Exploiting Subprograms in Genetic Programming

Anonymous
Anonymous
Anonymous

Anonymous
Anonymous
Anonymous

Anonymous
Anonymous
unamay@csail.mit.edu

Anonymous
Anonymous
Anonymous

ABSTRACT

Compelled by the importance of subprogram behavior, we investigate how much Behavioral Genetic Programming is sensitive to model bias. We experimentally compare two different decision tree algorithms analyzing whether it is possible to see significant performance differences given the model techniques select different subprograms and differ in how accurately they can regress subprogram behavior on desired outputs. We find no remarkable difference between REPTree and CART in this regard, though for a modest fraction of our datasets we find one algorithm or the other results in superior error reduction than the other. We also investigate alternate ways to identify useful subprograms beyond examining those within one program. We propose a means of identifying subprograms from different programs that work well together. It combines behavioral traces from multiple programs and uses the information derived from modeling with the combination. We find XX.

KEYWORDS

program synthesis; genetic programming; program semantics; behavioral evaluation; search operators; archive; multi-objective evolutionary computation

ACM Reference format:
Anonymous, Anonymous, Anonymous, and Anonymous. 2017. Exploiting Subprograms in Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference 2017, Berlin, Germany, July 15–19, 2017 (GECCO '17),* 6 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Our general goal is to improve the level of program complexity that genetic programming(GP) can routinely evolve[7]. This is toward fulfilling its potential to occupy a significant niche in the ever advancing field of program synthesis[4, 15, 19]. Behavioral genetic programming (BGP) is an extension to GP that advances toward this compelling goal[8, 9]. The intuition of the BGP paradigm is that, during evolutionary search and optimization, information characterizing programs by behavioral properties that extend beyond how accurately they match their target outputs. This information from a program "trace" can be effectively integrated into extensions of the algorithm's fundamental mechanisms of fitness-based selection and genetic variation.

To identify useful subprograms BGP commonly exploits program *trace* information (first introduced in [10] which is a capture of the output of every subprogram within the program for every test data point during fitness evaluation. The trace is stored in a matrix where the number of rows is equal to the number of test suite data points, and the number of columns is equal to the number of subtrees in a given program. The trace captures a full snapshot of all of the intermediate states of the program evaluation.

BGP then uses the trace to estimate the merit of each subprogram by treating its column as a feature (or explanatory variable) in a *model regression* on the desired program outputs. The accuracy and complexity of the model reveals how useful the subprograms are. The model, if it has feature selection capability, also reveals specific subprograms within the tree that are partially contributing to the program's fitness. BGP uses this information in two ways. It can integrate *model error* and *model complexity* into the program's fitness. Second, it maintains an archive of the most useful subtrees identified by modeling each program and uses them in an *archive-based crossover.* BGP has a number of variants that collectively yield impressive results, see [8, 9].

In this work, we explore various extensions to the BGP paradigm that are motivated by 2 central topics:

(1) **The impact of bias from the inference model on useful subprogram identification and program fitness.** Model techniques and even the implementation of the same technique can differ in inductive *bias*, i.e. error from assumptions in the learning algorithm, e.g. implementation of a "decision tree" algorithm. These differences, in turn, impact which subprograms are inserted/retrieved from the BGP archive and the model accuracy and model complexity factors that are integrated into a program's fitness. Therefore, we investigate how much BGP is sensitive to model bias. *How important is it which subprograms the model technique selects and how accurate a model is?* We answer these questions by comparing BGP competence under 2 different implementations of decision tree modeling which we observe

have different baises. Our investigation contrasts feature identification and model accuracy under the two implementations.

(2) **"Plays well with others?": Alternate ways to identify useful subprograms**. BGP uses the trace matrix from a program implying all the subprograms of one program are treated as a set of features during modeling. This means that feature selection and subprogram fitness estimation occurs within the program context. Essentially, each subprogram is juxtaposed with "relatives" – its parent, neighbors and even child in GP tree terms. Does this context provide the best means of identifying useful subprograms? It may not. Crossover moves a subprogram into another program so, to work most effectively, it should explore recombinations of subprograms that *work well in other subprograms and programs in the population.* We examine this idea by concatenating program traces from a set of programs, not solely one program. This demands a new measure of fitness to reflect subprogram participation in the model as a means of archive selection and program fitness. We examine concatenation of the entire population and sub-populations based on fitness to ask whether fitter programs with concatenated traces are more effective than an open playing field. i.e. the entire population of subprograms.

Our specific demonstration focus herein is symbolic regression. We choose SR because it remains a challenge and have good benchmarks [12] so it allows us to measure progress and extensions. It also has real world application to system identification and, with modest modification, machine learning regression and classification. In passing, we replicate BGP logic, making our project software available with an open source license.

We proceed as follows. We start with related work in Section 2. In Section 3 we provide descriptions of our methods of comparison and investigation. In Section 4 we provide implementation and experimental details and results. Section 5 concludes and mentions future work.

2 RELATED WORK

BGP is among a number of other approaches to program synthesis where progress has recently become more empirically driven, rather than driven by formal specifications and verification[1]. Alternate approaches to evolutionary algorithms include e.g. sketching[16], i.e. communicating insight through a partial program, generalized program verification[17], as well as hybrid computing with neural network and external mebory[3]

BGP takes inspiration, with respect to its focus on program behavior, from earlier work on implicit fitness sharing[13], trace consistency analysis and the use of archives as a form of memory[6]. In its introduction in [?] the preceding introduction of the trace matrix was noted within a system called Pattern Guided Genetic Programming, "PANGEA". In PANGEA minimum description length was induced from the program execution. Subsequently there have been a variety of extensions. For example, in the general vein of behaviorally characterizing a program by more than its accuracy [11] considers discovering search objectives for test-based problems. Also notable is Memetic Semantic Genetic Programming[2].

BGP was introduced as aa broader and more detailed take on Semantic GP. BGP and Semantic GP share the common goal of characterising program behaviour in more detail and exploiting this information. Whereas BGP looks at subprograms, semantic GP focuses on program output. Output is scrutinized for every test individually and a study of the impact of crossover on program semantics and semantic building blocks [14] was conducted. A survey of semantic models in GP [18] gives an overview of methods for their operators, as well as different objectives.

3 EXPLOITING SUBPROGRAMS

3.1 BGP Strategy

What emerges from the details of BGP's successful examples is a stepwise strategy:

(1) Capture the behavior of subprograms within a program in a trace matrix T.

(2) Regress T as feature data on the desired program outputs and derive a model M.

(3) Assign a value of merit w to each subprogram within M. Use this merit to determine whether it should be inserted into an archive. Use a modified crossover that draws subprograms from the archive.

(4) Integrate model error and complexity into program fitness so that subprogram behavior influences selection.

One example of the strategy is realized in [9] where, in Step (2), the fast RepTree (REPT- Reduced Error Pruning Tree) algorithm of decision tree classification from the WEKA Data Mining software library[5] is used for regression modeling. REPT builds a decision/regression tree using information gain/variance. In Step (3) merit is measured per Equation 1 where $|U(p)|$ is the number of subprograms (equivalently distinct columns of the trace) used in the model and e is model error.

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

3.2 Exploring Model Bias

Following our motivation to understand the impact of model bias on useful subprogram identification and program fitness, we first explore an alternative realization of BGP's strategy by using the CART optimized version of the CART decision tree algorithm[1]. CART (Classification and Regression Trees) is very similar to C4.5, but it differs in that it supports numerical target variables (regression) and does not compute rule sets. CART constructs binary trees using the feature and threshold that yield the largest information gain at each node. With the CART implementation we derive a model we denote by M_S and contrast it to deriving M which for clarity

[1]http://scikit-learn.org/stable/modules/tree.html#tree-algorithms-id3-c4-5-c5-0-and-cart

we now denote as M_R with M subscripted S for CART and R for REPTree.

3.3 Identifying Useful Subprograms

Next, following our goal to investigate alternate ways to identify useful subprograms based upon the observation that prior work derives useful subtree fitness from a model that references just the feature set of *one* program, we realize Steps (1) and (4) alternately. In Step (1) we first select a set of programs C from the population. We then form a new kind of trace matrix, T_c, by column-wise concatenating all T's of the programs in C. In a version we call **FULL**, T_c is then passed through Step (2). Steps (3) and (4) are omitted because in each generation only a single machine learning model is built so all of the selected trees put into the archive in a single generation have the same weight. This realization of the strategy allows us to experiment with different programs in C, trying C containing all the programs in the population, for diversity and, conversely trying elitism, holding only a top fraction of the population by fitness.

An alternate implementation, that we name **DRAW**, is to draw random subsets from the combined trace of the programs in C, 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. We modify Step (4) because subprograms in M come from the set T_C, not solely one program. The information we can integrate into program fitness is whether a program contributed a feature to M. For this we use w_c, see Equation 2 which weighs how many times a program's subprogram appeared in the model normalized by the number of features in the model. Let p' be the number of subtrees in the model from program p. We define the weight as w_c per Equation 2

$$w_c = 1 - \frac{p'}{U(M)} \tag{2}$$

Implementation details of **FULL** and of **DRAW** are provided the next section.

4 EXPERIMENTS

We start this section by detailing the benchmarks we use, the parameters of our algorithms and name our algorithm configurations for convenience. Section 4.2 then evaluates the impact of different decision tree algorithms. Section 4.3 the performance of **FULL** and **DRAW** for different configurations then compares the **FULL**, **DRAW** and program-based model techniques.

4.1 Experimental Data, Parameters

Our investigation uses 17 symbolic regression benchmarks from [12]. All of the benchmarks are defined such that the dependent variable is the output of a particular mathematical function for a given set of inputs. 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 benchmarks 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**: $x^6 - 2x^4 + x^2$; $x \in E[-1, 1, 20]$

We use a standard implementation of GP and chose parameters according to settings documented in [9].

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 benchmark.
- **Archive Capacity**: 50
- **Mutation Rate** μ: 0.1
- **Crossover Rate with Archive configuration** χ: 0.0
- **Crossover Rate with GP** χ: 0.9
- **Archive-Based Crossover Rate** α: 0.9
- **REPTree** defaults but no pruning
- **CART** defaults
- **Number of runs** 30

First we used 3 BGP algorithm configurations that use REPT to replicate [9]'s work on the symbolic regression benchmarks. These we call BGP2A, BGP4, BGP4A following precedent. In the name the digit 2 indicates that model error e and complexity z were not integrated into program fitness while 4 indicates they were. The suffix A indicates whether or not subprograms from the model were qualified for archive insertion and archive retrieval during BGP crossover. We observed results consistent with the prior work. Our open source software is available on GIT. This allowed us to proceed to evaluate feature selection sensitivity based on modeling algorithm.

4.2 Sensitivity to Model Bias

Q1. Does the feature selection bias of the model step matter?

Table 1 shows the results of running the 3 different configurations (BGP2A, BGP4, BGP4A) with different decision

	Configuration	Average Rank	
1	BP2A	REPT	1.82
2	BP2A	CART	2.94
3	BP4A	CART	3.06
4	BP4	CART	3.18
5	BP4A	REPT	4.65
6	BP4	REPT	5.35

Table 1: Comparison of impact of REPT vs CART for average fitness rank across all data sets.

tree algorithms. Averaged over rankings for each benchmark BP2A using REPT was best. For BP2A REPT outranked CART but when model error was integrated into the program fitness, whether an archive was used or not made no difference to CART beating REPT rankings by fitness, averaged across the benchmark set.

When we compare the result of using the archive while model error is integrated into the program fitness, for both REPT and CART it is better to use an archive than to forgo one. Comparing BP2A with BP4A where the configurations uses an archive CART or for REPT allows one to measure the impact of model error integration. In both cases it is not helpful to integrate model error into program fitness.

For a deeper diver, Table 2 shows the average best fitness at end of run (of 30 runs), for each benchmark. Averaging all fitness results, no clear winner is discernible. For certain comparisons CART will be superior while for others REPT is. We also show one randomly selected run of Keijzer1 running with REPT modeling and configuration BP4. We plot on the first row model error on the left and the fitness of the best program (right). The plots on the second row show number of features of model and number of subprograms in the best program (right). The plots on the third row show the ratio of number of model features to program subtrees (left) and ratio of model error to program fitness. Since the run is configured for BP4 program fitness integrates both model error and complexity.

We conclude that in this case of different decision tree algorithms perhaps the subtlety of contrast is not strong enough.

4.3 Aggregate Trace Matrices

For the algorithm configurations of this section which compare within **FULL** and within **DRAW**, we adopt a clearer notation. We drop the BGP prefix and we use M or $\hat{M}$ to designate whether model error was integrated into program fitness and A of $\hat{A}$ to designate whether subprograms were were qualified for archive insertion and archive retrieval during BGP crossover.

More details of the **DRAW** method are appropriate. We considered the model error tagged to a subprogram prior to archive entry by the BGP realization. It enhances the fitness of smaller subprograms by the $U|p|$ factor in its denominator.

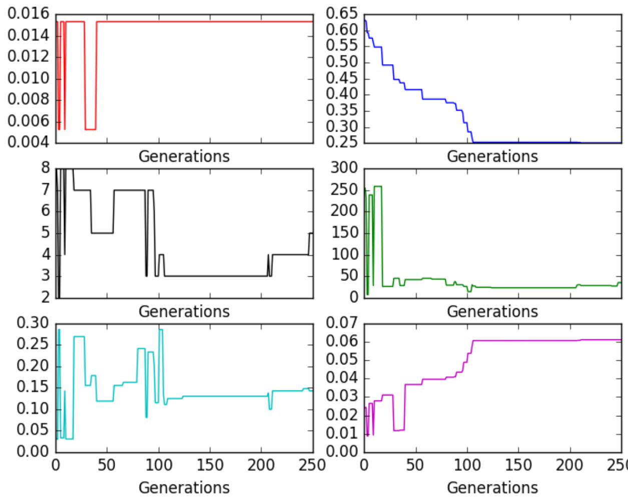

Figure 1: We take one run of Keijzer1 running with REPT modeling and configuration BP4. We plot on the first row model error on the left and the fitness of the best program (right). The plots on the second row show number of features of model and number of subprograms in the best program (right). The plots on the third row show the ratio of number of model features to program subtrees (left) and ratio of model error to program fitness. Since the run is configured for BP4 program fitness integrates both model error and complexity.

Therefore we designed **DRAW** to behave similarly. **DRAW** proceeds as follows:

(1) The population is sorted best to worst by program fitness (program error and size) using the NSGA crowding calculation.

(2) The sorted population is cut off from below at a threshold λ% to form C. The trace matrixes of every program in C are concatenated to form T_C which we call the subprogram pool. We sort C by size and select the smallest 20% of elements of C forming a sample we call K.

(3) Finally we draw from K at random a size. Then we select the equivalent number of columns at random from T_C and form a model. We repeat this step each time for the size of the population.

Q2. Can trace matrix concatenation which pools subprograms among different programs improve BGP performance?

We first asked what if $|C| = PopSize$? If we compare the result of **DRAW** to **FULL** we can gauge the difference of generating many more small models vs one bigger model. See the bottom line of Table!3 and

5 CONCLUSIONS AND FUTURE WORK

The paper's primary contributions are to explore two experimentally driven questions. The first question concerned the

		Keij1	Keij11	Keij12	Keij13	Keij14	Keij15	Keij4	Keij5	Nguy10	Nguy12	Nguy3	Nguy4	Nguy5	Nguy6	Nguy7	Nguy9	Sext
BP2A	REPT	**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**
	CART	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
BP4	REPT	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
	CART	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	REPT	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
	CART	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

Table 2: Average program error for best of run programs.

C	$\bar{M}A$	$M\bar{A}$	MA	$\bar{M}A$	$M\bar{A}$	MA
25	3.06	2.24	2.35	**1.65**	2.18	**1.41**
50	**2.29**	**1.82**	**1.88**	2.41	**1.88**	2.29
75	**2.29**	2.0	2.0	3.06	2.12	2.65
100	2.35	3.94	3.76	2.88	3.82	3.65

Table 3: DRAW (lhs) and FULL (rhs) average rank varying model fitness signal (M or $\hat{M}$) and use of archive (A or $\hat{A}$) for 17 benchmarks

		Keij1	Keij11	Keij12	Keij13	Keij14	Keij15	Keij4	Keij5	Nguy10	Nguy12	Nguy3	Nguy4	Nguy5	Nguy6	Nguy7	Nguy9	Sext
	Draw 25	0.306	0.704	0.976	0.436	0.768	**0.845**	0.371	0.975	0.162	0.341	**0.172**	0.301	0.056	0.074	0.132	0.241	0.058
$\bar{M}A$	Draw 50	0.286	**0.604**	0.969	0.422	0.731	0.866	0.376	**0.967**	0.107	0.353	0.194	0.295	0.045	0.089	**0.103**	0.159	0.059
	Draw 75	**0.246**	0.716	0.968	**0.325**	0.736	0.869	0.347	0.974	**0.089**	0.351	0.215	0.278	0.06	0.1	0.118	0.206	**0.044**
	Draw 100	0.253	0.695	0.969	0.382	**0.716**	0.877	**0.33**	0.972	0.123	0.356	0.217	0.285	**0.03**	0.129	0.115	0.165	0.047
	Full 25	0.278	0.812	**0.956**	0.621	0.761	0.88	0.457	0.977	0.232	0.385	0.297	0.33	0.058	0.159	0.204	0.212	0.062
	Full 50	0.279	0.883	0.979	0.564	0.748	0.921	0.411	0.981	0.294	0.387	0.337	0.37	0.06	0.264	0.195	0.301	0.086
	Full 75	0.302	0.864	0.976	0.604	0.804	0.925	0.453	0.982	0.364	0.395	0.316	0.361	0.059	0.271	0.225	0.306	0.088
	Full 100	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
	Draw 25	0.322	0.89	0.979	0.732	0.786	0.889	0.601	0.991	0.185	0.361	0.233	0.283	0.107	0.103	0.144	0.197	0.076
$M\bar{A}$	Draw 50	0.314	0.824	0.979	0.697	0.798	0.888	0.55	0.986	0.183	0.393	0.307	0.322	0.081	0.088	0.15	0.165	0.081
	Draw 75	0.337	0.865	0.979	0.723	0.819	0.886	0.562	0.99	0.22	0.356	0.236	0.246	0.064	0.108	0.136	0.242	0.088
	Draw 100	0.367	0.908	0.986	0.879	0.846	0.962	0.598	0.993	0.377	0.43	0.363	0.442	0.156	0.186	0.246	0.267	0.127
	Full 25	0.288	0.875	0.973	0.478	0.783	0.867	0.516	0.987	0.163	0.346	0.23	0.302	0.072	**0.069**	0.119	0.174	0.093
	Full 50	0.301	0.851	0.967	0.463	0.834	0.894	0.52	0.984	0.121	**0.338**	0.23	0.274	0.048	0.117	0.144	**0.132**	0.066
	Full 75	0.317	0.824	0.974	0.538	0.781	0.886	0.499	0.986	0.168	0.353	0.188	**0.23**	0.067	0.085	0.161	0.172	0.074
	Full 100	0.368	0.833	0.979	0.708	0.838	0.949	0.536	0.991	0.218	0.384	0.321	0.318	0.083	0.198	0.157	0.24	0.129
	Draw 25	0.301	0.808	0.976	0.625	0.798	0.919	0.385	0.984	0.211	0.361	0.287	0.329	0.082	0.205	0.147	0.336	0.072
MA	Draw 50	0.295	0.803	0.975	0.54	0.735	0.927	0.404	0.984	0.235	0.349	0.303	0.296	0.073	0.204	0.156	0.287	0.074
	Draw 75	0.292	0.797	0.975	0.567	0.73	0.937	0.426	0.988	0.274	0.364	0.257	0.329	0.06	0.171	0.124	0.318	0.074
	Draw 100	0.306	0.866	0.986	0.814	0.751	0.961	0.489	0.991	0.315	0.408	0.393	0.455	0.113	0.255	0.24	0.257	0.093
	Full 25	0.304	0.847	0.974	0.685	0.767	0.936	0.498	0.985	0.282	0.358	0.295	0.384	0.067	0.262	0.188	0.271	0.086
	Full 50	0.315	0.872	0.981	0.656	0.763	0.936	0.421	0.988	0.349	0.369	0.356	0.452	0.072	0.302	0.271	0.289	0.098
	Full 75	0.317	0.902	0.984	0.626	0.78	0.95	0.474	0.986	0.326	0.394	0.397	0.436	0.097	0.351	0.239	0.357	0.13
	Full 100	0.326	0.903	0.987	0.759	0.81	0.953	0.496	0.989	0.428	0.423	0.507	0.449	0.158	0.383	0.268	0.236	0.168

Table 4: Average fitness of each configuration across all data sets.

		Keij1	Keij11	Keij12	Keij13	Keij14	Keij15	Keij4	Keij5	Nguy10	Nguy12	Nguy3	Nguy4	Nguy5	Nguy6	Nguy7	Nguy9	Sext
	Draw 25	4	3	4	4	4	1	3	4	4	1	1	4	3	1	4	4	3
$\bar{M}A$	Draw 50	3	1	3	3	2	2	4	1	2	3	2	3	2	2	1	1	4
	Draw 75	1	4	1	1	3	3	2	3	1	2	3	1	4	3	3	3	1
	Draw 100	2	2	2	2	1	4	1	2	3	4	4	2	1	4	2	2	2
	Full 25	2	1	1	4	2	1	4	2	1	1	1	1	1	1	3	1	1
	Full 50	3	4	3	1	1	2	2	3	2	2	3	3	3	2	2	3	2
	Full 75	4	2	2	3	3	3	3	4	4	4	2	2	2	3	4	4	3
	Full 100	1	3	4	2	4	4	1	1	3	3	4	4	4	4	1	2	4
	Draw 25	2	3	2	3	1	3	4	3	2	2	1	2	3	2	2	2	1
$M\bar{A}$	Draw 50	1	1	3	1	2	2	1	1	1	3	3	3	2	1	3	1	2
	Draw 75	3	2	1	2	3	1	2	2	3	1	2	1	1	3	1	3	3
	Draw 100	4	4	4	4	4	4	3	4	4	4	4	4	4	4	4	4	4
	Full 25	1	4	2	2	2	1	2	3	2	2	2	3	3	1	1	3	3
	Full 50	2	3	1	1	3	3	3	1	1	1	3	2	1	3	2	1	1
	Full 75	3	1	3	3	1	2	1	2	3	3	1	1	2	2	4	2	2
	Full 100	4	2	4	4	4	4	4	4	4	4	4	4	4	4	3	4	4
	Draw 25	3	3	3	3	4	1	1	2	1	2	2	2	3	3	2	4	1
MA	Draw 50	2	2	1	1	2	2	2	1	2	1	3	1	2	2	3	2	3
	Draw 75	1	1	2	2	1	3	3	3	3	3	1	3	1	1	1	3	2
	Draw 100	4	4	4	4	3	4	4	4	4	4	4	4	4	4	4	1	4
	Full 25	1	1	1	3	2	1	4	1	1	1	1	1	1	1	1	2	1
	Full 50	2	2	2	2	1	2	1	3	3	2	2	4	2	2	4	3	2
	Full 75	3	3	3	1	3	3	2	2	2	3	3	2	3	3	2	4	3
	Full 100	4	4	4	4	4	4	3	4	4	4	4	3	4	4	3	1	4

Table 5: Rank based program error for best of run programs.

importance of a choice of modeling algorithm. We tried two algorithms for decision trees: REPT and CART. Neither of the algorithms produced significantly better results across all the 17 benchmarks. For some benchmarks there results were significantly different but, again, neither algorithms was consistently superior. The second investigation explored choosing different sets of subprograms for modeling. Rather than use all the subprograms of one program, it mixed subprograms across a subset of programs from the entire population. Our question was whether identifying useful subprograms in this way and integrating them into selection (via integration of model fitness for a program) and/or genetic variation (via archive based crossover) would yield superior error for the best GP model of the run. Again, we found our results to be equivocal. None of the configurations emerged consistently superior. Again, however ranking and error differed among benchmarks.

This work brings to light particular paths that to extend the concepts and understanding of BGP. Below we detail several avenues to explore.

- 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.
- Using a very different algorithm, one that also provides feature selection would be an interesting comparison to using REPT. Would the stronger dissimilarity between the modeling bias of LASSO have significant impact?
- 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.
- As is mentioned in Section **??** 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.

REFERENCES

[1] David Basin, Yves Deville, Pierre Flener, Andreas Hamfelt, and Jørgen Fischer Nilsson. 2004. Synthesis of programs in computational logic. In *PROGRAM DEVELOPMENT IN COMPUTATIONAL LOGIC*. Springer, 30–65.

[2] Robyn Ffrancon and Marc Schoenauer. 2015. Memetic Semantic Genetic Programming. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO '15)*. ACM, New York, NY, USA, 1023–1030.

[3] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, and others. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature* 538, 7626 (2016), 471–476.

[4] Sumit Gulwani. 2010. Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*. ACM, 13–24.

[5] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.* 11, 1 (Nov. 2009), 10–18. DOI:http://dx.doi.org/10.1145/1656274.1656278

[6] Thomas Haynes. 1997. On-line adaptation of search via knowledge reuse. *Genetic Programming* (1997), 156–161.

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

[8] Krzysztof Krawiec. 2015. *Behavioral Program Synthesis with Genetic Programming*. Studies in Computational Intelligence, Vol. 618. Springer International Publishing. DOI:http://dx.doi.org/doi:10.1007/978-3-319-27565-9

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

[10] Krzysztof Krawiec and Jerry Swan. 2013. Pattern-guided genetic programming. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. ACM, 949–956.

[11] Paweł Liskowski and Krzysztof Krawiec. 2016. Online discovery of search objectives for test-based problems. *Evolutionary computation* (2016).

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

[13] Robert I McKay. 2000. Fitness sharing in genetic programming. In *Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation*. Morgan Kaufmann Publishers Inc., 435–442.

[14] Nicholas Freitag McPhee, Brian Ohs, and Tyler Hutchison. 2008. Semantic building blocks in genetic programming. In *European Conference on Genetic Programming*. Springer, 134–145.

[15] Martin C Rinard. 2012. Example-driven program synthesis for end-user programming: technical perspective. *Commun. ACM* 55, 8 (2012), 96–96.

[16] Armando Solar-Lezama. 2008. Program synthesis by sketching. (2008).

[17] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. 2010. From program verification to program synthesis. In *ACM Sigplan Notices*, Vol. 45. ACM, 313–326.

[18] Leonardo Vanneschi, Mauro Castelli, and Sara Silva. 2014. A survey of semantic methods in genetic programming. *Genetic Programming and Evolvable Machines* 15, 2 (2014), 195–214.

[19] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. 2010. Automatic program repair with evolutionary computation. *Commun. ACM* 53, 5 (2010), 109–116.