F O U N D A T I O N S O F C O M P U T I N G A N D D E C I S I O N S C I E N C E S
Vol. 42 (2017) No. 4
ISSN 0867-6356
DE GRUYTER OPEN DOI: 10.1515/fcds-2017-0017 e-ISSN 2300-3405

Adaptive Test Selection for Factorization-based Surrogate Fitness in Genetic Programming

Krzysztof KRAWIEC, Paweł LISKOWSKI *

Abstract. Genetic programming (GP) is a variant of evolutionary algorithm where the entities undergoing simulated evolution are computer programs. A fitness function in GP is usually based on a set of tests, each of which defines the desired output a correct program should return for an exemplary input. The outcomes of interactions between programs and tests in GP can be represented as an interaction matrix, with rows corresponding to programs in the current population and columns corresponding to tests. In previous work, we proposed SFIMX, a method that performs only a fraction of interactions and employs non-negative matrix factorization to estimate the outcomes of remaining ones, shortening GP's runtime. In this paper, we build upon that work and propose three extensions of SFIMX, in which the subset of tests drawn to perform interactions is selected with respect to test difficulty. The conducted experiment indicates that the proposed extensions surpass the original SFIMX on a suite of discrete GP benchmarks.

Keywords: genetic programming, matrix factorization, surrogate fitness, test-based problems, recommender systems.

1. Introduction

Computational intelligence abounds in *test-based problems*, i.e. problems that feature *tests*, entities that embody pieces of knowledge about the problem. Training examples in machine learning, opponents in games, and environments in reinforcement learning and robotics are all examples of tests. In these settings, an intelligent agent interacts with tests and learns according to the outcomes of those interactions: a machine learning inducer builds an appropriate hypothesis, a game-playing algorithm adjusts its strategy, and a virtual or physical robot updates its policy. The common features of

*Institute of Computing Science, Poznan University of Technology, Poland, {kkrawiec, pliskowski}@cs.put.poznan.pl

these scenarios are that the number of tests may be large (or even infinite), interactions with them are largely independent, and each of them produces an outcome that may provide the agent with useful feedback.

Test-based problems attracted much attention in *coevolutionary algorithms*, a branch of evolutionary computation. *Competitive coevolution* devised there assumes iterative co-adaptation of *candidate solutions* (entities intended to solve a given problem) with *tests*. In the simplest case, the candidate solutions and tests dwell in separate populations, and the former are rewarded for the number of passed tests, while the latter for the number of solutions they fail. More sophisticated algorithms that prove better in practice may reward tests for their ability to *discern* the solutions, i.e. elicit diversified outcomes from them, as this helps providing gradient for a search process. Competitive coevolution is particularly appropriate when the number of tests is large, in which case it becomes difficult or impossible to let candidate solutions interact with all of them. Past work proved this search paradigm useful for learning game strategies [4], intelligent agents [33] and machine learners [2].

In competitive coevolution, the working population of tests changes with time. However, the conceptual framework of test-based problems comes in handy also in the more general setting of an evolutionary algorithm, where the set of tests T is fixed and given as a part of problem formulation, like the training set of examples in machine learning. This is the default setting for *genetic programming* (GP) that is the focus of this paper, where candidate solutions are symbolically represented executable structures like programs or expressions. In that case, a test is a pair $t = (in, out)$ of a program input in and the associated desired (expected) output out. An interaction of a candidate program p with t involves applying p to in and verifying whether the actual output produced by p is out, or otherwise how much it diverges from it. In the simplest case, the interaction outcome is just $[p(in) = out]$, where $[\,]$ is the Iverson bracket, i.e. 0 or 1 for p *failing* or *passing* t, respectively. This definition of interaction outcome is used in domains where programs return discrete values; in continuous domains, absolute or square error may be more appropriate.

In this paper, we are interested in the methods that employ the formalism of non-negative matrix factorization (NMF) to interpret and process the *interaction matrix* G that arises from confronting the programs in population with the tests in T. One of them, Surrogate Fitness via Factorization of Interaction Matrix (SFIMX) [23] heuristically estimates the fitness from G using NMF. Crucially, that estimation requires only some elements of G to be known, which implies that only some program-test interactions have to be conducted. This allows substantial reduction of required computational effort, because running programs on tests is usually the main cost component in GP.

The original SFIMX samples uniformly the interactions to be conducted in each generation of a GP run (see Section 2). In this work, we present three novel extensions of SFIMX (Section 3), where the sampling process is being adaptively biased to take into account the difficulty of individual tests. Experimental evaluation presented in Section 5 and summarized in Section 6 demonstrates that at least one of those variants systematically outperforms the original SFIMX in terms of success rate, i.e. the likelihood of producing a correct program, while imposing only a small computing

Algorithm 1 Surrogate Fitness via Factorization of Interaction Matrix (SFIMX).

Require: factorization rank k.

1: **function** SFIMX(P, T, α)
2: **for** $p \in P$ **do**
3: $T' \leftarrow \text{SAMPLE}(T, \alpha)$
4: **for** $t \in T'$ **do**
5: $g(p,t) \leftarrow \text{INTERACT}(p,t)$ ▷ apply program p to test t
6: $W, H \leftarrow \text{NMF}(G, k)$
7: $\hat{G} \leftarrow WH$ ▷ predict missing g_{ij}s
8: **for** $p_i \in P$ **do**
9: $f(p_i) \leftarrow \sum_{j=1}^{n} \hat{g}_{ij}$
10: **return** F

overhead.

2. SFIMX

As most other variants of evolutionary algorithms, GP performs iterative parallel search in the space of candidate programs, maintaining a working population P of m programs. Each iteration of that process, deemed *generation*, involves evaluation of programs in P, selecting the well-performing of them (*parents*), and their modification with search operators, which in turn leads to creation of *offspring* that form the subsequent population. This study focuses on the evaluation phase. Therein, each candidate program $p \in P$ interacts will all n tests from T, and the outcomes of those interactions are aggregated into a scalar *fitness value* that reflects the overall quality of a program. For instance in discrete domains, one often assumes that the fitness function $f(p)$ counts the number of tests passed by p, i.e.:

$$f(p) = \sum_{t \in T} [p(in) = out]. \tag{1}$$

Surrogate Fitness via Factorization of Interaction Matrix (SFIMX) [23] approximates f while reducing the number of required interactions between programs and tests. In each generation, SFIMX first calculates a sparse interaction matrix G between the programs in P and the tests in T. To this end, for every program $p \in P$, it *randomly* draws (without replacement) a subset of tests $T' \subset T$ to interact with of size $\lfloor \alpha n \rfloor$, where $\alpha \in (0, 1]$ is the parameter that controls the fraction of interactions to be calculated. Next, it performs interactions by applying p to the tests in T' and storing the resulting interaction outcomes in the appropriate cells of G. The remaining cells are treated as unknown. Then, G is factorized into non-negative components W and H that are used to reconstruct the missing interaction outcomes in G as:

$$\hat{G} = WH,$$

where $W \in \mathbb{R}^{m \times k}$ is a *weight matrix* and $H \in \mathbb{R}^{k \times n}$ is a *feature matrix*. Each program $p \in P$ is associated with a row in W (a vector $w_p \in \mathbb{R}^k$), and each test $t \in T$ corresponds to a column in H (a vector $h_t \in \mathbb{R}^k$).

To perform factorization, NMF solves the following optimization problem:

$$\min_{W,H} f(W,H) \equiv \frac{1}{2}||G - WH||_F^2 \;\; s.t. \;\; W, H \geq 0, \tag{2}$$

where $||\cdot||_F$ is the Frobenius norm. Once W and H are known, an estimate of the interaction outcome of p with t is given by the dot product of the two corresponding vectors:

$$\hat{g}(p,t) = w_p^T h_t = \sum_{j=1}^{k} w_{pj} h_{jt}. \tag{3}$$

We will refer to the elements of G by g_{ij} and of $\hat{G}$ by $\hat{g}_{ij}$, or alternatively $g(p,t)$ and $\hat{g}(p,t)$ (so that ps are assumed to correspond to is and matrix rows, and ts to js and matrix columns).

Finally, SFIMX calculates the fitness of a program $p \in P$ as:

$$f(p) = \sum_{j=1}^{n} \hat{g}_{ij}, \tag{4}$$

i.e. in the same way as in the conventional GP, the difference being that the basis of fitness calculation is here the partially estimated $\hat{G}$ matrix, rather than a complete G matrix. These steps are summarized in Algorithm 1.

The factorization model is optimized using only the known elements of G, and ignoring any missing (unknown) interaction outcomes. Implementations of NMF based on stochastic gradient descent are well suited for this task, as they allow training the model using only a sample of interaction outcomes. Furthermore, gradient-based NMF algorithms [19] usually converge in at most a few dozens of steps, even when G is relatively large. They also guarantee finding an optimal factorization, because the functional being optimized is convex w.r.t. W and H (though not with respect to both of them simultaneously).

To comply with NMF's requirement of G's elements being strictly positive, in line 5 of Algorithm 1, we set $g(p,t) = 1$ if p fails t and $g(p,t) = 2$ otherwise.

Note that $\alpha \geq \frac{1}{|T|}$ must hold for T' to be nonempty. Apart from α, SFIMX's main parameter is k, which controls the sizes of W and H, and the role of which should become more clear in the example that follows.

Example

Consider the population of programs $P = \{p_1, p_2, p_3, p_4\}$ and the population of tests $T = \{t_1, t_2, t_3, t_4, t_5\}$. Assume that SFIMX is run with $\alpha = \frac{3}{5}$ and yields the following

sparse interaction matrix G:

$$G = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \end{array} \begin{array}{c} \begin{array}{ccccc} t_1 & t_2 & t_3 & t_4 & t_5 \end{array} \\ \begin{pmatrix} 2 & & 1 & 2 & \\ & 2 & 1 & & 1 \\ 1 & & & 2 & 2 \\ 2 & 1 & & & 1 \end{pmatrix} \end{array}$$

Let $k = 3$. , The application of NMF to G (line 6 of Algorithm 1) returns the following decomposition into W and H:

$$W = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \end{array} \begin{array}{c} \begin{array}{ccc} f_1 & f_2 & f_3 \end{array} \\ \begin{pmatrix} 0.46 & 1.96 & 0.6 \\ 1.27 & 0.1 & 0.95 \\ 1.37 & 0.02 & 2.83 \\ 0.4 & 1.86 & 1.60 \end{pmatrix} \end{array}, \quad H = \begin{array}{c} \\ f_1 \\ f_2 \\ f_3 \end{array} \begin{array}{c} \begin{array}{ccccc} t_1 & t_2 & t_3 & t_4 & t_5 \end{array} \\ \begin{pmatrix} 0.48 & 1.50 & 0.01 & 0.41 & 0.41 \\ 0.87 & 0.14 & 0.19 & 0.77 & 0.01 \\ 0.11 & 0.09 & 1.02 & 0.50 & 0.51 \end{pmatrix} \end{array}.$$

When multiplied (line 7), W and H lead to the following reconstructed interaction matrix:

$$\hat{G} = WH = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \end{array} \begin{array}{c} \begin{array}{ccccc} t_1 & t_2 & t_3 & t_4 & t_5 \end{array} \\ \begin{pmatrix} 2 & 1.02 & 1 & 2 & 0.52 \\ 0.8 & 2 & 1 & 1.07 & 1 \\ 1 & 2.31 & 2.1 & 2 & 2 \\ 2 & 1 & 2.01 & 2.4 & 1 \end{pmatrix} \end{array}$$

Finally, in line 10, we calculate the fitness of particular programs by summing the corresponding rows of the reconstructed interaction matrix, which results in $f(p_1) = 6.54$, $f(p_2) = 5.87$, $f(p_3) = 9.41$, and $f(p_4) = 8.41$. Overall, SFIMX enabled calculating these values using $\alpha nm = 12$ known interaction outcomes, compared to $nm = 20$ interactions required by the conventional GP. ■

In the above example, the reconstructed matrix $\hat{G}$ perfectly reproduces the known interaction outcomes, so the square approximation error minimized by NMF (Eq. 2) attains zero. This is guaranteed to happen when $k \geq rank(G)$. In general, the approximation error tends to be greater for smaller k and greater α. It should be noted however that, regardless how well NMF reconstructs the known interaction outcomes, the unknown ones are only extrapolated, so the value of SFIMX's fitness will in general diverge from the true fitness (Eq. 1).

3. Extending SFIMX with adaptive test selection

In this work, we question the way in which SFIMX draws the tests to interact with in line 3 of Algorithm 1. Our aim is to improve its performance by replacing the uniform probability distribution that is used there and changing it adaptively as a run progresses. The motivation is that the outcomes of interactions with certain tests are likely to be more difficult to predict than others. Should that be true, then it

might be particularly beneficial to compute these interactions (i.e., run the candidate programs on them) rather than use SFIMX to estimate them from the remaining elements of G. And conversely: if there are grounds to claim that, for some programs and tests, the outcomes of their interactions are easy to predict, then it may be worth to not perform them and let them be estimated by NMF.

Obviously, whether a test (in, out) is difficult *for a given program* p cannot be known for certain without actually running p on in. Nevertheless, various measures of difficulty (e.g., the odds for t being passed by a random program) may be estimated based on the historical interaction outcomes, gathered throughout a GP run. This is the key idea behind the extensions we describe in the following: the interaction matrix is inspected in each generation, certain statistics are updated and used to parameterize the probability distribution employed by the SAMPLE function in line 3 of Algorithm 1. Crucially, only the actual (computed) interactions from the historical Gs are used for this purpose – the estimated ones in $\hat{G}$s are ignored. Therefore, the prediction of test difficulty is based on the interaction outcomes that were computed in the past, so no extra interactions are required. By inspecting the interaction matrix in every generation, the methods adapt the probability distribution to the current state of population, shifting SAMPLE's attention towards the tests that were the most challenging in the recent generations.

We propose three methods of controlling SAMPLE. The first one, dubbed ***Diff***, is based on *test difficulty* $r(t)$, which we define as the number of programs that did not pass t up to the current point of evolutionary run (i.e., up to the previous generation). For successive populations P, we update this number in a vector r (initially zeroed) for every test in T:

$$r(t) \leftarrow r(t) + |p \in P : \ g(p,t) = 1|. \tag{5}$$

In evaluation with SFIMX, r is L_1-normalized (i.e., divided by $\sum_t r(t)$) and used as the probability distribution by SAMPLE. As a consequence, the more difficult tests have a higher chance of being included in T'. We expect this probability distribution to improve the estimation of interaction outcomes, as the outcomes of interactions with easier tests should be in principle also easier to predict. More difficult tests, on the other hand, are typically characterized by greater variability in the interaction outcomes, thus we expect SFIMX to commit greater errors in their estimation.

In the second variant, ***Dist***, we employ the concept of *distinctions*, borrowed from competitive coevolution [8]. A test t is said to make a distinction between programs p_1 and p_2 if $g(p_1,t) \neq g(p_2,t)$. By analogy to r above, we use the current population to update the total number of distinctions $q(t)$ made so far by t in the following way:

$$q(t) \leftarrow q(t) + |\{(p_1,p_2) \in P \times P : \ g(p_1,t) \neq g(p_2,t)\}|, \tag{6}$$

where q is initially all-zeroes. Large values of q indicate the tests that *inform* search algorithm about the differences between programs, rather than about their absolute performance. Otherwise, *Dist* works as *Diff*, i.e. q is normalized and used by SAMPLE. Interestingly, there seems to be a link between the concept of distinctions and the concept of error variance used in works that combine GP with coevolution [30].

While the above two methods use ad-hoc measures of tests difficulty to infer which interaction outcomes are more difficult to predict, there exists a more direct alterna-

tive: measuring the difficulty of prediction using the difference between the known elements of G and the corresponding estimated elements of $\hat{G}$. The resulting method, referred to as ***Err*** in the following, accumulates those errors $e(t)$ throughout the run similarly to *Dist* and *Diff*,:

$$e(t) \leftarrow e(t) + \sum_{p \in P} |g(p,t) - \hat{g}(p,t)|, \tag{7}$$

where e is then normalized and used in SAMPLE. In contrast to r (Eq. 5) and q (Eq. 6) that depend solely on the performance of candidate programs, e depends also on the quality of factorization, and so reflects the inherent estimation difficulty (which depends, among others, on the factorization rank k).

Let us emphasize that, even though the probability distributions defined above replace the uniform distribution used in the basic SFIMX, drawing of tests is still performed independently for each program (and with replacement between individual programs). It is thus possible (though unlikely for the common proportions of population size and number of tests) for some tests to be not included in T' in a given generation. Also, let us reiterate that all these methods accumulate only the outcomes of the interactions that have been actually performed in the past, so that the quantities aggregated by the above formulas are not biased (at least directly) by the errors committed by NMF when estimating the unknown interaction outcomes.

4. Related Work

The concept of matrix factorization (MF) recently grew in popularity in machine learning and recommender systems. In collaborative filtering [29], where MF is often applied, the general idea is to model the user-item interactions with factors representing the latent characteristics of the users and items in the system, like for instance a preference class of users and a category of items. This model is trained using the available data, and later used to predict users'ratings of the previously unseen or new items. Such systems exploit the similarities in rating behavior among users for determining their predictions. These principles stood behind the winning solution to the famous Netflix Challenge, where MF was applied to predict millions of ratings [16]. MF is an example of model-based approach to recommender systems. Other approaches to item recommendation and rating prediction include, among others, Bayesian Clustering [3], Latent Semantic Analysis [12] and Singular Value Decomposition [1].

In SFIMX, the fitness based on reconstructing the matrix of interaction outcomes with MF can be treated as a *surrogate fitness*. In evolutionary computation, a surrogate fitness function provides a computationally cheaper approximation of the original objective function. Surrogates are particularly helpful in domains where evaluation is computationally expensive, e.g., when it involves simulation or engages large volumes of data (e.g., examples in GP). They usually rely on simplified models of the process being simulated, hence their alternative name: *surrogate models* [14]. In continuous

optimization, such models are typically implemented using low-order polynomials, Gaussian processes, or artificial neural networks.

In addition to SFIMX presented in Section 2 [23], several other studies attempted to reduce the number of evaluations in evolutionary algorithms (EAs) for test-based problems. The arguably simplest approach is to draw a subset of tests $T' \subset T$ and allow the candidate solutions interact only with them. This approach was studied in the context of EAs, where it is known as Random Sampling Evolutionary Learning [5]. Apart from speeding up the evolution, the motivation is that candidate solutions that perform well on different subsets of tests might have captured the knowledge about generalizing to all tests in T. Random selection of tests has been shown to improve the success rate and reduce overfitting [10].

In a similar spirit to SFIMX, matrix factorization has been used as the primary building block of cevolutionary interaction scheme designed for interactive domains [20]. The method adaptively decides how much of the interaction matrix to compute based on the learning speed statistics. The remaining outcomes of interactions between the pairs of coevolving individuals are then estimated using NMF. The method maintains the precision of round-robin tournament while, at the same time, considerably improves the learning speed.

Matrix factorization has also been used as a means to 'multiobjectivize' GP. Discovery of Objectives via Factorization (DOF) proposed in [22] employs NMF to heuristically derive a low number of search objectives from an interaction matrix, and uses these objectives to drive the search. The observation that motivates DOF is that NMF can be used to explain the interaction outcomes in G by characterizing both programs and tests in terms of factors (the W matrix) inferred from the patterns observed in their interactions. Each factor (a column in W) becomes a *derived objective*, recasting so the original single-objective GP problem as a multi-objective problem. In every generation, DOF feeds the factors from W directly into NSGAII, the state-of-the-art multiobjective selection method [7], in order to select parent programs and generate candidate solutions for the next generation.

Another method that aims at scrutinizing the individual outcomes of interactions and leveraging them for better performance is DOC [18]. In every generation, the algorithm identifies the groups of tests on which the programs in the current population behave *similarly* and clusters them together to give rise to new search objectives. Typically, a few such objectives emerge from clustering, each supposed to capture a subset of 'capabilities' exhibited by the programs in the context of other individuals in the population. The derived objectives replace then the conventional fitness function and are used to drive the selection process. DOC is rooted in the previous work in coevolutionary algorithms and test-based problems [21, 25].

Regarding other approaches, the SFIMX variants studied here can be likened to methods that redefine fitness function. The arguably best known approach of this type is implicit fitness sharing (IFS) introduced by Smith *et al.* [31] and further explored for genetic programming by McKay [27]. IFS estimates the difficulty of particular tests from an interaction matrix and weighs accordingly the rewards granted to programs. Higher rewards are provided for solving tests that are rarely solved by population members. The difficulties of tests are calculated from the current population and

thus change with evolution, which can help escaping local minima and diversifies the population. Diversification maintenance was also the main motivation for the recent Lexicase selection algorithm [11], that avoids aggregating interaction outcomes altogether: in each selection act, a random permutation of tests is generated, and the program from the current population which passes the longest uninterrupted sequence of tests is selected. Lexicase proved very effective in a range of contexts and applications.

Last but not least, certain aspects of the SFIMX variants considered in this paper link to the adversarial settings of coevolutionary algorithms (cf. the second paragraph of Introduction). Here, we adjust the probabilities of programs engaging with individual tests in a *predefined* manner, motivated by the anticipated benefits of, e.g., exposing programs more frequently to the more difficult tests. A competitive coevolutionary algorithm, with a separate population of programs and a separate population of tests that coevolve with them, can be seen as an *emerging* way of adjusting those probabilities. The extent to which this analogy holds depends on several factors, including the way in which the tests are evaluated (note the remark in Section 3 on SFIMX-*Dist* being inspired by the concept of distinctions, a specific way of evaluating tests in competitive coevolution [8]). A range of works in GP investigated the possibility of coevolving tests, starting from the seminal work by Pagie and Hogeweg [28], to the idea of coevolving fitness predictors along with programs [30]. The similarities notwithstanding, coevolutionary approaches are more suitable for open-ended settings; in a down-to-earth GP practice, one cares primarily (and in some cases exclusively) about the available tests and assumes that no other knowledge about the problem is given (and thus no additional tests that are guaranteed to be consistent with the target concept can be obtained).

5. Experimental evaluation

5.1. Methods

We examine the performance of SFIMX extensions in the domain of tree-based GP, following the experimental settings and benchmark problems used in [23]. All compared methods implement the generational evolutionary algorithm and share the same parameter settings, with the initial population filled with the ramped half-and-half operator, subtree-replacing mutation engaged with probability 0.1, subtree-swapping crossover engaged with probability 0.9, and tournament of size 7 in the selection phase. The fitness of each program $p \in P$ is computed based on the $\hat{G}$ as a sum of corresponding rows of the matrix (Eq. 4). Search stops when the 200 generations elapse. To verify whether an ideal program has been found, in the last generation of an evolutionary run, we evaluate the best program(s) on all tests. This amounts to computing additional $(1-\alpha)n$ interactions. Runs that find such an ideal program are considered a success.

We are interested in verifying whether the extensions *Diff*, *Dist*, and *Err* improve

the performance of a regular SFIMX algorithm in particular variants proposed in [23]. For this reason, we consider as the baselines all SFIMX configurations from [23], i.e.

- **SFIMX-full** that uses the highest factorization rank $k = \min(m, n)$,
- **SFIMX-half** with $k = n/2$ that forces the interaction outcomes to be compressed in half the number of weights in matrix W and features in matrix H, and
- **SFIMX-log** that uses the smallest rank $k = \lceil \log_2 n \rceil$.

We set the fraction of conducted interactions to $\alpha \in \{0.4, 0.5, 0.6\}$, as these proved most effective for the original SFIMX [23].

The population size is $|P| = 1000$ for the non-SFIMX methods (introduced below), while for SFIMX it is increased according to the number of spared evaluation cycles. As SFIMX spares $(1 - \alpha)mn$ interactions per run when running with population of size m and n tests in T, we increase its population size by the factor of $(1 - \alpha)$, so that it consists of $m + (1 - \alpha)m$ individuals. As a result, the overall computational budget is the same for SFIMX and the baseline configurations, and amounts to $1{,}000n$ interactions per generation and thus $200{,}000n$ interactions per run.

The non-SFIMX baseline methods include conventional Koza-style **GP** and Random Subset Selection (**RSS**). The latter calculates fitness using a subset of αn randomly selected tests, drawn anew in every generation. This proceeding is intended to mimic the evaluation scheme known in coevolutionary algorithms [5]. The computations were conducted on a cluster of uniform PCs, with 2.6 GHz Intel Xeon E5-2697 processors and 64 GB of memory. Interactions between programs and test were carried out in parallel (typically 8 threads per run). The methods discussed in this paper were implemented in Python programming language using evolutionary computation framework called DEAP [9].

5.2. Benchmark problems

The multiplicative update NMF algorithm used by SFIMX can in principle factor an arbitrary non-negative interaction matrix. However, obtaining good reconstructions for interaction outcomes that vary strongly in magnitude might be difficult. Unconstrained interaction outcomes can be expected when applying GP to continuous domains (so-called *symbolic regression*), where programs can commit arbitrarily large errors. Also, raw interaction outcomes in symbolic regression problems are signed (the difference between the real-valued actual and desired output) and as such would require a well-justified mapping to positive numbers to meet the non-negativity requirement of NMF. For these reasons, in this study we consider only problems with discrete interaction outcomes.

The first group are Boolean benchmarks, which employ the instruction set {*and, nand, or, nor*} and 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. For the majority

Maj v problems, *true* should be returned if more that half of input variables are *true*. For the multiplexer *Mulv,* the state of the addressed input should be returned (6-bit multiplexer uses two inputs to address the remaining four inputs). In the parity *Parv* problems, *true* should be returned only for an odd number of *true* inputs.

The second group of benchmarks are the algebra problems from Spector *et al.*'s work on evolving algebraic terms [32]. These problems dwell in a ternary domain (the admissible inputs and outputs are $\{0, 1, 2\}$) and use 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 is defined as follows:

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

We consider five algebras (the above a_1 and four others defined in [32]) and two tasks that are of interest to mathematicians [6]. In the *discriminator term* tasks (*Dsc* in the following), the goal is to synthesize an expression that accepts three inputs x, y, z and is semantically equivalent to the one shown below:

$$t^A(x, y, z) = \begin{cases} x & if\, x \neq y \\ z & if\, x = y \end{cases} \tag{8}$$

There are thus $3^3 = 27$ fitness cases in these benchmarks. The second tasks (*Mal*), consists in evolving a so-called *Mal'cev term*, i.e., a ternary term that satisfies the equation:

$$m(x, x, y) = m(y, x, x) = y \tag{9}$$

This condition specifies the desired program output only for some combinations of inputs: the desired value is undefined for $m(x, y, z)$, where x, y, and z are all distinct. As a result, there are only 15 fitness cases in the *Mal* tasks, the lowest of all considered benchmarks.

In total, the five algebras and two tasks give rise to 10 benchmarks in out algebra domain (e.g., Mal3 means the task of evolving the *Mal'cev* term using algebra a_3).

5.3. Discussion

Figure 1 shows the average best-of-generation fitness graphs for particular methods and benchmark problems, with 95% confidence intervals marked as semi-transparent bands. For brevity, we present only the best performing configurations, which happen to be those that used $\alpha = 0.4$ and *Diff* extension. By comparing the extended SFIMX variants with the corresponding baseline SFIMX configurations (marked by the same color on the graph), we observe significant improvements in learning speed and best-of-generation fitness in the methods that employ *Diff*. For certain problems, including Dsc1, Dsc2, Dsc5 and Mal4, the differences between the methods are particularly prominent, showing clear superiority of *Diff* to the original SFIMX. The best performance is achieved either by *Diff-Full* or *Diff-Half*, depending on the problem.

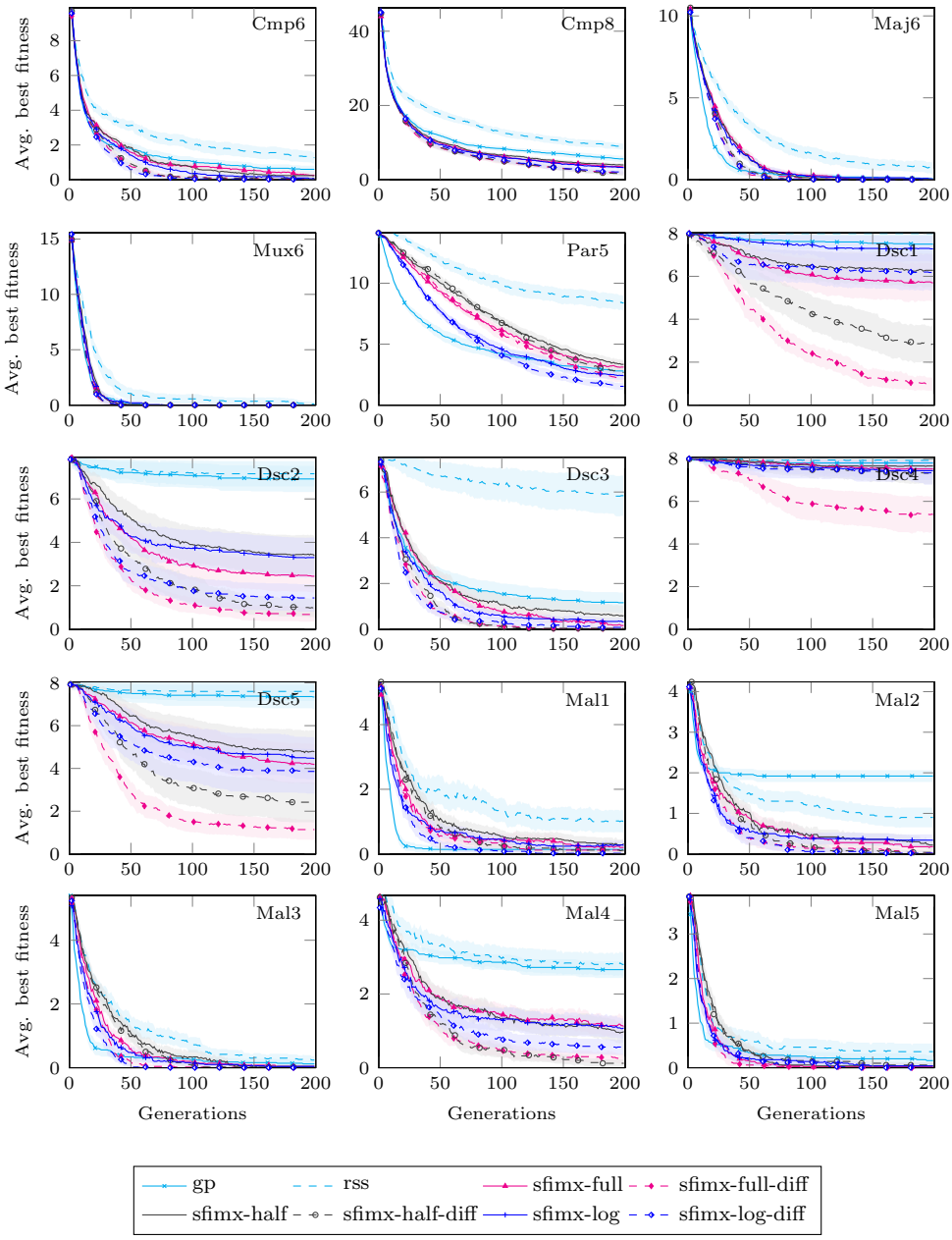

Figure 1. Average and .95-confidence interval of the best-of-generation fitness.

Table 1. Success rate (×100) of best-of-run individuals, averaged over 50 evolutionary runs.

α	Cmp6	Cmp8	Maj6	Mux6	Par5	Dsc1	Dsc2	Dsc3	Dsc4	Dsc5	Mal1	Mal2	Mal3	Mal4	Mal5	Rank	Time
								SFIMX-FULL									
0.4	74	4	100	100	6	12	18	90	0	20	82	82	98	40	98	15.83	09:52
0.5	88	6	94	100	16	0	18	82	0	12	76	88	98	30	92	19.47	06:49
0.6	82	12	96	100	12	4	2	76	0	14	76	70	100	24	90	22.17	06:17
								DIFF									
0.4	96	26	98	100	14	50	54	98	4	50	90	98	100	80	100	4.90	10:52
0.5	92	22	100	100	10	12	24	94	0	32	88	96	100	66	98	10.03	09:18
0.6	96	22	100	100	30	0	10	90	0	20	88	84	100	64	100	10.60	08:18
								DIST									
0.4	80	14	98	100	8	8	2	57	0	8	60	60	88	18	96	26.20	19:38
0.5	94	20	98	100	12	0	12	56	0	6	72	64	90	14	96	23.77	10:19
0.6	85	22	94	100	4	2	12	56	0	6	74	60	88	10	88	27.97	09:12
								ERRS									
0.4	86	12	96	100	8	4	10	60	0	20	88	87	96	20	96	19.43	11:32
0.5	92	14	96	100	12	6	6	70	0	22	64	70	96	20	92	20.13	10:51
0.6	86	10	96	100	10	2	4	70	0	24	74	84	96	16	92	22.23	08:36
								SFIMX-HALF									
0.4	86	4	96	100	4	6	12	70	0	20	80	78	94	36	94	21.77	10:01
0.5	85	12	97	100	16	2	22	78	0	12	82	74	98	36	96	16.13	06:00
0.6	80	12	96	100	8	0	4	64	0	6	96	66	98	36	100	22.03	05:23
								DIFF									
0.4	90	36	97	100	6	34	32	98	0	48	90	98	98	88	96	10.17	11:15
0.5	92	36	100	100	10	0	24	84	0	14	94	90	100	60	100	11.43	08:56
0.6	92	14	100	100	10	8	14	82	0	14	84	96	100	46	100	11.40	07:56
								DIST									
0.4	78	22	100	100	0	4	8	66	0	8	82	76	88	24	90	22.13	14:23
0.5	88	8	92	100	8	0	8	64	0	10	80	70	92	20	94	25.47	10:41
0.6	82	10	94	100	8	0	6	54	0	6	80	66	94	18	96	26.57	08:26
								ERRS									
0.4	80	4	92	98	4	6	18	90	0	12	80	74	100	42	96	21.27	13:48
0.5	96	12	98	100	14	0	8	74	0	8	76	92	98	28	98	17.50	09:30
0.6	86	16	98	100	6	0	10	86	0	6	80	74	90	28	94	21.30	08:09
								SFIMX-LOG									
0.4	94	10	98	100	12	4	20	72	0	28	70	66	100	36	96	17.63	09:43
0.5	94	12	98	100	14	4	12	74	0	4	84	68	84	22	96	19.50	07:36
0.6	84	12	96	100	10	0	12	70	0	8	72	54	96	16	92	25.53	06:56
								DIFF									
0.4	96	36	100	100	16	10	36	94	2	42	98	98	100	56	100	4.50	09:47
0.5	92	38	98	100	14	12	20	92	0	28	98	82	100	44	98	9.13	07:53
0.6	94	14	100	100	22	2	22	78	0	8	88	72	98	44	100	12.47	08:35
								DIST									
0.4	96	18	90	100	12	2	10	57	0	4	62	64	94	2	94	25.37	15:46
0.5	96	14	98	100	4	0	4	56	0	14	78	48	84	2	84	26.53	11:54
0.6	94	8	96	100	10	0	8	64	0	8	68	36	92	4	82	27.07	10:11
								ERRS									
0.4	86	20	100	100	8	2	12	76	0	18	78	64	100	22	98	17.97	12:52
0.5	92	8	98	100	8	0	12	66	0	20	80	48	96	16	96	21.73	09:54
0.6	80	18	89	100	22	2	2	72	0	6	76	52	86	2	96	25.03	09:17
								RSS									
0.4	52	0	55	95	0	0	0	45	0	0	68	48	78	25	85	35.50	06:19
0.5	65	0	45	95	0	0	0	45	0	0	68	48	78	8	85	35.67	05:19
0.6	68	2	52	92	2	0	0	32	0	5	68	65	82	8	88	35.30	05:10
								GP									
	56	6	94	100	6	0	0	30	0	6	88	2	90	0	82	31.17	04:28

Table 2. Post-hoc analysis of Friedman's test for *Diff*. Significant values (0.05) are in bold.

	SFIMX $\alpha = 0.4$							
	GP	RSS	FULL	DIFF-FULL	HALF	DIFF-HALF	LOG	DIFF-LOG
GP		0.946						
RSS								
FULL	0.233	**0.009**			0.782		1.000	
DIFF-FULL	**0.000**	**0.000**	**0.041**		**0.000**	0.911	**0.025**	1.000
HALF	0.989	0.467						
DIFF-HALF	**0.002**	**0.000**	0.782		**0.042**		0.494	
LOG	0.494	**0.037**			0.955			
DIFF-LOG	**0.000**	**0.000**	0.137		**0.001**	0.955	**0.042**	

Diff-Log performs slightly worse or falls in between the already mentioned configurations, but most importantly, it still achieves lower fitness than any baseline setups, including RSS and GP. These observations are confirmed by Table 1 that reports the success rates of all three extensions and the baselines, resulting from 50 runs of each configuration on every benchmark.

To provide an aggregated perspective on the performance of the *Diff* extension against the other methods, we employ the Friedman's test for multiple achievements of multiple subjects [15] on the best-of-run fitness. The p-value 1.41×10^{-9} strongly indicates that at least one method performs significantly different from the remaining ones. To determine the significantly different pairs, we conduct post-hoc analysis using symmetry test [13]. Table 2 presents the p-values for the hypothesis that a setup in a row is better than a setup in a column, with the significant values marked in bold. We report the results only for $\alpha = 0.4$ that consistently leads to the highest success rates (cf. Table 1). The comparison indicates that the improvement of *Diff* relative to the regular SFIMX across all configurations is indeed significant. In particular, *Diff-Full* is significantly better than *Full*, *Half* and *Log*, as well as the control setups GP and RSS. Similar observations can be made for *Diff-Half* and *Diff-Log* that outperform their counterparts. For other values of α (0.5 and 0.6), *Diff* still delivers improvements and surpasses SFIMX, GP and RSS on most benchmarks (cf. Table 1).

For a more detailed insight, we also rank all 40 configurations on each benchmark and present the averaged ranks in the last-but-one column of Table 1. The best overall average rank of 4.50 is achieved by *Diff-Log*. The second-best method is *Diff-Full* with the average rank of 4.90. The ranking reveals also that the methods employing the *Err* sampling typically rank better than *Dist*, which may suggest that in certain cases the errors from factorization are more useful than distinctions for shaping the probability distribution of test drawing. GP and RSS occupy the last few places in the ranking, achieving the lowest success rates of all compared methods. It is also interesting to see that *Log* behaves so well, regardless of the choice of parameters. By using the smallest factorization rank $k = \lceil \log_2 n \rceil$, it is on average computationally cheaper than the other methods. This advantage is crucial for larger problems that involve more tests and yield bigger interaction matrices. Indeed, in terms of pure wall clock time (cf. last column in Table 1), *Log* takes the least time to run a single trial compared to other SFIMX variants. *Half* and *Full* are more computationally expensive, which was expected, given that they operate on larger matrices.

Table 3. Post-hoc analysis of Friedman's test for *Dist*. Significant values (0.05) are in bold.

	SFIMX $\alpha = 0.4$						
	GP	RSS	FULL	DIST-FULL	HALF	DIST-HALF	LOG	DIST-LOG
GP		0.715						
RSS								
FULL	**0.002**	**0.000**		0.192	0.831	0.439	1.000	0.210
DIST-FULL	0.831	**0.039**						
HALF	0.192	**0.001**		0.966		0.999		0.973
DIST-HALF	0.550	**0.009**		1.000				1.000
LOG	**0.008**	**0.000**		0.387	0.959	0.689		0.413
DIST-LOG	0.810	**0.034**		1.000				

Table 4. Post-hoc analysis of Friedman's test for *Err*. Significant values (0.05) are in bold.

	SFIMX $\alpha = 0.4$						
	GP	RSS	FULL	ERR-FULL	HALF	ERR-HALF	LOG	ERR-LOG
GP		0.746						
RSS								
FULL	**0.006**	**0.000**		0.952	0.503	0.669	0.998	0.979
ERR-FULL	0.166	**0.001**			0.991	0.999		
HALF	0.669	**0.019**						
ERR-HALF	0.503	**0.008**			1.000			
LOG	**0.047**	**0.000**		1.000	0.889	0.960		1.000
ERR-LOG	0.112	**0.000**		1.000	0.974	0.995		

The overhead of collecting additional statistics required by SFIMX extensions typically does not exceed 10% of a trial time (except for *Dist*, where distinctions involve pairs of programs and thus imply quadratic complexity). Nevertheless, GP and RSS tend to complete faster. These differences in run times stem primary from SFIMX performing NMF in every generation of an evolutionary run. In this study, we used our own implementation of NMF, for the sake of clarity and to avoid dependency on third-party libraries. Nevertheless, switching to a more efficient implementation (including, for instance, a potentially very efficient GPU implementations[1]) is possible, and would certainly largely reduce the SFIMX overheads resulting from the use of NMF. We expect such an implementation diverge only slightly in execution times from GP and RSS, our baseline methods. Further reduction of SFIMX run time could be easily achieved by lowering the number of gradient descent steps taken while learning the NMF model. We used 200 steps in order to guarantee convergence, but typically a good factorization can be found in at most a few dozen of steps. For this reason, we argue that it is sufficient to identify the computational effort with the number of interactions taken place between the individuals in order to reliably compare the methods discussed in this study.

In contrast to *Diff*, *Dist* and *Err* fail to deliver any performance boost upon the original SFIMX configuration. The baseline SFIMX setups obtain better success rates and typically rank higher in Table 1. According to Tables 3 and 4, which present the post-hoc analysis for $\alpha = 0.4$, *Dist* and *Err* outperform only RSS. We hypothesize that these methods may fail to differentiate the easy tests from those that are difficult

[1] https://github.com/ebattenberg/nmf-cuda

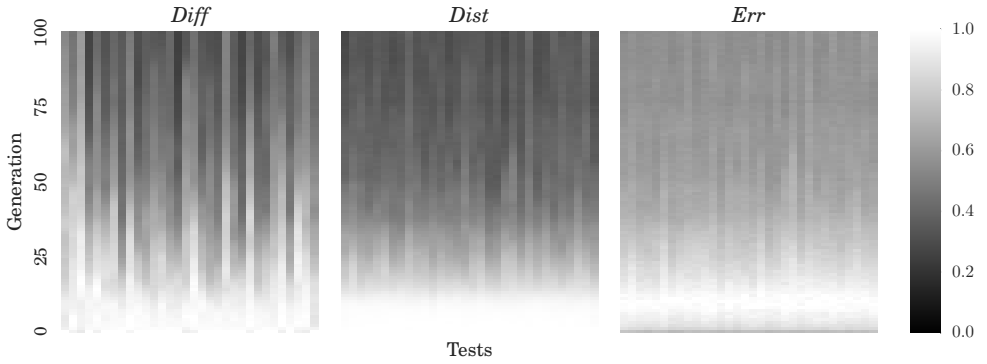

Figure 2. Normalized difficulty of 32 tests in Par5 problem during the first 100 generations as indicated by the three methods: *Diff*, *Dist* and *Err*. The brighter the color, the more difficult the test.

to pass, and do not bias the sampling distribution towards the difficult tests.

In order to gain a deeper insight into the differences between *Diff*, *Dist* and *Err*, in Fig. 2 we plot the changes of the normalized difficulty of all 32 tests in the Par5 problem for the first 100 generations. To create the graphs, we first computed the test difficulty according to Eqs. 5-7 (Section 3). Then, the resulting 32-element vectors were averaged to form the mean test difficulty across 50 runs. Finally, the resulting 32×100 matrix is normalized and presented as a heatmap, with brighter colors corresponding to harder tests.

Judging from the graphs in Fig. 2, *Diff* starts to differentiate tests' difficulty from the early stages of evolutionary runs and manages to maintain that differentiation with time. It also seems to discriminate roughly equal numbers of easy and difficult tests, as illustrated by the dark and bright stripes in the heatmap. As evolution proceeds, the brighter stripes fade away, some faster than others, as programs in the population adapt and solve the more difficult tests. *Dist* behaves similarly, however the differences in the difficulty measure seem to be more subtle and, judging from relatively low performance of this variant, insufficient to shape the probability distribution in a way that would make an impact on the success rate. *Err* is characterized by the most uniform distribution of all three methods. Apparently, NMF's estimation error turns out to be roughly uniformly distributed across all tests. All in all, the visualization provided in Fig. 2 confirms that neither *Dist* nor *Err* discern the tests well enough to guide search more effectively than the corresponding SFIMX baselines, and this is most likely the reason why they are unable to enhance SFIMX's performance.

Finally, we verify whether the methods improve the predictive capabilities of SFIMX. In Table 5 we present the mean absolute estimation error calculated as $\sum_i |G_i - \hat{G}_i|$, where i iterates over generations of a run ($i \in [1, 200]$). To calculate this error, we compute both the complete interaction matrix G alongside with the estimated one ($\hat{G}$), even though the former is never used by fitness. Bold values indicate that an extended configuration achieves lower error than its SFIMX counter-

Table 5. Mean absolute error (×100) when reconstructing G in SFIMX with $\alpha = 0.4$, averaged over 50 evolutionary runs. Bold cells indicate configurations with lower error than the regular SFIMX.

Method	Cmp6	Cmp8	Maj6	Mux6	Par5	Dsc1	Dsc2	Dsc3	Dsc4	Dsc5	Mal1	Mal2	Mal3	Mal4	Mal5
								SFIMX							
Full	11.83	9.75	12.73	14.21	19.93	11.39	16.45	17.14	9.77	16.66	26.72	26.05	26.21	26.03	28.52
Half	13.31	10.76	14.13	16.28	22.09	12.23	16.75	18.13	10.48	16.44	30.66	29.49	27.34	28.36	32.69
Log	13.53	13.01	14.63	17.47	21.31	9.80	14.23	17.43	8.69	13.66	28.74	25.97	24.80	24.76	31.61
								DIFF							
Full	**11.66**	**9.50**	**12.53**	**14.20**	20.17	**10.74**	**15.10**	**15.92**	10.35	**14.84**	**26.62**	**24.95**	**24.18**	**23.07**	29.32
Half	**12.96**	**10.47**	**14.05**	**15.60**	**21.57**	12.41	**16.73**	**17.39**	**10.36**	**14.45**	**29.95**	**27.49**	**26.10**	**25.69**	33.49
Log	**13.40**	**12.83**	**14.38**	**16.80**	**20.50**	**9.22**	**13.96**	**15.80**	8.90	**12.82**	**27.88**	**24.71**	24.81	**23.23**	**30.66**
								DIST							
Full	**11.66**	**9.61**	12.82	14.52	20.38	12.06	**16.13**	**16.55**	10.09	**15.00**	27.25	**25.94**	**25.01**	**24.31**	29.96
Half	**13.19**	**10.57**	14.16	**16.04**	**21.64**	**11.33**	16.84	**17.89**	11.33	**16.30**	30.80	29.76	**26.75**	**27.71**	32.76
Log	**13.49**	**12.92**	**14.34**	**17.03**	**21.20**	10.19	15.98	**16.81**	**8.53**	15.64	29.85	26.20	24.90	**24.12**	31.69
								ERR							
Full	**11.79**	**9.47**	**12.31**	14.33	21.51	19.25	19.15	17.93	15.30	21.86	28.14	28.44	26.35	26.35	30.24
Half	**13.22**	**10.42**	**14.12**	**16.24**	23.45	20.87	20.87	19.83	13.50	23.66	32.39	32.15	29.01	30.25	34.61
Log	14.07	**12.82**	14.84	17.62	23.58	14.42	19.42	18.47	10.65	19.40	31.52	31.04	27.02	29.30	33.67

part. *Diff* and *Dist* tend to systematically decrease the error on most of problems, however the reduction typically does not exceed 12%. *Err* performs noticeably worse, managing to improve the error on just a handful of problems. By juxtaposing these results with the success rates from Table 1, we may conclude that the performance improvement of *Diff* goes on par with the more accurate predictions.

6. Conclusions

The SFIMX extensions proposed in this paper corroborate our earlier claims [24, 18, 26, 23, 17] that tests not only vary in difficulty, but also that this variability can be exploited to make search more effective. We used this property of test-based problems to shape the probability with which the tests are being drawn for interactions. That proved overall beneficial, though the boosts in success rate with respect to SFIMX are not always significant, the trend is clear.

It is not unlikely that further improvements could be achieved with introduction of additional mechanisms. For instance, all methods considered here base their estimates on the entire history of evolutionary run, i.e. on the interaction outcomes for programs from the most recent generations, as well as of the not so well-performing programs from the initial generations. One may argue that some form of *aging* applied to the estimates (e.g., exponential smoothing) may make them more up-to-date, better tuned to the capabilities of the programs in the current population, and thus more beneficial for success rate.

A natural follow-up of this study could be engagement of *Diff*, *Dist*, *Err* to other test-based evaluation methods for GP, some of which employ NMF to obtain a multi-dimensional characterization of candidate programs [22]. As we argued in Introduction, such multi-faceted evaluation helps maintaining a behaviorally diversified population and so makes search more likely to find the basins of high-quality solutions.

Acknowledgments

P. Liskowski acknowledges support from grant 2014/15/N/ST6/04572 funded by the National Science Centre, Poland. K. Krawiec acknowledges support from grant 2014/15/B/ST6/05205 funded by the National Science Centre, Poland. The authors acknowledge support from grant no. 09/91/DSPB/0630. The computations were performed in Poznan Supercomputing and Networking Center.

References

[1] Bell R., Koren Y., and Volinsky C. Modeling relationships at multiple scales to improve accuracy of large recommender systems. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 95–104. ACM, 2007.

[2] Bongard J. and Lipson H. Active coevolutionary learning of deterministic finite automata. *Journal of Machine Learning Research*, 6(Oct):1651–1678, 2005.

[3] Breese J. S., Heckerman D., and Kadie C. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, pages 43–52. Morgan Kaufmann Publishers Inc., 1998.

[4] Chellapilla K. and Fogel D. B. Evolving an expert checkers playing program without using human expertise. *IEEE Transactions on Evolutionary Computation*, 5(4):422–428, 2001.

[5] Chong S. Y., Tino P., Ku D. C., and Xin Y. Improving Generalization Performance in Co-Evolutionary Learning. *IEEE Transactions on Evolutionary Computation*, 16(1):70–85, 2012.

[6] Clark D. M. Evolution of algebraic terms 1: term to term operation continuity. *International Journal of Algebra and Computation*, 23(05):1175–1205, 2013.

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

[8] Ficici S. G. and Pollack J. B. Pareto optimality in coevolutionary learning. In Kelemen J. and Sosík P., editors, *Advances in Artificial Life, 6th European Conference, ECAL 2001*, volume 2159 of *Lecture Notes in Computer Science*, pages 316–325, Prague, Czech Republic, 2001. Springer.

[9] Fortin F.-A., De Rainville F.-M., Gardner M.-A., Parizeau M., and Gagné C. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[10] Gonçalves I., Silva S., Melo J. B., and Carreiras J. M. Random sampling technique for overfitting control in genetic programming. In *Genetic Programming*, pages 218–229. Springer, 2012.

[11] Helmuth T., Spector L., and Matheson J. Solving uncompromising problems with lexicase selection. *IEEE Transactions on Evolutionary Computation*, 19(5):630–643, Oct. 2015.

[12] Hofmann T. Collaborative filtering via gaussian probabilistic latent semantic analysis. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 259–266. ACM, 2003.

[13] Hollander M., Wolfe D. A., and Chicken E. *Nonparametric statistical methods*, volume 751. John Wiley & Sons, 2013.

[14] Jin Y., Olhofer M., and Sendhoff B. A framework for evolutionary optimization with approximate fitness functions. *IEEE Transactions on Evolutionary Computation*, 6:481–494, 2002.

[15] Kanji G. K. *100 statistical tests*. Sage, 2006.

[16] Koren Y., Bell R., and Volinsky C. Matrix factorization techniques for recommender systems. *Computer*, (8):30–37, 2009.

[17] Krawiec K. and Lichocki P. Using co-solvability to model and exploit synergetic effects in evolution. In Schaefer R., Cotta C., Kolodziej J., and Rudolph G., editors, *PPSN 2010 11th International Conference on Parallel Problem Solving From Nature*, volume 6239 of *Lecture Notes in Computer Science*, pages 492–501, Krakow, Poland, 11-15 Sept. 2010. Springer.

[18] Krawiec K. and Liskowski P. Automatic derivation of search objectives for test-based genetic programming. In Machado P., Heywood M. I., McDermott J., Castelli M., Garcia-Sanchez P., Burelli P., Risi S., and Sim K., editors, *18th European Conference on Genetic Programming*, volume 9025 of *LNCS*, pages 53–65, Copenhagen, 8-10 Apr. 2015. Springer.

[19] Lee D. D. and Seung H. S. Algorithms for non-negative matrix factorization. In *Advances in neural information processing systems*, pages 556–562, 2001.

[20] Liskowski P. and Jaśkowski W. Accelerating coevolution with adaptive matrix factorization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, pages 457–464, New York, NY, USA, 2017. ACM.

[21] Liskowski P. and Krawiec K. Discovery of implicit objectives by compression of interaction matrix in test-based problems. In *Parallel Problem Solving from Nature–PPSN XIII*, pages 611–620. Springer, 2014.

[22] Liskowski P. and Krawiec K. Non-negative matrix factorization for unsupervised derivation of search objectives in genetic programming. In Friedrich T., editor, *GECCO '16: Proceedings of the 2016 Annual Conference on Genetic and Evolutionary Computation*, pages 749–756, Denver, USA, 20-24 July 2016. ACM.

[23] Liskowski P. and Krawiec K. Surrogate fitness via factorization of interaction matrix. In Heywood M. I., McDermott J., Castelli M., Costa E., and Sim K., editors, *EuroGP 2016: Proceedings of the 19th European Conference on Genetic Programming*, volume 9594 of *LNCS*, pages 68–82, Porto, Portugal, 30 Mar.–1 Apr. 2016. Springer Verlag.

[24] Liskowski P. and Krawiec K. Discovery of search objectives in continuous domains. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, pages 969–976, New York, NY, USA, 2017. ACM.

[25] Liskowski P. and Krawiec K. Online discovery of search objectives for test-based problems. *Evolutionary Computation*, 25(3):375–406, 2017.

[26] Liskowski P., Krawiec K., Helmuth T., and Spector L. Comparison of semantic-aware selection methods in genetic programming. In Johnson C., Krawiec K., Moraglio A., and O'Neill M., editors, *GECCO 2015 Semantic Methods in Genetic Programming (SMGP'15) Workshop*, pages 1301–1307, Madrid, Spain, 11-15 July 2015. ACM.

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

[28] Pagie L. and Hogeweg P. Evolutionary consequences of coevolving targets. *Evolutionary Computation*, 5(4):401–418, Winter 1997.

[29] Ricci F., Rokach L., and Shapira B. *Introduction to recommender systems handbook*. Springer, 2011.

[30] Schmidt M. D. and Lipson H. Coevolution of fitness predictors. *IEEE Transactions on Evolutionary Computation*, 12(6):736–749, Dec. 2008.

[31] Smith R. E., Forrest S., and Perelson A. S. Searching for diverse, cooperative populations with genetic algorithms. *Evolutionary computation*, 1(2):127–149, 1993.

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

[33] Stanley K. O. and Miikkulainen R. Competitive coevolution through evolutionary complexification. *J. Artif. Intell. Res.(JAIR)*, 21:63–100, 2004.

Received 12.04.2017, Accepted 23.08.2017