Exploiting Subprograms in Genetic Programming

May 1, 2017

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 that 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 that one algorithm results in superior error reduction than the other. We also investigate alternative 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. This method combines behavioral traces from multiple programs and uses the information derived from modeling the combined program traces.

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[19, 15, 4]. Behavioral genetic programming (BGP) is an extension to GP that advances toward this compelling goal[9, 8]. The intuition of the BGP paradigm is that during evolutionary search and optimization, we can identify 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 the 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 each 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. If the model has feature selection capability, it

also reveals specific subprograms within the tree that are partially contributing to the program's fitness. BGP uses this information in two ways. It integrates *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 [9, 8].

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 sensitive BGP is 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 biases. Our investigation contrasts feature identification and model accuracy under the two implementations.

2. **"Plays well with others?"**: **Alternate ways to identify useful subprograms**. In BGP, the trace matrix is calculated for each program in the population. 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 children 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, the BGP process should explore recombinations of subprograms that *work well with other subprograms and programs in the population.* We examine this idea by concatenating program traces from a set of programs, not solely one program. We then feed the concatenated trace into a model regression. This demands a new measure of fitness to reflect how many subprograms each program contributes to the resulting model. This new measure of fitness can be integrated into the program's fitness. We examine concatenation of the entire population and sub-populations that are selected based on fitness.

Our specific demonstration focus herein is symbolic regression. We choose SR because it remains a challenge and has 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]. Alternative approaches to evolutionary algorithms include sketching[16], i.e. communicating insight through a partial program, generalized program verification[17], as well as hybrid computing with neural network and external memory[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 [10] 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 a 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. For each program in the population, capture the behavior of each subprograms 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 in 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 e and complexity c into program fitness.

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 in the same manner that we derive a model from REPT. We denote the model derived for CART by M_S and contrast it with the model derived from REPT, which we now denote by M_R.

3.3 Identifying Useful Subprograms

Next we realize alternative implementations of the BGP strategy. We do this to investigate alternative ways of identifying useful subprograms, considering that prior work only considers models that are trained on the trace of a single program. 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). We proceed with step (3), but it is important to note that the weights given to the subprograms considered for the archive are identical, because only a single model is built. Step (4) is altered to incorporate model contribution in place of model error and complexity. The model is built from the features of many programs, so the model error and model complexity for each individual program are undefined. Model contribution measures how many features each program contributes to M. For this we use w_c, which is given by Equation 2, where p' is the number of features in M from program p, and $|U(M)|$ is the total number of features from T_c used in M. This method 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.

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

An alternative implementation, that we name **DRAW**, is to draw random subsets from T_c, and build a model on each. This would possibly contribute to a more robust archive, if it can be populated with subtrees that are frequently selected by the machine learning model. We modify Step (3) to account for the possibility that a specific subtree was chosen to be in more than one model. In this case, the subtree's merit w is set to be the sum of the assigned merit values each time the subtree is chosen. Step (4) is modified in the same way as it is in **FULL**.

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

Implementation details of **FULL** and of **DRAW** are provided in 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 evaluates the performance of **FULL** and **DRAW** for different configurations then compares **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**[2]: $\{+, -, *, /, \log, \exp, \sin, \cos, -x\}$
- **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

We use the same four program fitness functions used in [9] (in addition to model contribution fitness which is described in Section 3.3). Program error f, is given by Equation 3, 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. Program size s, is given by Equation 4, where $|p|$ is the number of nodes in the tree that defines the program. Model error e, is given by Equation 5, where M is the output of the machine learning model when it is evaluated on the trace of the program. Model complexity c, is given by Equation 6, where $|M|$ is the size of the model.

$$f = 1 - \frac{1}{1 + d_m(\hat{y}, y)} \tag{3}$$

$$s = 1 - \frac{1}{|p|} \tag{4}$$

$$e = 1 - \frac{1}{1 + d_m(M, y)} \tag{5}$$

$$c = 1 - \frac{1}{|M|} \tag{6}$$

[2]Note that for our implementation of /, if the denominator is less than 10^{-6} we return 1, and for our implementation of log, if the argument is less than 10^{-6} we return 0.

First we use the 3 BGP algorithm configurations that use REPT to replicate [9]'s work on the symbolic regression benchmarks. These we call BP2A, BP4, BP4A following precedent. In the name the digit 2 indicates that model error e and complexity c 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. When the A is omitted ordinary crossover is used. We observe results consistent with the prior work. Our open source software is available on Github. This allowed us to proceed to evaluate feature selection sensitivity to the modeling algorithm.

It is important to note, that for each configuration we report regression, i.e. training set performance. We are primarily interested in exploring subprogram behavior and how to assemble subprograms. Reporting generalization would complicate the discussion without materially affecting our conclusions.

4.2 Sensitivity to Model Bias

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

	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.

Table 1 shows the results of running the 3 different configurations (BP2A, BP4, BP4A) each with the two decision tree algorithms. Averaging over the rankings across each benchmark we find that BP2A using REPT is best. For BP2A, REPT outranks CART but when model error is integrated into the program fitness, (i.e. BP4A and BP4) regardless of whether or not an archive is used, CART is superior to REPT.

When we compare the results of using the archive while model error is integrated into the program fitness (i.e. BP4A to BP4), for both REPT and CART it is better to use an archive than to forgo one. Comparing BP2A with BP4A, we can measure the impact of model error and complexity integration. We find that for both CART and REPT it is not helpful to integrate model error and complexity into program fitness.

For a deeper dive, at the specific benchmark level, 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 in Figure 1. 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. No discernible difference arose among this sort of plot. This is understandable given the stochastic nature of BGP.

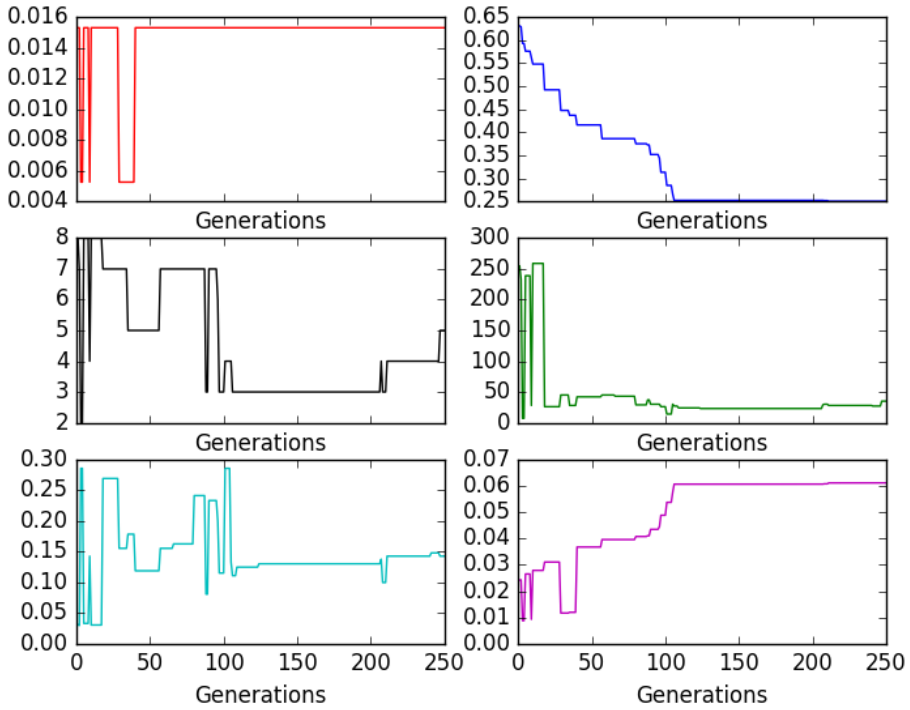

Figure 1: We take one run of Keijzer1 running with REPT modeling and configuration BP4. We plot on the first row model error on the left and the fitness of the best program (right). The plots on the second row show number of features of model and number of subprograms in the best program (right). The 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.

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

In this section, we compare various configurations of **FULL** and **DRAW**. For the algorithm configurations of this section, we adopt a clearer notation. We drop the BGP prefix and use M to denote when program contribution is integrated into program fitness, and $\hat{M}$ to denote when it is not. We use A to denote when subprograms are qualified for archive insertion and archive retrieval during BGP crossover, and $\hat{A}$ to denote when ordinary crossover is used.

More details of the **DRAW** method are appropriate. Referencing [9] we analyze the formula for computing the weight of a given subtree (see Equation 1). We note that the $|U(p)|$ factor in its denominator indirectly increases the weight of smaller subprograms. This occurs because smaller programs yield

smaller models (i.e. smaller $|U(p)|$), and smaller programs have smaller subprograms. Therefore we designed **DRAW** to also favor the archiving of smaller subprograms. **DRAW** proceeds as follows:

1. The population is sorted best to worst by program fitness (program error and size) using the NSGA pareto front crowding calculation because BGP is multi-objective.

2. The sorted population is cut off from below at a threshold $\lambda\%$ to form C. The trace matrixes of every program in C are concatenated to form T_C which we call the subprogram pool.

3. We next sort the population by **size** and select the smallest 20% forming a size sample we call K.

4. Finally we draw from K at random to obtain the number of subprograms that will be collectively modeled. 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. This generates multiple smaller collections of diverse subprograms.

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

We first asked what if C is composed of **every** subprogram in the population, i.e. $|C| = PopSize$? While this C using **FULL** would only support one model being derived, it would give all subprograms in the population an opportunity to be used with each other in the model as features. Similarly, by favoring many smaller combinations drawn from all subprograms, **DRAW** would, through repetition, give all subprograms in the population an opportunity to be used with some of the all the others. If we compare the result of **DRAW** and **FULL** we can gauge the difference between generating many more small models vs one bigger model, when every subprogram in the population is "eligible" to be selected as a model feature. This comparison is detailed on the bottom line of Table 3. The leftmost averaged ranking results (by average fitness, across the 17 benchmarks) for different model and archive options are from **DRAW** and the rightmost are from **FULL**. The data reveals that using all the subprograms, with either **FULL** or **DRAW** is NOT advantageous. Further empirical investigation to understand this result should consider two issues: 1. the program size to fitness distribution of the population each generation could be leading to very large number of subprograms and 2. the modeling algorithm (REPT) may be overwhelmed, in the case of **FULL**, by the number of features, given the much smaller number of training cases for the regression.

Next we can consider the rankings of each configuration across different selections for the subprogram pool C. When $\lambda = 25$ the model feature options are from the highest fitness tier of the population. In 4 of 6 cases, this appears to *impede* the error of the best of run program, as measured by average ranking. In 4 of 6 cases, including all 3 of **DRAW**, sizing the subprogram pool to be slightly less elitist ($\lambda = 50$ or $\lambda = 75$) was better. But extending λ to 100 appears to be too diverse. Table 4 and Table 5 provide more detailed average fitness and ranking information, i.e. results for each individual benchmark.

Finally, we compare these configurations to the three original BGP configurations. We find that the best performing method is highly dependent on the specific benchmark, and that overall none of the configurations is shown to be the clear winner.

5 Conclusions and Future Work

The paper's primary contributions are to explore two subprogram value driven questions. The first question addressed the importance of a choice of modeling algorithm. The modeling algorithm can impact selective pressure (because model fitness can be integrated back into program fitness) and genetic variation (because subprograms used by a model can be inserted into the BGP archive and used in BGP crossover). 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 the average fitness results were significantly different but, again, neither of algorithms was consistently superior in each case. Using a completely different modeling technique, i.e. one different from decision trees altogether, that also provides feature selection would be an interesting comparison to using REPT. All feature selection algorithms are stochastic so their results vary. Perhaps the stochasticity of any algorithm overwhelms the impact of a particular technique's bias. We next will consider whether the stronger dissimilarity between the modeling bias of LASSO and decision trees has significant impact. LASSO is a linear technique and has a regularization pressure parameter making it an interesting option.

Our 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 of run program. 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 extend the concepts and understanding of BGP. There are several avenues that could be explored: 1. 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.

2. 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.

3. In BGP 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.

Acknowledgments

Suppressed to honor the author blind review process.

References

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

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

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

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

[5] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

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

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

[8] Krzysztof Krawiec. *Behavioral Program Synthesis with Genetic Programming*, volume 618 of *Studies in Computational Intelligence*. Springer International Publishing, 2015.

[9] 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.

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

[11] Paweł Liskowski and Krzysztof Krawiec. 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, et al. Genetic programming needs better benchmarks. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 791–798. ACM, 2012.

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

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

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

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

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

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

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

		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: Comparison of different decision tree algorithms: REPT and CART on average program error for best of run programs (averaged across 30 runs). N.B. program error does NOT include program size. During evolution the fitness of a program integrates program error and size per[9]

C	$\hat{M}A$	$M\hat{A}$	MA	$\hat{M}A$	$M\hat{A}$	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
$\hat{M}A$	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
	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
$M\hat{A}$	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
	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
MA	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
	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: Sampling subprograms for modeling across the population, not from one program. Two methods **DRAW** and **FULL** were evaluated. Data shows average fitness of each algorithm configuration across all benchmarks.*

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