Behavioral Programming: A Broader and More Detailed Take on Semantic GP

Krzysztof Krawiec*
Institute of Computing Science
Poznan University of Technology
60965 Poznan, Poland
krawiec@cs.put.poznan.pl

Una-May O'Reilly
CSAIL
Massachusetts Institute of Technology
Cambridge, MA
unamay@csail.mit.edu

ABSTRACT

In evolutionary computation, the fitness of a candidate solution conveys sparse feedback. Yet in many cases, candidate solutions can potentially yield more information. In genetic programming (GP), one can easily examine program behavior on particular fitness cases or at intermediate execution states. However, how to exploit it to effectively guide the search remains unclear. In this study we apply machine learning algorithms to features describing the intermediate behavior of the executed program. We then drive the standard evolutionary search with additional objectives reflecting this intermediate behavior. The machine learning functions *independent of task-specific knowledge* and discovers potentially useful components of solutions (subprograms), which we preserve in an archive and use as building blocks when composing new candidate solutions. In an experimental assessment on a suite of benchmarks, the proposed approach proves more capable of finding optimal and/or well-performing solutions than control methods.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program synthesis*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*

Keywords

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

1. MOTIVATION

Most of the algorithms developed in the field of evolutionary computation (EC) obstinately insist on using an objective function directly associated with the given task as their central, if not the only, gauge of candidate solutions. If no additional performance information is available (i.e., the objective function is a black-box), and there is no access into the internal structure of candidate solutions, this is indeed necessary. However, in many domains there is much more information available about the problem solving nature of a candidate solution as well as access to its internal structure. This is particularly true in genetic programming (GP), where the objective function is typically an aggregate calculated over a set of fitness cases while the candidate solutions are programs that are both syntactically and semantically decomposable. As a consequence, nothing precludes scrutinizing program output for particular fitness cases, or examining program behavior on the level of particular program components (subprograms). In such cases, relying exclusively on the original objective function is a mere artifact of researchers' habits and conventions, rather than necessity.

This habit may be crippling because one cannot expect difficult learning and optimization problems to be efficiently solved by heuristic algorithms that are driven by a scalar objective function which provides low-information feedback. For example, consider the domain of Boolean function synthesis, a common benchmark suite for GP. Given a programming language with k binary instructions and m input variables, there are programs composed of n instructions, where $cat(n)$ is the nth Catalan number. Thus, even for the relatively simple multiplexer-11 benchmark ($m = 11$) and instruction set of $k = 4$ binary instructions, the number of 'minimal' programs that can *potentially* solve this problem (i.e., fetching every input variable only once) is a staggering 2.93×10^{11}. To believe that *any* heuristic search algorithm can efficiently traverse such a big space using an objective function with a range of only $2^{11} = 2048$ values (and thus conveys only 11 bits of information) is very optimistic, if not naive. The problem is general. Symbolic regression, despite having a continous objective function, which is arguably more fine grained, also falls victim. By shaving off diminishing fractions of error, evolution often chases the noise in data and ends up overfitting to it.

This problem becomes even more grave when a search quickly converges to solutions of high quality, all of which have very similar or even identical fitness. The selective pressure ceases to be able to differentiate them, the search gradient is lost, and random search becomes the only way to improve. This partially explains why, though attempts have been made to address the above problem from a few angles and GP is capable of solving some complex problems

*Research conducted during K. Krawiec's stay at the ALFA Group, Computer Science and Artificial Intelligence Laboratory (CSAIL), MIT.

GECCO'14, July 12–16, 2014, Vancouver, BC, Canada.
Copyright 2014 ACM 978-1-4503-2662-9/14/07 ...$15.00.
http://dx.doi.org/10.1145/2576768.2598288 .

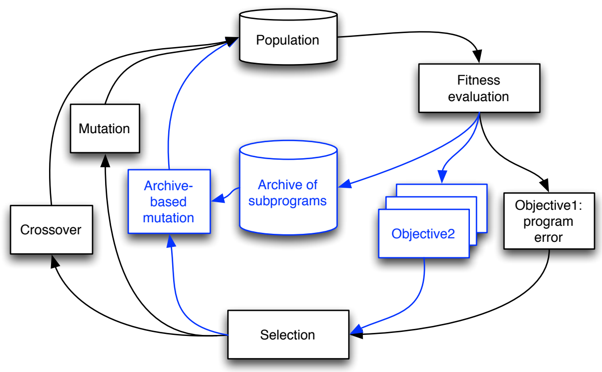

Figure 1: Conventional genetic programming workflow (in black) and the components added by behavioral programming (in blue).

at a reasonable computational expense, it does not scale well with problem size.

From a wider evolutionary perspective, the reliance on a task-oriented fitness function alone is coherent with *gradualism*, the cornerstone of classical Darwinism. However, even if we assumed that gradualism drives biological evolution (though it has been questioned many times), we posit that it is *not necessarily most appropriate for problem solving* (in a broad meaning of this term, i.e., including search, optimization, and learning). Humans rarely solve problems gradually, case by case. Instead, they inductively and deductively reason to be able to identify regularities and perform conceptual leaps. They need to reason about how they achieved their goals, not simply whether or not they achieved them.

Our general claim is thus that we should seek *search drivers* based on the general problem solving ability *of intermediate behavior* in order to make learning and optimization metaheuristics more 'intelligent' and, in effect, possibly more efficient. We therefore propose *behavioral programming*, which conceptually fits under the umbrella of semantic genetic programming. It consults both the semantics of GP outputs (like existing semantic GP variants) and intermediate behavior, with the help of task-agnostic machine learning. In this paper, we demonstrate the underlying conceptual framework of behavioral programming by embedding an implementation of it within conventional tree-based genetic programming (GP) in a system we call BPGP. BPGP retrieves detailed, intra-execution information on program behavior and uses it to (i) 'multiobjectivize' [8] the search space, (ii) identify the potentially valuable pieces of code, and (iii) reuse them later in search using an archive. Our experimental analysis shows that this 'better-informed' search algorithm attains radically better performance when compared to methods driven by conventional objective functions alone.

2. BEHAVIORAL PROGRAMMING

Behavioral programming extends the workflow of conventional GP adding three components (see Fig. 1): behavioral evaluation, archiving of useful subprograms, and an archive-based search operator. The former two execute in the fitness evaluation phase, while the archive-based search operator is used in the breeding phase (possibly along the conventional search operators like mutation or crossover). We detail these components in the following sections.

2.1 Behavioral evaluation

Behavioral evaluation, preliminary version of which has been proposed under the name of PANGEA (Pattern-Guided Evolutionary Algorithm) in [9], can be conveniently presented as an extension of conventional fitness evaluation.

In standard tree-based GP, the fitness function iterates over a set of fitness cases of the form (x, y), where x is *program input* (typically one or more input variable), and y is the *desired output* for x. The input x is fed into the program p (i.e., made available to its leaves), and p is executed, which consists of traversing p's tree nodes in an inorder sequence, and propagating the partial results upwards in the tree. Upon completion, execution produces *program output* $\hat{y}$. This process, repeated over all tests, results in a vector of program outputs $\hat{\mathbf{y}}$, which is subsequently compared to the vector of desired outputs $\mathbf{y}$ (the *target*). This is typically done using some form of metric $|\,|$, so that p ultimately receives fitness

$$f(p) = |\mathbf{y}, \hat{\mathbf{y}}|. \tag{1}$$

As evolution proceeds, programs represented as trees often experience excessive growth (a.k.a. bloat), which increases evaluation cost and makes the search process less effective. In response to that, program size is taken into account. In the simplest case, this boils down to calculating program length (size):

$$s(p) = |p| \tag{2}$$

where $|p|$ is simply the number of tree nodes. This measure can be used to make decisions at various stages of the evolutionary loop, e.g., for resolving ties on f in tournament selection, or as another objective. In the latter case, the program is characterized by a *multiobjective fitness*

$$(f(p), s(p)) \tag{3}$$

where both objectives are minimized.

Behavioral evaluation does not affect the above process (and thus has no impact on $f(p)$ nor $s(p)$), but rather works *with* it. Prior to running the program on a given input x, an empty list l is created. Then, as execution traverses the program tree, the output produced by every visited program node is appended to l. When the program terminates, the resulting list gathers all intermediate outcomes of execution and so forms the *trace* of program p for a given test. If the considered programs are expressions, free of conditional statements and loops (as we assume in this paper), the nodes of program tree are visited in the same order for every fitness case, and the lists l have the same length n for every fitness case. Let l_i denote the trace obtained for the ith test. The traces resulting from evaluation on particular fitness cases are gathered in a *trace table* of the following structure:

$$\begin{array}{ccccc} l_1[1] & l_1[2] & \ldots & l_1[n] & y_1 \\ l_2[1] & l_2[2] & \ldots & l_2[n] & y_2 \\ \vdots & \vdots & & \vdots & \vdots \\ l_t[1] & l_t[2] & \ldots & l_t[n] & y_t \end{array} \tag{4}$$

where $l_i[j]$ denotes the jth element of a list for the ith test, and t is the number of tests.

The trace table is subsequently treated as inputs that can be passed to a machine learning algorithm, where the last column plays the role of the label (dependent variable), while the preceding columns are *features* (independent variables).

If the original desired output y is categorical, y_i's are *class labels* and the task is *classification*; if y is continuous, it is a *response variable* and the task is *regression*.

In either case,the learning algorithm is trained on the data of the trace table (4), which results in a *model* $M(p)$ of the data, which indirectly reveals a *behavioral description* of the program's intermediate execution. This description can be characterized by two measures: *error* $e(M(p))$ on the training data (goodness of fit) and *complexity* $c(M(p))$ denoting M's complexity.

The technical definitions of these measures may vary depending on domain and application, but there is a central motivation for adopting them as additional program characteristics. If a program produces output $\hat{\mathbf{y}}$ that is far from the desired output $\mathbf{y}$ (which implies an inferior score on f), or is excessively large (inferior score on s), it may still host subprograms that can turn out to be useful when, e.g., rearranged/composed in a different way, and/or used in other programs. The machine learning process is intended to discover such subprograms among the population by analyzing the trace data and using behavioral features (columns in table (4)) that relate to the desired output of the program (the dependent variable of the learning task (4)). If a program exhibits such features, the resulting model M is likely to achieve low error e. However, sometimes a model may attain low error even without particularly good features at hand, i.e., by overfitting to the training data or even by rote learning in an extreme case. This is where model complexity c comes into picture: its role is to promote succinct models in the hope of evolving general subprograms..

The tuple gathering all characteristics derived from p, i.e.,

$$(f(p), s(p), c(M(p)), e(M(p)) \tag{5}$$

is the final result of behavioral evaluation. By analogy to bi-objective evaluation (Eq. (3)), the elements of the above tuple are treated as search objectives, and reported to the multiobjective evolutionary search process. Other ways of exploiting this information are conceivable, but remain future work.

2.2 Archiving useful subprograms

Behavioral evaluation results in two measures of the behavioral model $M(p)$ derived from the evaluated program p, which provide more information on p (compared to the conventional single- or two-objective evaluation). However, those measures are still only scalar values, while the model itself captures even more information about program behavior and the prospective utility of its components.

It is thus natural to seek means of directly exploiting the information contained in the behavioral model, rather than only characterize it quantitatively. For this we assume that the model can explicitly reveal *which* trace features it uses to make predictions (of all the features gathered from all program tree nodes). For models represented as classification (or regression) trees, which we employ in this paper, we retrieve that information by traversing the decision tree and gathering the features used by decision nodes. However, model representation is largely irrelevant, and the fact that we deal with program *trees* and examine models represented as decision/regression *trees* is incidental (and those two should not be confused). For many other 'white-box' (i.e., more or less symbolic) algorithms (e.g., decision rules, Bayesian nets) such information is equally available, and even the 'blackbox' representations (like linear regression models or artificial neural networks) do not preclude such possibility.

As the features used by a model correspond one-to-one to subprograms in the evaluated tree, the knowledge of whether they are included in the model immediately allows us to pinpoint the *useful subprograms* in the program tree and collect them into a set $U(p)$. Subsequently, we gather these useful subprograms of all individuals in population in a global *archive* A, a prioritized queue of a fixed length, maintained throughout the entire evolutionary run. The priority is based on a measure we associate with a useful subprogram, p' called its *utility,* u:

$$u(p') = \frac{1}{(1 + e(p))|U(p)|} \tag{6}$$

Useful subprograms are added to the archive until it reaches capacity. At that point the archive is reset and re-populated by utility-proportional selection. Subprogram utility helps promote subprograms that contribute to good models (low error $e(p)$) and from within compact models (low $U(p)$).

2.3 Reusing useful subprograms: archive-supplied mutation

The ultimate goal of archiving useful subprograms is their reuse in new candidate solutions. For compatibility with the EC conceptual toolkit, we implement this component as a mutation-like search operator.

Given a parent program tree p, the operator picks at random a node in p and replaces it (and the subtree rooted in it) with a subprogram drawn at random from the archive A. Here, the subprograms are selected from archive in exactly the same manner that determines their survival in the archive (Section 2.3), i.e., proportional to utility (Eq. 6). Therefore, subprograms of higher utility are more likely to become components of other programs.

This operator, though applied to a single parent program and in this sense mutation-like, can be considered as a form of crossover, because it outfits the offspring with a piece of code that has been previously retrieved from another individual (in the current or previous generations). As a result, the archive-supplied mutation cannot use arbitrary subprograms, in particular cannot use instructions other than those currently present in archive. For this reason, we use conventional mutation side-by-side with this operator (Section 4).

3. RELATED WORK

The closest counterpart to BPGP or behavioral programming in GP literature is semantic GP, where, rather than assessing program quality aggregated over all tests (i.e., conventional fitness), program output is scrutinized for every test individually. This avenue of GP research was initiated by McPhee *et al.*, who studied the impact of crossover on program semantics and semantic building blocks [14], although earlier work on implicit fitness sharing paved its way [13]. The more contemporary studies [15, 16] reason about the geometry of semantic space. Behavioral programming can be seen as semantic GP pushed one level of detail deeper, i.e., not limited to program output, but to program behavior over its entire execution. Also, from start to finish in terms of behavior.

Other past works employ alternative search objectives, even if not explicitly framed in that way. The concept of multiobjectivization by Knowles *et al.* [8] was an early forerunner of the idea that additional objectives can, paradoxically, sometimes make the problem easier. However, Knowles *et al* and similar studies assumed that the additional objectives would be proposed by the experimenter because then task-specific sub-goals could be expressed. In contrast, behavioral programming discovers subsolutions autonomously (i.e., it induces behavioral models from program traces without explicitly formulating what is the desired intermediate behavior of a program). Behavioral programming can be seen thus as an intermediate approach between conventional single-objective search and novelty search [10], where the task-driven objective is discarded altogether.

With respect to the use of archive, behavioral programming can be likened to methods that maintain repositories of code pieces. Automatically defined functions can be seen as such repositories of subprograms, albeit only local (i.e. attached to an individual) and devoid of any directed maintenance (i.e., the content of the repository is controlled only by evolution). A more similar approach is implemented in *run transferable libraries* [18] that collect program fragments throughout a GP run and reuse them in separate evolutionary runs applied to other problems. Rosca and Ballard [17] create and use an analogous library within a single evolutionary run, with sophisticated mechanism for assessing subroutine utility, and entropy for deciding when a new subroutine should be created. Haynes [4] integrated a distributed search of genetic programming-based systems with 'collective memory', albeit only for redundancy detection. Other approaches involving some form of library include reuse of assemblies of parts within the same individual [6] and explicit expert-driven task decomposition using layered learning [1]. None of these methods however analyze program behavior to make a decision about which subprograms should be archived, and only some of them associate utility with a subprogram.

4. EXPERIMENTAL ANALYSIS

Behavioral programming involves behavioral evaluation (Section 2.1) and manipulation of subprograms (Sections 2.2 and 2.3). From practical perspective, it is worth asking whether and how these components affect the efficiency of evolutionary search, and if there is any synergy between them. To answer these questions, we conduct a thorough cross-domain comparative assessment of various configurations of BPGP, using certain configurations of conventional GP as baselines.

4.1 The problems

Table 1 presents the 35 benchmark problems used in our experiment, which come from three domains we characterize by different data types: Boolean (8 benchmarks), categorical (10 benchmarks), and regression (17 benchmarks). Table 1 summarizes the problems, listing the instruction set used for each, and for every problem, the number of variables, tests, and the cardinality of the search space (where countable). Note that none of the instruction sets contains constants.

The targets for particular **Boolean problems** are defined as follows. For an v-bit comparator *Cmp* v, a program is required to return *true* if the $\frac{v}{2}$ least significant input bits encode a number that is smaller than the number represented by the $\frac{v}{2}$ most significant bits. In case of the majority *Maj* v problems, *true* should be returned if more that half of the input variables are *true*. For the multiplexer *Mul* v, the state of the addressed input should be returned (6-bit multiplexer uses two inputs to address the remaining four inputs, 11-bit multiplexer uses three inputs to address the remaining eight inputs). In the parity *Par* v problems, *true* should be returned only for an odd number of *true* inputs.

The **categorical problems** come from Spector *et al.*'s work on evolving algebraic terms [20] and dwell in the ternary domain: the admissible values of program inputs and outputs are $\{0, 1, 2\}$. The peculiarity of these problems consists of using only one binary instruction in the programming language, which defines the underlying algebra. For instance, for the a_1 algebra, the semantics of that instruction (a1 in Table 1) is defined as follows:

a_1	0	1	2
0	2	1	2
1	1	0	0
2	0	0	1

For brevity, we refer the reader to [20] for the definition of the remaining algebra problems.

For each of the five algebras considered here, we consider two tasks (of four discussed in [20]). In *discriminator term* tasks (*D*-* in Table 1), the goal is to synthesize an expression (using only the one given instruction) that accepts three inputs x, y, z and is semantically equivalent to

$$t^A(x, y, z) = \begin{cases} x & if\, x \neq y \\ z & if\, x = y \end{cases} \tag{7}$$

Given three inputs and ternary domain, this gives rise to $3^3 = 27$ fitness cases for these benchmarks.

The second task defined for each of algebras (*M*-* in Table 1), consists in evolving a so-called *Mal'cev term*, i.e., a ternary term that satisfies

$$m(x, x, y) = m(y, x, x) = y \tag{8}$$

This condition specifies the desired program behavior only for the indicated combinations of inputs. In other words, the desired value for $m(x, y, z)$, where x, y, and z are all distinct, is not determined. As a result, there are only 15 fitness cases in our Mal'cev tasks, the lowest of all considered benchmarks.

The **regression problems** considered here come from [12] and include both univariate and multivariate target functions. The univariate ones (*Keij1*, *Keij4*, *Nguy3..7* and *Sext*) use 20 tests uniformly distributed in the $[-1, 1]$ interval, except for the *Keij4* benchmark which uses the $[0, 10]$ interval. The remaining problems are predominantly bivariate, and involve $5 \times 5 = 25$ fitness cases uniformly distributed on the two-dimensional grid. The only exception is *Keij5*, which hosts three input variables, with $4 \times 4 \times 4 = 64$ fitness cases distributed equidistantly in the cube. For other details on these benchmarks, see [12][1].

4.2 Configurations

Conceptually, there are two main aspects in which BPGP differs from conventional GP: the behavioral evaluation and manipulation of subprograms. To determine how the presence of these components impacts search performance, we

[1]The original formulation of some of these benchmarks in [12] assumes random drawing of fitness cases from variables' domains. However, to assure reproducibility of results we employ deterministic equidistant grid.

Table 1: **The benchmarks. v – number of input variables, m– the number of tests, k – number of unique program semantics.**

	Instr. set	Problem	v	m	k
Boolean	and, nand, or, nor	Cmp6	6	64	2^{64}
		Cmp8	8	256	2^{256}
		Maj6	6	64	2^{64}
		Maj8	8	256	2^{256}
		Mux6	6	64	2^{64}
		Mux11	11	2,048	2^{2048}
		Par6	6	64	2^{64}
		Par8	8	256	2^{256}
Categorical	$a_1(x,y)$	D-a1	3	27	3^{27}
	$a_2(x,y)$	D-a2	3		
	$a_3(x,y)$	D-a3	3		
	$a_4(x,y)$	D-a4	3		
	$a_5(x,y)$	D-a5	3		
	$a_1(x,y)$	M-a1	3	15	3^{15}
	$a_2(x,y)$	M-a2	3		
	$a_3(x,y)$	M-a3	3		
	$a_4(x,y)$	M-a4	3		
	$a_5(x,y)$	M-a5	3		
Regression	$+, -, *,$ %, sin, cos, log, exp, $-x$	Keij1,Keij4	1	20	–
		Nguy3,Nguy4	1		
		Nguy5,Nguy6	1		
		Nguy7,Sext	1		
		Keij5, Keij11	2		
		Keij12, Keij13	2		
		Keij14, Nguy9	2		
		Nguy10, Nguy12	2		
		Keij15	3		

design 3 configurations. In BPGP4 and BPGP4A there are 4 objectives, (see Formula 5 and the latter configuration uses an archive). BP2A uses two objectives and an archive. GP1 is the standard single-objective GP. We employ generational GP with tournament selection. For single-objective configurations (GP1, GP1L), tournament selection with the conventional tournament size 7 is used. For multiobjective setups, e.g.GP2, we employ Non-Dominated Sorting Genetic Algorithm II (NSGA-II, [2]) at the selection stage, an *ipso facto* standard of multiobjective evolutionary optimization.

As BPGP can perform quite well with small population, we set population size to 100. However, this may be inconvenient for conventional GP, which is know to require larger populations. To make comparison fair, we consider additional configuration GP1L that uses population size 1000. Because the maximal number of generations is the same in all configurations, GP1L's computational budget (max. number of evaluations) is ten times greater than for the other configurations.

Search operators. We employ three search operators: (i) subtree-replacing mutation, present in all setups and engaged with probability 0.1, (ii) subtree-swapping crossover, and (iii) archive-supplied mutation (Section 2.3). To eliminate the interference of other factors, operators (ii) and (iii) proceed in the same way, except only for the source of subprograms. While the conventional subtree mutation plants a randomly generated subtree at the selected locus, the archive-based operator uses for that purpose a subprogram drawn at random from the archive, as described in Section 2.3.

Table 2: **Configurations. For search operators, the numbers denote the probability at which an operator is being engaged (blanks are zeros).**

	Setup	GP1	GP1L	GP2	BPGP4	BP2A	BPGP4A
Objectives	f	✓	✓	✓	✓	✓	✓
	s			✓	✓	✓	✓
	c				✓		✓
	e				✓		✓
Operators	Mutation	0.1	0.1	0.1	0.1	0.1	0.1
	Crossover	0.9	0.9	0.9	0.9		
	A-Mutation					0.9	0.9
	Pop. size	100	1000	100	100	100	100
	Tourn. size	7	7	4	4	4	4

Node selector. To lessen potential bloat, rather than using a conventional method for drawing mutation and regression loci in programs, we employ 'home brewed' *uniform depth node selector.* This selector calculates tree depth d, draws uniformly a random number d' from the interval $[0, d]$, and returns a randomly chosen loci at depth d' in the tree. In this way, the probability of node selection does not grow exponentially with node depth (as it is the case in the conventional methods). Mutations and crossovers close to the root node become more likely, and bloat is thus partially constrained. We employ this node selection method in all search operators. Preliminary experiments have shown that it is beneficial for standard GP.

Behavioral model. We chose REPTree [3] as the algorithm for deriving behavioral models, which we motivate twofold. Firstly, REPTree can induce both classification as regression trees, so it can handle problems from all three domains we consider in this paper without any adjustments or parametrization. Secondly, certain technical tricks make it remarkably faster than other decision tree inducers.

Though REPTree can post-prune (simplify) the induced decision trees, we disable this option, and make it always grow maximal trees (i.e., it always splits a decision node if any gain in entropy can be achieved). In this way, we expect the size of the induced classifier reflect the complexity of the underlying training data.

Archive. Archive capacity is set to 50. N.B. this is the only parameter of BPGP.

Objectives. NSGA-II resolves ties between solutions in the same tier of ranking by resorting to *sparsity* measure [2], which involves calculating the Euclidean distance between the points on the Pareto front. This requires the objectives to range in the same interval, so that particular objectives have on average the same impact on sparsity. To that aim, we define the objectives (Eq. 5) as follows:

		Boolean	*Categorical*	*Regression*
Program error	f	$d_h(\mathbf{y}, \hat{\mathbf{y}})/t$		$1 - \frac{1}{1+d_c(\mathbf{y},\hat{\mathbf{y}})}$
Program size	s	$1 - 1/\lvert p\rvert$		
Model complexity	c	$1 - 1/\lvert M\rvert$		
Model error	e	$d_h(M, \hat{\mathbf{y}})/t$		$1 - \frac{1}{1+d_c(M,\hat{\mathbf{y}})}$

where d_h and d_c denote respectively Hamming and city-block distance, t is the number of tests, $|p|$ denotes the length of program p (number of nodes in its tree), and$|M|$ is the number of nodes in REPTree M.

The objectives defined in this way are minimized and assume values from interval $[0, 1]$. We handle numerical imprecision in regression problems by assuming that $|y_i - \hat{y}_i| = 0$ whenever $|y_i - \hat{y}_i| < 10^{-15}$.

Termination. A run is terminated when an ideal solution is found ($f = 0$) or the maximum number of 250 generations has elapsed. We assume that the final outcome of an evolutionary run is the individual that attained the best fitness (lowest f) at any stage of the run. All data reported below characterize such individuals, averaged over 30 independent evolutionary runs for every setup and benchmark; a total of 30 ×35 benchmarks ×6 configurations = 6300 runs has been conducted.

Let us emphasize that, except for the elements of the setup that have to differ across domains because of their different natures, all benchmarks from all domains use the same parameter settings. For parameters not mentioned here we assume their default values in the ECJ library [11].

4.3 The results

Success rate. We first assume the strong-AI perspective and consider our benchmarks as *search problems*. In that case, the objective is to find an ideal solution to the problem, and any other outcome is considered a failure. The appropriate performance measure for this type of problems is success rate, i.e., the percentage of runs that produced an ideal individual.

Because of the large total number of benchmarks (35), we present the detailed results in a separate Table 3, while in the following we report and discuss the aggregated outcomes. We report the outcome of the Friedman's test for multiple achievements of multiple subjects [7]. Compared to ANOVA, it does not require the distributions of variables in question to be normal (which we cannot assume for most of the performance indicators considered here). It is then worth pointing out that the results presented below are very conservative in terms of statistical significance, i.e., many insignificant differences could become significant given larger pool of benchmarks.

Before passing to the pairwise comparisons, we report the average ranks of the methods (the lower, the better):

BPGP4A	GP1L	BP2A	BPGP4	GP1	GP2
2.43	3.10	3.36	3.43	**3.86**	**4.83**

The p-value for Friedman test is $\ll 0.001$, which strongly indicates that at least one method performs significantly different from the remaining ones. We conducted post-hoc analysis using symmetry test [5]: bold font above marks the methods that are outranked at 0.05 significance level by the *first* method in the ranking (BPGP4A in this case).

The BPGP4A variant ranks the best, outperforming conventional GP methods, including even the GP1L configuration, which has ten times greater number of evaluations at its disposal. For GP1L, the difference is statistically insignificant but the difference in ranking might signal significance if it holds over more benchmarks.

BPGP methods managed to solve (at least once in 30 evolutionary run) five problems that remained unsolved by GP algorithms. These are (see Table 3): D-a1, D-a4, D-a5, Keij4, and Par8. On the other hand, the only benchmark that they could not solve while GP could is Nguy5.

Error rate. In this section we treat the benchmarks as *optimization problems*. In that case, the objective is to minimize the error, and the appropriate performance measure is the error (f) of the best-of-run program.

We report the results for this performance measure in an analogous way to the previous section. The Friedman test is conclusive, and methods' ranks are as follows:

BPGP4A	GP1L	BP2A	BPGP4	GP1	GP2
2.17	2.21	**3.31**	**3.71**	**3.79**	**5.8**

Again, BPGP4A ranks the best, and this time even beats the BPGP configurations that miss one of the components: behavioral evaluation (BP2A) and subprogram archive (BPGP4).

Predictive accuracy. For the Boolean and categorical problems considered in this paper, the training sets contain all fitness cases, so predictive accuracy cannot be assessed. However, it is possible for the regression problems. To this aim, we calculate the error of best-of-run individuals on 10,000 tests drawn at random from the same domain (hypercube) as for the training set. The ranking according to this performance measure is:

BPGP4	BPGP4A	BP2A	GP1	GP1L	GP2
2.65	3.35	3.47	3.47	3.59	4.47

The BPGP-based methods rank best again, but the differences in ranks are less prominent, and, as a result, the Friedman test is inconclusive. However, its p-value is relatively low (0.142), which makes it likely that BPGP superiority would become significant on a larger repertoire of problems.

Run time. Behavioral evaluation and maintenance of an archive impose additional computational costs. To assess their impact, we measure the total run time of every algorithm (which terminates if ideal solution is found or when the number of generation elapses). Methods ranked according to runtime are (see detailed results in Table 3):

GP1	GP2	BPGP4	BPGP4A	BP2A	GP1L
1.09	**3.11**	**3.77**	**3.97**	**4.09**	**4.97**

Expectantly, conventional GP running with small population is much faster than all the other methods. However, it performs much worse on the other measures considered above.

It is interesting to see that the BPGP-based methods are faster than GP1L. In combination with the results on other measures, this allows us to conclude that B4A does not only find better solutions that GP1L, but also does it at a lower computational expense.

Program size. The ranking of configurations according to the size of the best-of-run program is as follows:

GP2	BPGP4	BP2A	BPGP4A	GP1	GP1L
1.26	**2.57**	**2.89**	**3.91**	**4.81**	**5.56**

GP2 wins on this measure, which may be attributed to the fact that (i) it uses program size s as one of the objectives (contrary to GP1 and GP1L), and (ii) s is the only additional objective (other than program error f) that it employs (as opposed to BPGP4 and BPGP4A that also use yet another two objectives, c and e). As a result, s is a very important objective in this configuration, and allows is to produce very compact programs. However, given the poor performance of GP2 on the other measures, one may argue that the impact of this objective in this particular configuration is detrimental (as opposed to BPGP4 and BPGP4A, where it acts as only as one of four objectives).

The above ranking takes into account the best-of-run individuals of all evolutionary runs, whether successful or not. For the successful runs only, the outcome is

BPGP4	GP2	BP2A	BPGP4A	GP1	GP1L
1.35	2.35	3.2	4.1	**4.9**	**5.1**

It turns out then that BPGP4 is able to provide ideal solutions that are on average even smaller than those of GP2 (though the difference is not statistically significant).

4.4 Experiment summary

The experimental results clearly indicate that BPGP is more likely to find an ideal solution than the traditional GP methods, proves capable of solving problems that GP struggles with, produces programs that commit smaller error and generalize better, requires moderate computational effort, and in some configurations (BPGP4) yields smaller ideal solutions.

Another important outcome is that combining behavioral evaluation with an archive-based search operator (in BPGP4A) seems to be beneficial, particularly for success rate and program error. This corroborates the hypothesis posed earlier, that these two components can complement each other and lead to synergy. However, this does not mean that behavioral evaluation or archive-based search operator alone have no virtues: for instance, BPGP4 seems a viable method for finding compact programs that generalize very well.

5. CONCLUSION

The gathered evidence suggests that behavioral programming is a promising methodology for evolutionary synthesis of programs. BPGP, a method that represents this approach, provides superb results for many problems from various domains, is universal by not requiring any adjustments to a domain or problem, is almost parameterless (except for archive size), and can employ any suitable machine learning technique as a generator of behavioral models.

Another appeal of behavioral programming is that the additional 'search drivers' are in a sense *invented* by the search algorithm. By relying on ML-induced behavioral models, behavioral programming can autonomously detect behavioral regularities/patterns that reveal potentially useful candidate solutions (via behavioral evaluation) and parts thereof (via manipulation of subprograms). No background knowledge or human ingenuity is necessary for that purpose: the experimenter is not required to, e.g., specify, in addition to providing an objective function, which properties/qualities of subsolutions are desirable. This makes behavioral programming attractive when compared to, e.g., some reinforcement learning methods in which the additional search drivers have to be explicitly provided [19].

One might wonder why could such search drivers be more useful than the original objective function? In the end, it is the objective function that comes as a part of problem formulation and ultimately characterizes solution quality. Unfortunately, as demonstrated in countless studies on fitness-distance correlation, in problems with complex fitness landscapes, with intricate neighborhood structure, plateaus, and local optima, there is absolutely no guarantee that the objective function correlates with the amount of computational effort it will take to solve the problem. Therefore, it is justified to suppose that other objectives (or a combination thereof, as in this study) may lead a given search algorithm more efficiently. As odd as it sounds, the objective function can be sometimes precisely the *worst possible* search driver, as it is the case in deceptive problems.

On a higher level of discourse, we anticipate that behavioral programming is capable of joining a new *paradigm* for program evolution/synthesis, that is based upon semantics, a.k.a. behavior. The particular realization of behavioral programming in this study, BPGP was embedded in conventional tree-based genetic programming. However, nothing precludes applying it to other program representations, or in combination with other search mechanisms.

Acknowledgments. The authors thank the members of the ALFA Group. Krzysztof Krawiec acknowledges financial support from Fulbright Commission and grant no. DEC-2011/01/B/ST6/07318. Una-May O'Reilly acknowledges support from the Li Ka Shing Foundation. Both authors thank Faustynka, Dominika and Michasia Krawiec for verifying Table 3.

6. REFERENCES

[1] A. Bajurnow and V. Ciesielski. Layered learning for evolving goal scoring behavior in soccer players. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 1828–1835, Portland, Oregon, 20-23 June 2004. IEEE Press.

[2] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *Evolutionary Computation, IEEE Transactions on*, 6(2):182 –197, apr 2002.

[3] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[4] T. Haynes. On-line adaptation of search via knowledge reuse. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 156–161, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[5] M. Hollander and D. Wolfe. *Nonparametric Statistical Methods.* A Wiley-Interscience publication. Wiley, 1999.

[6] G. S. Hornby and J. B. Pollack. Creating high-level components with a generative representation for body-brain evolution. *Artif. Life*, 8(3):223–246, 2002.

[7] G. Kanji. *100 Statistical Tests.* SAGE Publications, 1999.

[8] J. D. Knowles, R. A. Watson, and D. Corne. Reducing local optima in single-objective problems by multi-objectivization. In *EMO '01: Proceedings of the First International Conference on Evolutionary Multi-Criterion Optimization*, pages 269–283, London, UK, 2001. Springer-Verlag.

[9] K. Krawiec and J. Swan. Pattern-guided genetic programming. In C. Blum, et al., editors, *GECCO '13: Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 949–956, Amsterdam, The Netherlands, 6-10 July 2013. ACM.

[10] J. Lehman and K. O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 19(2):189–223, Summer 2011.

Table 3: Detailed characteristics of best-of-run individuals, averaged over 30 evolutionary runs. Bold marks the best result for each benchmark and performance indicator (not necessarily statistically significant).

	Success rate (×100)						Average error						Average run time [seconds]						Average program size [nodes]					
	GP1	GP1L	GP2	BP2A	BP4A	BP4	GP1	GP1L	GP2	BP2A	BP4A	BP4	GP1	GP1L	GP2	BP2A	BP4A	BP4	GP1	GP1L	GP2	BP2A	BP4A	BP4
Cmp06	43	77	0	**100**	**100**	97	.015	.007	.089	**.000**	**.000**	.001	48	418	140	68	**15**	64	351	392	**15**	80	156	52
Cmp08	3	57	0	60	**100**	20	.020	.005	.139	.019	**.000**	.025	233	2587	279	825	**220**	933	466	643	**11**	108	242	70
D.A1	0	0	0	13	**47**	0	.216	.212	.222	.131	**.026**	.074	**2**	19	50	128	136	117	8	21	**1**	35	134	48
D.A2	0	10	0	7	**70**	7	.190	.181	.222	.128	**.014**	.059	**5**	53	49	128	95	124	71	74	**1**	33	202	58
D.A3	20	53	0	47	**97**	47	.085	.062	.222	.052	**.001**	.021	**14**	131	49	97	36	94	192	233	**1**	69	152	47
D.A4	0	0	0	0	**23**	0	.222	.222	.222	.211	**.060**	.109	**1**	6	49	104	180	142	**1**	**1**	**1**	6	196	83
D.A5	0	0	0	7	**80**	20	.222	.222	.222	.191	**.011**	.056	**1**	7	50	110	96	126	**1**	**1**	**1**	13	168	68
Keij1	0	0	0	0	0	0	.125	**.073**	.257	.275	.234	.278	**26**	567	99	129	151	116	189	447	**34**	36	60	51
Keij11	7	7	3	10	**33**	20	.659	.604	.882	.212	**.110**	.322	**14**	243	104	160	151	118	90	177	36	29	**27**	34
Keij12	0	0	0	0	0	0	.802	**.661**	.966	.932	.929	.902	**31**	504	103	140	173	135	181	296	**37**	65	112	78
Keij13	10	10	17	7	0	**30**	.231	**.125**	.607	.245	.346	.214	**28**	505	88	139	166	92	161	280	**26**	37	38	34
Keij14	0	0	0	0	0	0	.708	**.657**	.855	.683	.683	.774	**19**	339	86	122	143	113	100	207	**20**	32	35	37
Keij15	0	0	0	0	0	0	.721	.597	.846	**.576**	.613	.697	**24**	412	128	193	213	156	141	295	**45**	64	59	57
Keij4	0	0	0	**3**	0	0	.358	**.249**	.508	.281	.265	.272	**29**	585	93	144	160	127	187	390	**46**	62	85	68
Keij5	0	0	0	0	0	0	.362	**.246**	.759	.479	.583	.569	**43**	783	200	376	468	316	129	243	**26**	40	49	46
M.A1	**83**	**83**	0	40	**83**	43	.013	**.011**	.242	.040	**.011**	.038	**4**	36	66	82	41	61	181	173	**14**	53	142	40
M.A2	50	63	0	50	**97**	30	.053	.038	.187	.047	**.002**	.064	**9**	70	75	64	21	74	184	181	**18**	61	160	44
M.A3	83	90	3	77	**97**	33	.011	.007	.262	.016	**.002**	.044	**8**	49	59	47	27	66	240	253	**13**	67	104	38
M.A4	27	37	0	23	**100**	43	.102	.073	.269	.129	**.000**	.044	14	147	68	92	**9**	65	255	240	**13**	40	115	43
M.A5	77	**100**	13	93	**100**	90	.018	**.000**	.171	.007	**.000**	.007	**4**	21	65	28	14	29	132	146	**19**	57	74	37
Maj06	83	90	3	90	**100**	90	.003	.002	.048	.002	**.000**	.002	**21**	145	203	71	36	132	361	365	**35**	149	280	79
Maj08	23	53	0	47	**93**	0	.013	.005	.051	.010	**.001**	.032	**329**	3170	1156	1769	2019	1834	687	938	**94**	295	563	145
Mux06	**100**	**100**	17	**100**	**100**	87	**.000**	**.000**	.090	**.000**	**.000**	.011	**3**	**3**	132	13	10	38	169	169	**19**	68	117	39
Mux11	53	97	3	93	**100**	10	.020	.001	.200	.005	**.000**	.056	**1391**	3809	6757	9713	9780	13362	481	496	**32**	193	303	70
Nguy10	50	53	30	**60**	43	57	.050	**.026**	.171	.054	.079	.105	**8**	120	82	88	121	76	47	98	**22**	25	27	**22**
Nguy12	0	0	0	0	0	0	.301	**.262**	.436	.335	.314	.289	**17**	246	86	123	164	132	90	152	**13**	41	51	49
Nguy3	**23**	**23**	**23**	3	0	13	.101	**.035**	.255	.127	.136	.286	**15**	310	76	133	145	89	122	245	**22**	49	52	25
Nguy4	7	7	3	3	10	**13**	.130	**.045**	.328	.172	.138	.219	**17**	404	92	122	123	91	119	350	**27**	52	47	37
Nguy5	**3**	**3**	**3**	0	0	0	.019	**.007**	.062	.032	.026	.021	**15**	340	95	115	132	107	104	291	**19**	25	**19**	20
Nguy6	43	43	50	83	**93**	80	.034	.020	.105	.005	**.002**	.024	**9**	213	64	46	39	53	71	183	18	19	17	**15**
Nguy7	**7**	**7**	0	3	0	3	.052	**.023**	.156	.037	.040	.134	**16**	315	101	125	142	104	110	255	**25**	70	87	29
Nguy9	47	50	33	73	87	**100**	.147	.117	.082	.027	.002	**.000**	**8**	104	73	77	78	14	50	85	9	10	15	**7**
Par06	0	13	0	23	**100**	13	.155	.075	.429	.133	**.000**	.137	**118**	2065	188	466	233	404	677	1018	**24**	128	356	108
Par08	0	0	0	0	**40**	0	.309	.207	.486	.374	**.115**	.326	**365**	7359	648	2081	3792	2845	696	1343	**23**	138	581	168
Sext	**3**	**3**	0	0	0	**3**	.029	**.019**	.089	.031	.041	.048	**22**	436	105	137	151	118	164	372	**34**	36	37	45

[11] S. Luke. ECJ evolutionary computation system, 2002. (http://cs.gmu.edu/eclab/projects/ecj/).

[12] J. McDermott, D. R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaskowski, K. Krawiec, R. Harper, K. De Jong, and U.-M. O'Reilly. Genetic programming needs better benchmarks. In T. Soule, et al., editors, *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 791–798, Philadelphia, Pennsylvania, USA, 7-11 July 2012. ACM.

[13] R. I. B. McKay. Fitness sharing in genetic programming. In D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H.-G. Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 435–442, Las Vegas, Nevada, USA, 10-12 July 2000. Morgan Kaufmann.

[14] N. F. McPhee, B. Ohs, and T. Hutchison. Semantic building blocks in genetic programming. In M. O'Neill, L. Vanneschi, S. Gustafson, A. I. Esparcia Alcazar, I. De Falco, A. Della Cioppa, and E. Tarantino, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 134–145, Naples, 26-28 Mar. 2008. Springer.

[15] A. Moraglio, K. Krawiec, and C. G. Johnson. Geometric semantic genetic programming. In C. A. Coello Coello, V. Cutello, K. Deb, S. Forrest, G. Nicosia, and M. Pavone, editors, *Parallel Problem Solving from Nature, PPSN XII (part 1)*, volume 7491 of *Lecture Notes in Computer Science*, pages 21–31, Taormina, Italy, Sept. 1-5 2012. Springer.

[16] A. Moraglio and A. Mambrini. Runtime analysis of mutation-based geometric semantic genetic programming for basis functions regression. In C. Blum, et al., editors, *GECCO '13: Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 989–996, Amsterdam, The Netherlands, 6-10 July 2013. ACM.

[17] J. P. Rosca and D. H. Ballard. Discovery of subroutines in genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 9, pages 177–201. MIT Press, Cambridge, MA, USA, 1996.

[18] C. Ryan, M. Keijzer, and M. Cattolico. Favorable biasing of function sets using run transferable libraries. In U.-M. O'Reilly, T. Yu, R. L. Riolo, and B. Worzel, editors, *Genetic Programming Theory and Practice II*, chapter 7, pages 103–120. Springer, Ann Arbor, 13-15 May 2004.

[19] S. Singh, R. L. Lewis, A. G. Barto, and J. Sorg. Intrinsically motivated reinforcement learning: An evolutionary perspective. *IEEE Trans. on Auton. Ment. Dev.*, 2(2):70–82, June 2010.

[20] L. Spector, D. M. Clark, I. Lindsay, B. Barr, and J. Klein. Genetic programming for finite algebras. In M. Keijzer, et al., editors, *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1291–1298, Atlanta, GA, USA, 12-16 July 2008. ACM.