rates	1.313	**0.826**	1.612	2.425	0.727	1.839	2.051	4.254	1.266	1.094	0.8	1.176	0.602	1.398	0.803	4.092	1.135

Table B.6: Standard deviation of runtime in seconds.

		Keij1	Keij11	Keij12	Keij13	Keij14	Keij15	Keij4	Keij5	Nguy10	Nguy12	Nguy3	Nguy4	Nguy5	Nguy6	Nguy7	Nguy9	Sext
GP		0.0	0.0	0.0	3.333	0.0	0.0	0.0	0.0	**63.333**	0.0	36.667	0.0	0.0	90.0	0.0	80.0	0.0
BP2A	REPTree	0.0	3.333	0.0	3.333	0.0	0.0	0.0	0.0	33.333	0.0	10.0	0.0	0.0	43.333	0.0	30.0	0.0
	Full Pop	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.333	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Scikit Learn	0.0	0.0	0.0	5.882	0.0	0.0	0.0	0.0	17.647	0.0	11.765	0.0	0.0	35.294	0.0	76.471	0.0
	Randomized	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	13.333	0.0	6.667	0.0	0.0	60.0	0.0	70.0	0.0
	Larger Archive	0.0	**16.667**	0.0	0.0	0.0	0.0	0.0	0.0	20.0	0.0	10.0	0.0	0.0	30.0	0.0	26.667	0.0
	Different Rates	0.0	0.0	0.0	10.0	0.0	0.0	0.0	0.0	30.0	0.0	10.0	0.0	0.0	43.333	0.0	53.333	0.0
BP4	REPTree	0.0	0.0	0.0	3.333	0.0	0.0	0.0	0.0	0.0	0.0	**43.333**	0.0	0.0	46.667	0.0	40.0	0.0
	Lasso	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	56.667	0.0	13.333	0.0	0.0	70.0	0.0	46.667	0.0
	Scikit Learn	0.0	11.765	0.0	**11.765**	0.0	0.0	0.0	0.0	47.059	0.0	17.647	0.0	0.0	**100.0**	0.0	**82.353**	0.0
BP4A	REPTree	0.0	3.333	0.0	0.0	0.0	0.0	0.0	0.0	23.333	0.0	10.0	0.0	0.0	33.333	0.0	20.0	0.0
	Scikit Learn	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.882	0.0	11.765	0.0	0.0	35.294	0.0	52.941	0.0
	Randomized	0.0	0.0	0.0	3.333	0.0	0.0	0.0	0.0	0.0	0.0	10.0	0.0	0.0	36.667	0.0	40.0	**3.333**
	Larger Archive	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	6.667	0.0	10.0	0.0	0.0	20.0	0.0	33.333	0.0
	Different Rates	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	16.667	0.0	23.333	0.0	0.0	26.667	0.0	16.667	0.0

Table B.7: Percentage of runs that generated a perfect individual.

Appendix C

Fixed Parameters

- **Tournament size**: 4
- **Population size**: 100
- **Number of Generations**: 250
- **Maximum Program Tree Depth**: 17
- **Function set**: $\{+, -, *, /, \log, \exp, \sin, \cos, -x\}$
- **Terminal set**: Only the features in the data set.

Appendix D

Run Configurations

D.1 Key

- μ: mutation rate
- χ: crossover rate
- α: archive-based crossover rate
- f: program error fitness function
- s: program size fitness function
- c: model complexity fitness function
- e: model error fitness function

D.2 Configurations

1. **GP**:
    - Rates: $\mu = 0.1; \chi = 0.9; \alpha = 0.0$
    - Fitness Functions: $\{f, s\}$
    - Archive Capacity: No Archive
    - Model: No Model
2. **BP2A - REPTree**:
    - Rates: $\mu = 0.1; \chi = 0.0; \alpha = 0.9$

- Fitness Functions: $\{f, s\}$
- Archive Capacity: 50
- Model: REPTree run on the trace of each program in the population

3. **BP2A - Full Pop**:

   - Rates: $\mu = 0.1; \chi = 0.0; \alpha = 0.9$
   - Fitness Functions: $\{f, s\}$
   - Archive Capacity: 50
   - Model: REPTree run on the combined traces of all of the programs in the population

4. **BP2A - Lasso**:

   - Rates: $\mu = 0.1; \chi = 0.0; \alpha = 0.9$
   - Fitness Functions: $\{f, s\}$
   - Archive Capacity: 50
   - Model: Lasso run on the trace of each program in the population

5. **BP2A - Scikit Learn**:

   - Rates: $\mu = 0.1; \chi = 0.0; \alpha = 0.9$
   - Fitness Functions: $\{f, s\}$
   - Archive Capacity: 50
   - Model: Scikit Learn DecisionTreeRegressor run on the trace of each program in the population

6. **BP2A - Randomized**:

   - Rates: $\mu = 0.1; \chi = 0.0; \alpha = 0.9$
   - Fitness Functions: $\{f, s\}$
   - Archive Capacity: 50
   - Model: REPTree run on the trace of each program in the population, but only the weights are kept. The subtrees corresponding to the weights that

are placed in the archive are drawn uniformly at random from the program subtrees

7. **BP2A - Larger Archive**:

   - Rates: $\mu = 0.1; \chi = 0.0; \alpha = 0.9$
   - Fitness Functions: $\{f, s\}$
   - Archive Capacity: 150
   - Model: REPTree run on the trace of each program in the population

8. **BP2A - Different Rates**:

   - Rates: $\mu = 0.05; \chi = 0.0; \alpha = 0.95$
   - Fitness Functions: $\{f, s\}$
   - Archive Capacity: 50
   - Model: REPTree run on the trace of each program in the population

9. **BP4- REPTree**:

   - Rates: $\mu = 0.1; \chi = 0.9; \alpha = 0.0$
   - Fitness Functions: $\{f, s, c, e\}$
   - Archive Capacity: No Archive
   - Model: REPTree run on the trace of each program in the population

10. **BP4- Lasso**:

    - Rates: $\mu = 0.1; \chi = 0.9; \alpha = 0.0$
    - Fitness Functions: $\{f, s, c, e\}$
    - Archive Capacity: No Archive
    - Model: Lasso run on the trace of each program in the population

11. **BP4- Scikit Learn**:

    - Rates: $\mu = 0.1; \chi = 0.9; \alpha = 0.0$
    - Fitness Functions: $\{f, s, c, e\}$
    - Archive Capacity: No Archive

- Model: Scikit Learn DecisionTreeRegressor run on the trace of each program in the population

12. **BP4A- REPTree**:

    - Rates: $\mu = 0.1; \chi = 0.0; \alpha = 0.9$
    - Fitness Functions: $\{f, s, c, e\}$
    - Archive Capacity: 50
    - Model: REPTree run on the trace of each program in the population

13. **BP4A- Lasso**:

    - Rates: $\mu = 0.1; \chi = 0.0; \alpha = 0.9$
    - Fitness Functions: $\{f, s, c, e\}$
    - Archive Capacity: 50
    - Model: Lasso run on the trace of each program in the population

14. **BP4A- Scikit Learn**:

    - Rates: $\mu = 0.1; \chi = 0.0; \alpha = 0.9$
    - Fitness Functions: $\{f, s, c, e\}$
    - Archive Capacity: 50
    - Model: Scikit Learn DecisionTreeRegressor run on the trace of each program in the population

15. **BP4A- Randomized**:

    - Rates: $\mu = 0.1; \chi = 0.0; \alpha = 0.9$
    - Fitness Functions: $\{f, s, c, e\}$
    - Archive Capacity: 50
    - Model: REPTree run on the trace of each program in the population, but only the weights are kept. The subtrees corresponding to the weights that are placed in the archive are drawn uniformly at random from the program subtrees

16. **BP4A- Larger Archive**:

    - Rates: $\mu = 0.1; \chi = 0.0; \alpha = 0.9$
    - Fitness Functions: $\{f, s, c, e\}$
    - Archive Capacity: 150
    - Model: REPTree run on the trace of each program in the population

17. **BP4A- Different Rates**:

    - Rates: $\mu = 0.05; \chi = 0.0; \alpha = 0.95$
    - Fitness Functions: $\{f, s, c, e\}$
    - Archive Capacity: 50
    - Model: REPTree run on the trace of each program in the population

Bibliography

[1] Ignacio Arnaldo, Krzysztof Krawiec, and Una-May O'Reilly. Multiple regression genetic programming. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 879–886. ACM, 2014.

[2] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[3] Pavlos S Efraimidis and Paul G Spirakis. Weighted random sampling with a reservoir. *Information Processing Letters*, 97(5):181–185, 2006.

[4] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[5] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[6] Krzysztof Krawiec and Una-May O'Reilly. Behavioral programming: a broader and more detailed take on semantic gp. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 935–942. ACM, 2014.

[7] James McDermott, David R White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Jaskowski, Krzysztof Krawiec, Robin Harper, Kenneth De Jong, et al. Genetic programming needs better benchmarks. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 791–798. ACM, 2012.

[8] Kirill Müller. Accelerating weighted random sampling without replacement. *Arbeitsberichte Verkehrs-und Raumplanung*, 1141, 2016.

[9] Kalyan Veeramachaneni, Ignacio Arnaldo, Owen Derby, and Una-May O'Reilly. Flexgp. *Journal of Grid Computing*, 13(3):391–407, 2015.