Behavioral Search Drivers for Genetic Programing

Krzysztof Krawiec[1,⋆] and Una-May O'Reilly[2]

[1] Poznan University of Technology, 60-965 Poznań, Poland
[2] Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, MA, USA

Abstract. Synthesizing a program with the desired input-output behavior by means of genetic programming is an iterative process that needs appropriate guidance. That guidance is conventionally provided by a fitness function that measures the conformance of program output with the desired output. Contrary to widely adopted stance, there is no evidence that this quality measure is the best choice; alternative *search drivers* may exist that make search more effective. This study proposes and investigates a new family of *behavioral search drivers*, which inspect not only final program output, but also program behavior meant as the partial results it arrives at while executed.

1 Introduction

A typical optimization problem can be formalized as $p^* = \arg\min_{p \in P} f(p)$, where f is the *objective function* being optimized (minimized for the sake of this paper), and P is the space of candidate solutions (programs in the case of genetic programming, GP). When searching the entire space P is computationally infeasible, a heuristic search algorithm is used to find a solution $\hat{p}$ that brings $f(\hat{p}) - f(p^*)$ as low as possible. The heuristic employs f to drive the search process; in particular, in evolutionary computation it is common to use f as the fitness function.

Employing the objective function in its original form as such *search driver* appears natural, as it clearly defines the search goal. However, finding an optimal solution is the *ultimate* goal of the algorithm, the reaching of which depends on the decisions made in particular iterations of the search process. To succeed, a search algorithm should make the right decisions in possibly all iterations. Putting that into evolutionary terms, it should promote solutions that are *evolvable*, i.e. likely to turn into better solutions in subsequent iterations. However, evolvable solutions are not necessarily preferred by the objective function, as it typically has no insight into the *prospective* quality of a candidate solution.

We argue that using objective function as a search driver is not always desirable and that better alternatives exist. In GP, additional information can be gathered from program behavior, meant as partial outcomes it arrives at during execution, and used to promote evolvable programs. In [1] we proposed a specific

⋆ Work conducted as a visiting scientist at CSAIL, MIT.

M. Nicolau et al. (Eds.): EuroGP 2014, LNCS 8599, pp. 210–221, 2014.
© Springer-Verlag Berlin Heidelberg 2014

variant of such *behavioral evaluation*, termed Pattern-Guided Genetic Programming (PANGEA), and demonstrated its strengths on a set of benchmarks. Here, we present a rationale for and detailed analysis of behavioral evaluation.

2 Background

A GP task is a set T of *tests* (fitness cases). A test is a pair (x, y), where x is the input to be fed into a program, and y is the desired output. In general, xs and ys can be arbitrary objects, however we limit our interest here to synthesis of Boolean functions, so x is a vector of values of Boolean input variables, and y is a Boolean desired output.

The fitness of a program p is a measure of the conformance of its output with the desired outputs. For each test $(x, y) \in T$, p is provided with input x and executed, returning output which we denote as $p(x)$. We say that p *solves test* (x, y) if $p(x) = y$. The fitness of a program is simply the number of tests it does not solve, i.e.:

$$f(p) = |\{(x, y) \in T : p(x) \neq y\}|. \tag{1}$$

For Boolean problems we will prefer a more concise vector formulation. The desired outputs of tests in T can be gathered into a vector called *target* t, and the outputs produced by p for tests from T into its *semantics* $s(p)$. Then, fitness is the Hamming distance between program semantics and target:

$$f(p) = |s(p) - t|. \tag{2}$$

The fitness defined in this way is obviously minimized fitness, and a program p solves a task if $f(p) = 0$.

3 Motivation

A good search algorithm should be able to find an optimal solution (or a decent suboptimal solution) given limited computational resources. To this aim, the objective function it employs should convey the information about (e.g., be proportional to) the number of steps required to reach the goal (an optimal solution). By 'step' we will mean in the following a single application of a search operator (here: mutation).

Unfortunately, conventional fitness functions used in GP (Eq. 2) do not meet this expectation. To illustrate this, consider the example shown in Table 1. Column t defines the target of a 3-argument Boolean function synthesis task. The next column presents the outputs (semantics) of a program p_0 =(x_1 *and* (x_2 or x_3)) which happens to be an optimal solution to this problem. By mutating p_0 (replacing the *and* instruction with the *or* instruction, as underlined) we obtain program p_1, which commits error 4 on this task (according to Hamming distance). By mutating p_1 again, we obtain p_2, whose error amounts to 2.

Let us now revert this process and assume that p_0 has not been found yet, and p_1 and p_2 are two candidate solutions (e.g., individuals in a population in an evolutionary run). Because p_2 commits smaller error than p_1, a conventional based

Table 1. An exemplary sequence of two mutants p_1, p_2 obtained from program p_0 via one-point mutations (marked in bold)

x_1	x_2	x_3	t	p_0 (x_1 and (x_2 or x_3))	p_1 (x_1 **or** (x_2 or x_3))	p_2 (x_1 **or** (x_2 **and** x_3))
0	0	0	0	0	0	0
0	0	1	0	0	1	0
0	1	0	0	0	1	0
0	1	1	0	0	1	1
1	0	0	0	0	1	1
1	0	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	1	1	1
Objective function f:				0	4	2

fitness function will favor it over p_1. This is unfortunate, because p_2 requires two mutations to reach p_0, while for p_1 one mutation is sufficient[1].

The example illustrates the problem signaled in the Introduction. The objective function is by definition the right yardstick for assessing program *quality*. However, in general it does not correlate well with the number of search steps to the optimum, and so it is not necessarily the best means to predict the 'prospective' quality of a candidate solution and drive the search process. In the following we demonstrate that, at least for domains like GP, alternative search drivers can be defined that prove better in that respect.

4 Behavioral Evaluation of Programs in GP

Behavioral evaluation can be explained by starting with conventional fitness, which is the discrepancy between program output (the value returned by the root node in case of tree-based GP) and the desired output, aggregated over the tests (Eq. 2). Behavioral evaluation, in contrast, takes into account not only the final program output, but also program behavior meant as the intermediate values returned by program subexpressions. In this paper we consider tree-based GP[2], so we collect the values calculated by the k topmost nodes of program tree, i.e., the k first nodes when traversing the program tree in breadth-first order (if the tree happens to be smaller, we use all tree nodes). These values, collected for all tests in T, form a $|T| \times k$ table. The ith table column corresponds to the ith node in the tree and thus forms a *feature* that describes program behavior at that point of its execution. In particular, the first feature corresponds to the root node and thus captures the output of entire program $p(x)$.

We then extend this table by an extra column, which holds the search target t. The extended table forms a training set that defines a task of supervised machine learning from examples, with t serving as a decision attribute. A classifier v is induced from this set and the properties of that classifier are used to define the

[1] In general, another optimal solution $p^* \neq p_0$, $s(p^*) = t$ may exist that can be reached from p_2 through a single mutation. This however does not invalidate this argument. We will return to this issue when explaining our sampling procedure.

[2] In [1], we employed an analogous procedure to gather behavioral features from linear programs written in the Push language.

behavioral fitness of the evaluated program p. We then define behavioral fitness as an aggregate of three components:

1. The conventional objective, i.e., the error committed by the program, $f(p)$,
2. The complexity $c(v(p))$ of the classifier $v(p)$ induced from the training set,
3. The error $e(v(p))$ the classifier v commits on the training set.

The rationale behind taking into account the latter two components is as follows. The trained classifier maps (perfectly or not) the program behavior (captured in the features) onto the desired program output. In an ideal case, its predictions perfectly match the desired output and thus $e(v) = 0$. The closer that error is to zero, the more we can claim that the features (and indirectly program behavior) *relate* to the desired output.

However, classifier error does not tell the whole story about the relatedness between program behavior and the desired output, because the classifier can itself be more or less complex. Consider two programs for which the induced classifiers commit the same error. If one of them is simpler than the other, then we can claim that the behavior of the corresponding program is closer related to the desired output. In the case of decision trees used in [1] and here, complexity can be conveniently expressed as the number of decision tree nodes.

The key motivation for behavioral evaluation is that relatedness may convey information about the *prospective* quality of a program. Low error and/or low complexity indicate that program arrives at intermediate results (captured in features) that can be mapped onto the desired output, and thus a small number of transformations (e.g, mutations) can turn it into an optimal program. Conventional fitness function is insensitive to this aspect of program characteristics, as it observes only the final program outcome and measures only its direct match with the desired output.

5 The Experiment

In following we examine several behavioral search drivers by considering the particular measures defined in the previous section (f, c, e) separately and in aggregation. We are primarily interested in how well a search driver correlates with the expected number of search steps that a program p needs to undergo to reach the search target t, i.e., to arrive at the descendant p' such that $s(p') = t$.

To avoid bias towards a specific set of GP tasks (e.g., commonly used benchmarks), we consider a large sample of tasks. For Boolean tasks with tests enumerating all combinations of input variables, there is one-to-one correspondence between tasks and targets, so we will use these terms interchangeably.

Sampling Procedure. To carry out the analysis, we would ideally consider a sample of programs with known distances from a given target, where by distance we mean the *minimal* number of mutations required to reach the optimum. Obtaining such data is however computationally challenging, as distance is the length of the shortest sequence of mutations, and the number of such sequences grows exponentially with program size. Moreover, as we intend to consider many targets, relying on distance becomes technically infeasible.

Algorithm 1. The pseudocode of the function generating a single random walk in the space of programs. The result is a list of programs of length $n+1$, where each program is a mutant of its predecessor.

```
1:  function RandomWalk(n)
2:      p_0 ←RandomProgram( )
3:      walk ← (p_0)
4:      for i ← 1...n do
5:          repeat
6:              p_i ← Mutate(p_{i−1})
7:          until p_i ≠ p_0
8:          walk ← append(walk, p_i)
9:      return walk
10: end function
```

Due to this limitation, we abandon the use of minimal number of steps in favor of the *expected* number of steps. We generate a random program, assume that it defines a target (GP task), and run a random walk from that program using single-point mutation as a search operator. We chose this search operator as it introduces minimal change possible (compared to, e.g., subtree-replacing mutation). Such a random walk can be seen as a search process in reverse (albeit not explicitly driven by any search driver).

The sampling procedure is shown in Algorithm 1. RandomWalk generates a starting program p_0 by calling the RandomProgram function (which is guaranteed to return a program with non-trivial, i.e., non-constant, semantics). The program defines a target $t = s(p_0)$. p_0 is then mutated n times, and the ordered list of mutants representing the walk is returned by the function upon its completion. However, mutants are not allowed to be syntactically identical with the starting program of the walk (i.e., a walk is not allowed to turn into a cycle; see line 7 of the algorithm). Without this constraint, some walks would return to the starting point, and regularities in the results would be harder to notice.[3]

RandomWalk does not guarantee that i is the *smallest* number of mutations required to transform the starting program p_0 into the ith program of the walk, p_i (and vice versa); a shorter sequence of mutations connecting p_0 and p_i may exist. Also, RandomWalk does not ensure that i is the smallest number of mutations that have to be applied to p_i in order to reach the *target* t. A shorter sequence of mutations may exist that transforms p_i into a yet another program $p' \neq p_0$ such that $s(p') = s(p_0) = t$.

By relying on random walks (and thus on the expected rather than the minimal number of mutations), in the following experiment we are able to consider large programs composed of up to 255 tree nodes (tree depth limit 7), which with four instructions leads to search space cardinality of the order of 4^{255}.

[3] Note however that we allow a walk to revisit any other search point except for the target. This is our deliberate design choice to make the walks behave analogously to a search process, which may cycle, however it terminates when it reaches the target.

Table 2. The parameters of experimental setup

Instruction set: nonterminals:	and, or, nand, nor
Instruction set: terminals:	up to 12 input variables
Program generation (RandomProgram)	ramped half-and-half
Minimal tree size	23 (minimal binary tree using 12 variables)
Maximal tree size	255 (full binary tree of depth 7)
Mutation operator (Mutate)	single point mutation
Walk length	16

Experimental Setup. We compare the characteristics of different search drivers on the domain of Boolean function synthesis (Table 2). The search operator used to generate our random walks is single point mutation, which selects a random node in a program tree and replaces it with another instruction of the same arity. We chose this operator as it introduces a minimal change in program code and thus may be likened to single bitflip mutation in genetic algorithms. The instruction set thus does not contain the 'not' instruction, because it cannot be modified using single point mutation. Terminals are not mutated, so the set of active variables and the target t remain unchanged in a given random walk.

The RandomWalk procedure (Algorithm 1) used to generate programs is ramped half-and-half (RHH) with ramp set to depth 7. As all instructions are binary, half of the programs in the sample (those generated by the 'full' part of RHH) are full binary trees of depth 7, with 127 leaves, which makes them likely to use all or almost all input variables. For the programs generated using the 'grow' case of RHH, we impose a lower size limit of $2 * 12 - 1 = 23$ nodes, so that even the smallest programs have the chance to use all 12 input variables. This limit is also essential for generating sufficiently long random walks (for small trees, a walk generated using single point mutation is doomed to return to the starting point quickly, when it exhausts all combinations of instructions).

The behavioral search drivers (Section 4) gather 15 features from program execution and use J4.8 decision tree inducer to build a classifier [2][3] (an unpruned tree is used, i.e., option -U). Note that this learning method is insensitive to the ordering of attributes, so for instance swapping the arguments of a commutative instruction in a program does not affect its behavioral evaluation.

A random walk's starting program may happen to use all input variables, but is not guaranteed to do so. The input variables that are absent in the starting program become irrelevant for a walk. Thus, although the total number of variables is 12, even a one-variable task may occur in the sample (the zero-variable trivial tasks are rejected at the spot). For every task we determine the number of effective variables, and in the following we factor the results for tasks with variable number varying from 6 to 12 (we assume that tasks with five or fewer variables are too trivial to reveal the kind of regularities we search for). In this way, we avoid aggregation over tasks with different numbers of variables, which would make interpretation of results more difficult.

Absolute (raw) Values of Search Drivers. In this experiment we observe how the search drivers vary along random walks. We compare the search drivers

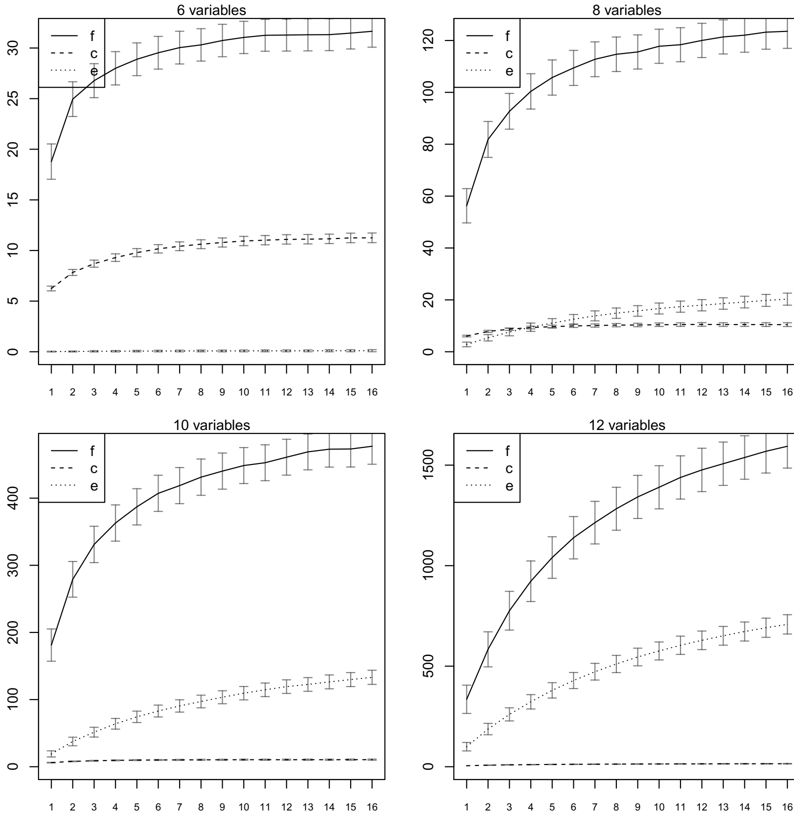

Fig. 1. Search drivers as a function of the number of mutations, for tasks with 6, 8, 10, and 12 relevant variables. f: program error, c: classifier complexity, e: classifier error. Whiskers mark 10% of standard deviation.

presented in Section 4: program error (f), and the behavioral measures: the complexity of the classifier induced from program behavior (c), and the classifier error (e). We first build a sample of random walks by calling RandomWalk $100,000$ times. For every walk, we iterate over its elements (mutated programs), and apply every search driver to them, assuming that the target is defined by the first program in the walk ($t = s(p_0)$). We factor the results by the number of relevant variables in a problem.

Figure 1 presents the search drivers as functions of the number of mutations, averaged over the walks in the sample. Let us emphasize that, while these averaged curves are clearly monotonic, the search drivers for *individual* walks most often are not: a subsequent mutation is likely to decrease the value of the driver (as illustrated in the example in Section 3). Thus, the variance of the raw data is very high: the whiskers show only 10% of standard deviation. Presenting individual walks would make the figures completely illegible.

As mutations accumulate, the features gathered from the behavior of a program become less related to the target, so the classifier has to be more complex

(increasing c) to correctly predict the desired output, and it tends to commit more errors (increasing e). Depending on problem size, the behavioral search drivers grow simultaneously (which happens here for problems with 8 and 10 variables[4]), or individually: for the small problems (6 variables) the classifier is often perfect ($e = 0$), while for the difficult ones (12 variables) building a well-performing classifier becomes impossible so c levels-off. However, at least one of them keeps rising with the subsequent mutations, and in this sense they complement each other. This suggests that c and e can be aggregated to form a compound search driver that would monotonically increase with the number of mutations and avoid stagnation. Given the relative comparison of standard deviations, there is a chance that such compound driver could be less prone to leveling-off than the conventional fitness function f, but this analysis cannot be deemed conclusive in this respect.

The decision tree classifiers are very small on average (search driver c), even for tasks that involve more input variables. This can be explained by strong interdependencies between features collected from GP subexpressions. For instance, consider a program that contains a compound expression (p_1 *and* p_2), and that this expression and its subexpressions p_1, p_2, are behavioral features used by our approach. If a given decision tree node uses the compound expression, the features p_1 and p_2 will be always *true* in the 'positive' branch of that node, and the tree induction algorithm will not use them in that part of the tree.

Correlation Analysis. Above we analyzed the *absolute* values of search drivers. However, what matters in practice is often only whether a search driver increases or decreases with the expected number of mutations to target, particularly when the search is driven by *relative* comparisons of candidate solutions (e.g., tournament selection). Thus, here we analyze how search drivers *qualitatively correlate* with the number of accumulated mutations.

We use the sample of walks generated in the previous experiment. First, we factor it with respect to the number of relevant variables. Next, we plot the Spearman correlation coefficient between a search driver and the number of mutations from all data points with up to l mutations, where $l = 1..16$. We choose the Spearman coefficient as it relies on ranks and thus cares only about the qualitative differences between values (as argued above). Effectively then, a point with abscissa l in the graphs shows correlation coefficient for random walks trimmed to length l.

The results, shown in Fig. 2, are partially consistent with the absolute values shown in Fig. 1. As mutations accumulate, the correlation coefficients for f and c deteriorate, as these search drivers tended to levels-off the most in Fig. 1 (except for c for 6 variables). The classifier error e however maintains relatively stable correlation along the walk, though it is rather low compared to the remaining drivers (so low that for 6 variables the plot is out of plotting range). Overall, the behavioral drivers start becoming competitive and often provide better correlation than f, Interestingly, they frequently do so for long random walks, which

[4] The growth of c is barely visible due to the range of vertical axis.

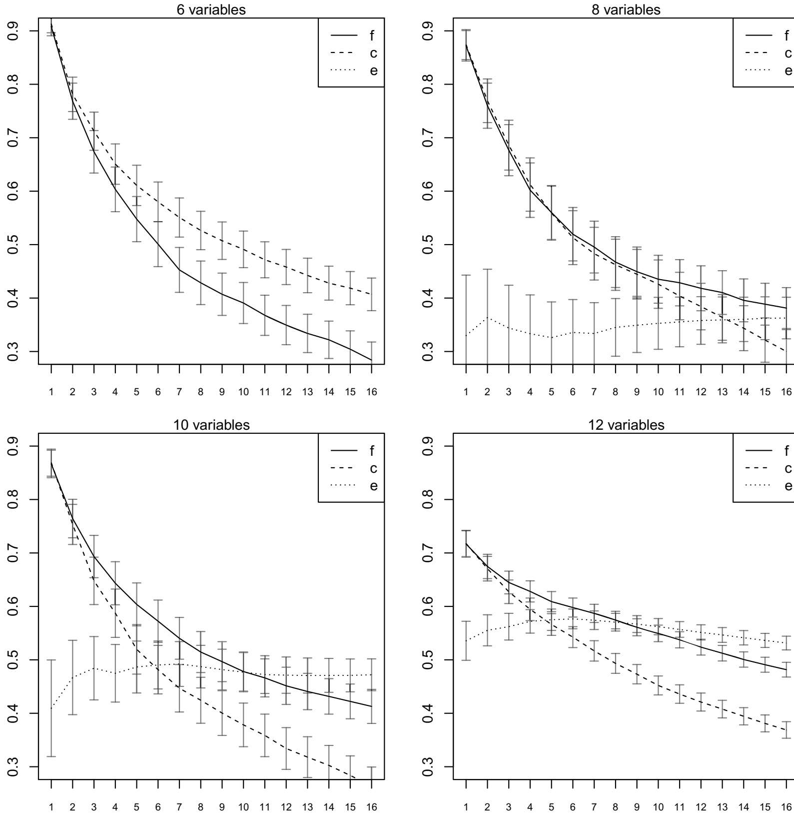

Fig. 2. Spearman coefficient between the number of mutations in a walk and search drivers, for walks of different total length (horizontal axis). Whiskers mark 0.99-confidence intervals.

looks particularly attractive, as it suggest that they may provide better search gradient far from the target.

The new result in comparison to Fig. 1 is that c and e are largely uncorrelated (if they were, their plots would have to be similar). This suggests, even more than Fig. 1, that c and e may complement each other in a nontrivial way.

Correlation Analysis for Compound Search Drivers. As the individual search drivers saturate after a number of mutations (Fig. 1), we do not expect any of them alone to be a useful search driver, so here we try to aggregate them.

Direct additive aggregation of our search drivers would be unjustified, as c is the number of tree nodes, while f and e are expressed in tests (and c is typically much smaller, see Fig. 1). To provide a common platform for these quantities, we resort to information theory and attempt to estimate the amount of information conveyed by these components. For simplicity, rather than calculating the *exact* number of bits required to encode f, e, and c, we simply pass each of them

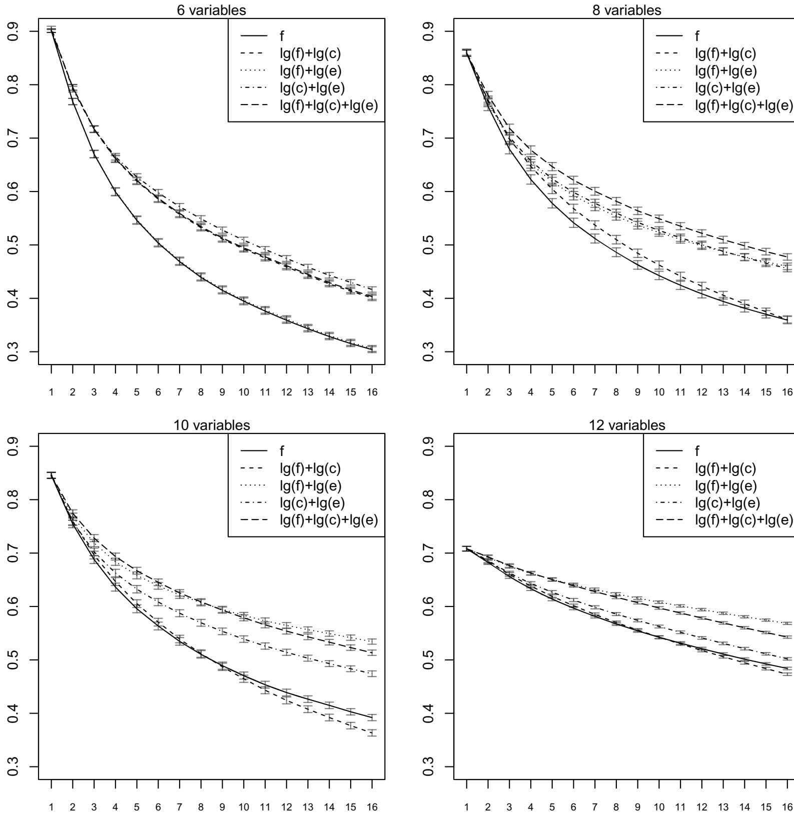

Fig. 3. Spearman correlation between the number of mutations in a walk and **aggregates** of search drivers, for walks of different total length (horizontal axis). Whiskers mark 0.99-confidence intervals.

through a logarithm, meant as a rough measure of information content (see, e.g., [4] for considerations on exact formulas).

We consider compound measures constructed from all combinations of f, c, and e, and show them in Fig. 3, with the plot for f repeated after Fig. 2 for reference[5]. Comparison of these plots to individual search drivers in Fig. 2 clearly suggests that fusing the behavioral search drivers is beneficial. For all combinations of components, correlation is not worse, and often better, than that for conventional fitness f. In particular, the aggregate that consistently (i.e., for all considered numbers of variables) provides the highest or close to the highest correlation involves all components ($\lg(f) + \lg(c) + \lg(e)$), which corroborates the outcomes we obtained with PANGEA [1].

The 0.99-confidence intervals clearly indicate that the compound search drivers are in most cases significantly better than the conventional fitness function f. However, what can be the potential implications for the efficiency of a search

[5] We skip the logarithmic transform for f, as it is monotonic so it cannot affect ranking, and thus has no effect on Spearman correlation.

algorithm? A thorough answer to this question requires a separate investigation; for Push programs behavioral evaluation dramatically improved search efficiency [1]. Here we can demonstrate how the behavioral search drivers extend the reach of effective learning gradient. For instance, for the 10-variables problems, fitness function has correlation of 0.51 for walks of length 8, while $\lg(f) + \lg(c) + \lg(e)$ provides roughly the same correlation for walks up to length 16. In other words, the behavioral search driver maintains roughly the same capability of estimating solution's distance from the target for programs that are more than twice as far from it. As the number of programs that can be reached by a random walk grows exponentially with walk length, the 'basins of attraction' for behavioral search drivers can be much greater than for the conventional fitness function.

The correlation coefficients for behavioral fitness functions are greater than those for conventional fitness, yet still far from perfect. However, attaining full correlation is unrealistic, as it requires the ability to perfectly predict the number of mutations required to reach the optimum. Nevertheless, all correlations reported here are statistically significant: the control values obtained by permutation testing are well below 0.1 (null hypothesis: no interrelationship).

6 Related Work

The approach presented here is novel in its attempt to exploit program's *internal behavior* for search efficiency. However, there are examples of using alternative, non-behavioral search drivers in GP, the most notable being implicit fitness sharing [5] and its extensions [6].

The way in which we investigated the various performance measures resembles the studies on fitness-distance correlation (e.g., [7]). However, the performance measures under scrutiny here included not only conventional fitness, but also the behavioral drivers. Also, we relied on the expected number of search steps, rather than a distance, as a measure of anticipated computational effort.

The MDL principle has been used in GP means of controlling the trade-off between model complexity and accuracy. For instance, Iba *et al.* [8] used it to prevent bloat in GP by taking into account the error committed by an individual as well as the size of the program. A few later studies followed this research direction (see, e.g. [9]).

By focusing mostly on the effects of program execution (the partial outcomes reflected in trace features) rather than on syntax, behavioral evaluation can be seen as following the recent trend of semantic GP, initiated in [10]. Interestingly, it also resembles evolutionary synthesis of features for machine learning and pattern/image analysis tasks [11]. However, here the classifier serves only as a scaffolding for evolution; it is supposed to provide 'gradient' when the program output alone is unable to do so.

7 Conclusion

We demonstrated that behavioral search drivers provide more reliable information about the expected number of mutations needed to reach the optimum.

Given two candidate solutions, behavioral evaluation is more likely to predict correctly which of them requires fewer modifications to reach the search target, which has obvious implications for search efficiency. Also, by providing a more comprehensive information of program's *prospective* quality, it extends the effective learning gradient further from the target, and promotes evolvability.

The presented results characterize the domain of Boolean functions, and abstract from any specific task in that domain. Given the analogous results obtained with with a different program representation and for non-Boolean problems [1], we hypothesize that behavioral evaluation can be potentially leveraged in different genres of GP.

Acknowledgments. The authors thank the reviewers for rich feedback and constructive suggestions. Both authors acknowledge support from the Li Ka Shing Foundation, and K. Krawiec acknowledges support from the Polish-U.S. Fulbright Commission and from grants no. DEC-2011/01/B/ST6/07318 and 91507.

References

1. Krawiec, K., Swan, J.: Pattern-guided genetic programming. In: Blem, C., et al. (eds.) GECCO 2013: Proceeding of the Fifteenth Annual Conference on Genetic and Evolutionary Computation Conference, Amsterdam, The Netherlands, pp. 949–956. ACM (2013)
2. Quinlan, J.: C4.5: Programs for machine learning. Morgan Kaufmann (1992)
3. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: an update. SIGKDD Explor. Newsl. 11(1), 10–18 (2009)
4. Quinlan, J.R., Rivest, R.L.: Inferring decision trees using the minimum description length principle. Inf. Comput. 80(3), 227–248 (1989)
5. Smith, R., Forrest, S., Perelson, A.: Searching for diverse, cooperative populations with genetic algorithms. Evolutionary Computation 1(2) (1993)
6. Krawiec, K., Lichocki, P.: Using co-solvability to model and exploit synergetic effects in evolution. In: Schaefer, R., Cotta, C., Kołodziej, J., Rudolph, G. (eds.) PPSN XI. LNCS, vol. 6239, pp. 492–501. Springer, Heidelberg (2010)
7. Tomassini, M., Vanneschi, L., Collard, P., Clergue, M.: A study of fitness distance correlation as a difficulty measure in genetic programming. Evolutionary Computation 13(2), 213–239 (2005)
8. Iba, H., Sato, T., de Garis, H.: System identification approach to genetic programming. In: Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Orlando, Florida, USA, vol. 1, pp. 401–406. IEEE Press (1994)
9. Zhang, B.T., Mühlenbein, H.: Balancing accuracy and parsimony in genetic programming. Evolutionary Computation 3(1), 17–38 (1995)
10. McPhee, N.F., Ohs, B., Hutchison, T.: Semantic building blocks in genetic programming. In: O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcázar, A.I., De Falco, I., Della Cioppa, A., Tarantino, E. (eds.) EuroGP 2008. LNCS, vol. 4971, pp. 134–145. Springer, Heidelberg (2008)
11. Krawiec, K., Bhanu, B.: Visual learning by evolutionary and coevolutionary feature synthesis. IEEE Transactions on Evolutionary Computation 11(5), 635–650 (2007)