Surrogate Fitness via Factorization of Interaction Matrix

Paweł Liskowski(✉) and Krzysztof Krawiec

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

Abstract. We propose SFIMX, a method that reduces the number of required interactions between programs and tests in genetic programming. SFIMX performs factorization of the matrix of the outcomes of interactions between the programs in a working population and the tests. Crucially, that factorization is applied to matrix that is only partially filled with interaction outcomes, i.e., sparse. The reconstructed approximate interaction matrix is then used to calculate the fitness of programs. In empirical comparison to several reference methods in categorical domains, SFIMX attains higher success rate of synthesizing correct programs within a given computational budget.

Keywords: Genetic programming · Test-based problem · Recommender systems · Machine learning · Surrogate fitness

1 Introduction

Conventional fitness evaluation in genetic programming (GP) consists in applying a program to multiple tests (fitness cases) and aggregating the observed differences between the actual and desired program output. Running a program multiple times can be computationally costly, especially when it involves nontrivial computation or requires processing large amount of data. Computational expense becomes particularly high when programs grow large (a common ailment of GP) or engage loops.

Lowering computational cost incurred by evaluation by simply reducing the number of tests is often not a viable option. Few tests implies inaccurate fitness, and consequently a poorly informed search process. Moreover, discarding tests may cause a task to be formally underspecified (underconstrained). For instance, a set of tests for a multiplexer problem that misses even a single test does not technically specify that problem anymore. Also, if the differences between the actual and desired program outputs are discrete, a low number of tests leads to coarse-grained fitness that often fails to differentiate solutions. When, as it is common, the actual and desired output can be compared only for equality, the outcome of a program-test interaction is binary and fitness can assume only $n + 1$ values for n tests.

M. Heywood et al. (Eds.): EuroGP 2016, LNCS 9594, pp. 68–82, 2016.
DOI: 10.1007/978-3-319-30668-1_5

Various means for reducing the number of required program-test interactions other than plain discarding of tests have been proposed in the past. Most of them fall under the category of surrogate fitness and involve measurement of similarity between the inputs of particular tests. In the simplest scenario, an unknown output of a program for a test t is substituted with the known output of that program for a similar test t'. However, designing an appropriate input similarity measure for a given problem requires domain knowledge. And once designed, such a measure may bias the selection of tests to be used as surrogates and lower the likelihood of synthesizing the correct program.

In this paper, we propose a method that builds a Surrogate Function via Factorization of Interaction Matrix (SFIMX) and reduces so the number of interactions. SFIMX, detailed in Sect. 4, is applicable to any domain where interaction outcomes can be encoded as numbers (e.g., symbolic regression, Boolean, integer, etc.) and, unlike the similarity measures mentioned above, does not require additional knowledge. It engages the well-known algebraic concept of matrix factorization, that recently grew in popularity in machine learning and recommender systems. SFIMX is straightforward, performs well in practice (Sect. 6), and has interesting conceptual implications, which we elaborate on in Conclusions.

2 Background

The desired behavior of a program to be synthesized in GP is specified by a set of tests (fitness cases), each being a pair $(x, y) \in T$ of the input x fed into a program and the *desired output* y expected to result from that program execution. T may be sampled from a potentially infinite universe $\mathcal{T} \supset T$.

A GP algorithm solving a program synthesis task maintains a population of programs $P \subset \mathcal{P}$. In every generation, each program $p \in P$ is tested on every test $(x, y) \in T$, in which p is applied to x and returns an output $p(x)$. In other words, p engages in an *interaction* with a test t. The outcome of that interaction can be characterized by a scalar *interaction function* $g(p, t)$. If $p(x) = y$, p is said to *pass* the test and $g(p(x), y) = 1$. Otherwise, we set $g(p(x), y) = 0$ and say that p *fails* (x, y). In this paper, we assume that interaction outcomes are binary, i.e., $g : \mathcal{P} \times \mathcal{T} \to \{0, 1\}$, though in general various *degrees* of passing tests could be considered (for instance by grading them according to the similarity of the actual and desired output).

We gather the outcomes of interactions in an *interaction matrix* G. For a population of m programs and n tests in T, G is an $m \times n$ matrix where g_{ij} is the outcome of interaction between the ith program p_i and jth test t_j. The conventional GP fitness that rewards a program for the number of passed tests can be then written as

$$f(p_i) = \sum_{j=1}^{n} g_{ij}, \tag{1}$$

or alternatively as

$$f(p) = |\{t \in T : g(p, t) = 1\}|. \tag{2}$$

As it follows from the above, *all* elements of G need to be calculated in order to assess fitness values of all programs in P. Therefore, mn program executions are required in every generation of a GP run.

3 Factorization of Interaction Matrix

The motivation behind all methods that aim at reducing the number of program-test interactions is the potential redundancy of interaction matrix. The simplest form of redundancy is test duplication: though we referred above to T as a *set,* it is in practice usually implemented as a *list*, so duplicates are allowed.

Redundancy may also manifest when tests are different but all (or many) programs behave identically on them (in terms of passing or failing). Consider the task of synthesizing a sorting program, where tests are pairs (x, y) of lists and y is the sorted version of x. If the programming language of consideration does not allow any other operation on list elements than comparisons, then once a program passed a specific test of sorting a list of length, say, four, it will pass all other tests with the same permutation of four elements.

The SFIMX method proposed here aims at more subtle type of redundancy, i.e., when the value of the interaction of a program p with a given test t can be reconstructed from the responses of p and other programs in a population to t and other tests. More precisely, reconstructed by means of linear combinations of interaction outcomes. To this aim, we apply the well-known technique of matrix factorization (MF).

Formally, given an non-negative matrix G (interaction matrix in our case) and a desired rank $k \ll min(m, n)$, non-negative matrix factorization (NMF) [1] searches for non-negative matrices (*factors*) W and H that give a lower rank approximation of G as:

$$G \approx WH \;\; s.t. \;\; W, H \geq 0, \tag{3}$$

where $W \in \mathbb{R}^{m \times k}$ is traditionally called *weights matrix* (or *basis matrix*) and $H \in \mathbb{R}^{k \times n}$ is *feature matrix.* Note that each test $t \in T$ is associated with a column in H (a vector $h_t \in \mathbb{R}^k$) and each program $p \in P$ is associated with a row in W (a vector $w_p \in \mathbb{R}^k$). For clarity, we abuse the notation and index the elements, rows, and columns of matrices with programs p and tests t.

The problem given by Eq. 3 is commonly reformulated as the following optimization problem:

$$\underset{W,H}{min} \; f(W, H) \equiv \frac{1}{2}||G - WH||_F^2 \;\; s.t. \;\; W, H \geq 0, \tag{4}$$

where $|| \cdot ||_F$ is the Frobenius norm.

In the simplest scenario, MF model is trained by fitting to the observed interaction outcomes in G. Notice that if G's rank is $\leq k$, there exists an exact solution to (3). However, as it will become clear in a moment, our goal is to generalize in a way that allows predicting unknown interaction outcomes. Thus, caution should be exercised to avoid overfitting the observed data in G. A common extension of the basic MF formula that addresses this issue is *regularization*, which can be implemented by adding a parameter λ and modifying the squared error objective function:

$$\underset{W,H}{min}\ f(W,H) \equiv \frac{1}{2}||G - WH||_F^2 + \lambda(||W||_F^2 + ||H||_F^2)\ \ s.t.\ \ W,H \geq 0. \tag{5}$$

The minimization problem given by (5) is not convex in both W and H simultaneously, however it is convex in either W or H. Thus, by keeping one matrix constant, the other can be found with a simple least squares computation. This strategy is widely known as alternating least squares [25]. Expression (5) can also be minimized using stochastic gradient descent, however the most popular approach to solve this optimization problem is the multiplicative update algorithm [18], which alternates the following two steps:

$$w_{pk} \leftarrow w_{pk}\frac{(GH^T)_{pk}}{(WHH^T)_{pk}} \tag{6}$$

$$h_{kt} \leftarrow h_{kt}\frac{(W^TG)_{kt}}{(W^TWH)_{kt}} \tag{7}$$

In each iteration, the new values of W and H are found by multiplying the current one by a factor that depends on the quality of approximation in (3). The quality of approximation improves monotonically with the application of the above rules [18]. The update rules are applied for a fixed number of iterations or until the error given by the left-hand side of (5) is sufficiently small.

As it follows from (3), to predict an interaction outcome of a program p with a test t from the matrices W and H found by solving (5), we calculate the dot product of two vectors corresponding to p and t:

$$\hat{g}_{pt} = w_p^T h_t = \sum_{k=1}^{k} w_{pk}h_{kt} \tag{8}$$

Crucially for SFIMX, G can be factorized in the above way *even if some of its elements are unknown*, i.e., when G is *sparse*. This makes matrix factorization a powerful tool in machine learning, where it can be used to fill in the gaps in a large matrix (of, e.g., users' recommendations [12]) even if only small part of that matrix is known for certain. However, the update rules given by (6) and (7) implicitly assume that the input matrix is complete. In order to make them work for sparse matrices, a small modification must be introduced so that the unobserved outcomes in G are masked by zeros and ignored during training of the NMF model. Let M be a binary mask where $m_{pt} = 1$ if g_{pt} is known and $m_{pt} = 0$ otherwise. Then the update rules for so-called *weighted non-negative matrix factorization* (WNMF) [20] become:

$$w_{pk} \leftarrow w_{pk}\frac{((M \odot G)H^T)_{pk}}{((M \odot (WH))H^T)_{pk}}, \tag{9}$$

$$h_{kt} \leftarrow h_{kt} \frac{(W^T(M \odot G))_{kt}}{(W^T(M \odot (WH)))_{kt}}, \tag{10}$$

where $\odot$ is the Hadamard (element-wise) product.

4 The SFIMX Algorithm

Based in observations made in the previous section, we propose a method dubbed Surrogate Fitness via Factorization of Interaction Matrix (SFIMX). The method expects two parameters: the factorization rank k and desired density $\alpha \in (0, 1]$ of partial interaction matrix. SFIMX employs the NMF formalisms described in Sect. 3 to replace the conventional fitness evaluation stage of GP algorithm with the following steps:

1. Calculate *in part* the sparse interaction matrix G between the programs from the current population P and the tests from T in the following way:
   (a) For each program p, draw a nonempty random subset of tests $T' \subset T$ of size $\alpha|\mathrm{T}|$ to interact with, where $\alpha \in (0, 1]$ is the parameter that controls the fraction of interactions to be calculated.
   (b) Apply p to tests in T', placing the interaction results in the appropriate cells of the corresponding row of G.
   (c) Fill in the remaining (missing) entries in G with zeros.
2. Factorize G in non-negative components W and H using the multiplicative update algorithm ((9) and (10)).
3. Use the obtained matrices to reconstruct the interaction outcomes by calculating $\hat{G} = WH$.
4. Compute from $\hat{G}$ the fitness of each program $p \in P$ using the conventional formula (1), by substituting g_{ij}s with the values taken from $\hat{G}$, i.e., $\hat{g}_{ij}$s.

For the purpose of the above algorithm it is mandatory to redefine the original interaction function $g(p, t)$ defined in Sect. 2, because zero is reserved for missing interaction outcomes. We assume that g returns 1 if p fails (x, y) and 2 if p solves (x, y). Note also that $\alpha \geq \frac{1}{|T|}$ must hold for T' to be nonempty.

Example. Consider 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 matrix of interactions G between P and T:

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

Let $k = 3$. In step 2 of SFIMX, application of 50 iterations of the multiplicative update algorithm to G results in the following factorization:

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

When multiplied (step 3 of SFIMX), 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 step 4, 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 $\alpha|T||P| = 12$ known interaction outcomes, compared to $|T||P| = 20$ interactions required by the conventional method. ■

In the above example, the reconstructed matrix $\hat{G}$ perfectly reproduces the known interaction outcomes, so the square approximation error (5) attains zero. This is guaranteed to happen when $k \geq rank(G)$. In general the approximation error will have the tendency to be greater for smaller values of k and greater values of α.

Properties of SFIMX. Predictions made by the method are based on how similar programs interact with the tests in T. The similarity in behavior of two programs is calculated based on the similarity in the outcomes of interactions with certain tests. Missing interaction outcomes are predicted based on the feedback from other programs and tests in the population.

As a result, evaluation in SFIMX is *contextual*: prediction $\hat{g}_{pt}$ made for a missing outcome depends not only on corresponding p and t but also on other programs in P and other tests in T. All available outcomes of interactions between programs in P and tests in T together determine the MF model and therefore influence how the predictions for missing outcomes are made. As the programs evolve with time, so does the model. Therefore, SFIMX performs NMF anew with each generation to model the missing interaction outcomes.

By factorizing interaction matrix G, the programs and the tests are projected into a reduced latent spaces that capture their most salient abstract features. The weight matrix H has one column for every abstract feature and one row for every program, and maps the features to the programs. The values in W state how much each feature *applies* to each program. Feature matrix H, on the other hand, has a row for each abstract feature and a column for each test. Every value in H indicates the extent to which a test possesses an abstract feature.

Interestingly, NMF with the least squares objective (Eq. 4) is characterized by an inherent clustering property, i.e., it clusters the columns of interaction matrix G. If additional orthogonality constraint on H is added, i.e., $HH^T = I$, then the minimized objective is equivalent to the one of K-means clustering (except for the non-negativity constraint), i.e., the sum of square of distances from clusters' centroids. In such a case, NMF can be viewed as a relaxed form of K-means where the matrix W contains non-negative cluster centroids and the elements of H are cluster membership indicators. This convergence helps understand how the problem of finding similar programs is internally tackled by NMF. It also reveals certain similarities to the recently proposed DOC algorithm, which we touch upon in the following review of related work.

5 Related Work

The values calculated by SFIMX can be treated as a *surrogate fitness*. Also known as *approximate fitness function* or *response surface* [10], 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. They usually rely on simplified models of the process being simulated, hence yet another alternative name: *surrogate models*. In continuous optimization, such models are typically implemented using low-order polynomials, Gaussian processes, or artificial neural networks. Occasionally, surrogate models have been also used in GP. For instance, in [8], Hildebrandt and Branke proposed a surrogate fitness for GP applied to job-shop scheduling problems. A metric was defined that reflected the behavioral similarity between programs, more specifically how the programs rank the jobs. Whenever an individual needed to be evaluated, that metric was used to locate its closest neighbor in a database of historical candidate solutions and neighbor's fitness was used as a surrogate.

Several other studies in GP attempted to reduce the number of programs' evaluations. An arguably simplest approach is to draw a subset of tests $T' \subset T$ and allow the programs interact only with them. This approach was also investigated in the context of evolutionary algorithms, where it is known as Random Subset Selection (RSS) [3]. Apart from speeding up the evolution, the motivation is that programs that perform well on various different subsets might have captured essential knowledge to generalize to all tests in T. Random selection of tests has been shown to improve the success rate and reduce overfitting [6].

SFIMX redefines fitness function. Several other methods proposed in the past in GP do that too, albeit usually not in terms of linear algebra. The arguably oldest approach of this type is implicit fitness sharing introduced by Smith *et al.* [26] and further explored for genetic programming by McKay [21,22]. IFS lets the evolution assess the difficulty of particular tests and *weighs* the rewards granted for solving them. In this sense, IFS treats tests as limited resources: programs *share* the rewards for solving particular tests, each of which can vary from $\frac{1}{|P|}$ to 1 inclusive. Higher rewards are provided for solving tests that are rarely solved by population members, while importance of tests that are easy is diminished. The assessed difficulties of tests change with evolution, which can help escaping local minima and diversifies population. Diversification maintenance was also the main motivation for the recent lexicase selection algorithm [7], that avoids aggregating interaction outcomes altogether and differentiates programs by comparing them on randomly selected tests.

Another method that aims at scrutinizing the individual outcomes of programs' interactions and leveraging them for better performance is DOC [15]. 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 this process, each of which is intended to capture a subset of 'capabilities' exhibited by the programs in the context of other individuals in population. The newly derived objectives replace then the conventional fitness function are used to drive the selection process. DOC is inspired by previous work in coevolutionary algorithms and test-based problems in [19].

Relying on binary interaction outcomes that only state whether a given test has been passed by a program or not stays in close resemblance to *test-based problems* originating in coevolutionary algorithms [2,5]. In test-based problems, candidate solutions interact with multiple environments – tests. Typically, the number of such environments is very large, making it infeasible to evaluate candidate solutions on all of them. Depending on problem domain, tests may take on the form of, e.g., opponent strategies (when evolving a game-playing strategy) or simulation environments (when evolving a robot controller). Solving a test-based problem requires a learning algorithm to generalize from a sample of tests. Similarly in GP, a synthesized program is expected to generalize beyond the training set and tests often do not enumerate all possible program inputs.

Last but not least, there are certain connections between SFIMX and semantic GP [24] and behavioral [14,16,17] GP methods that define program semantics as the vector of outputs produced by a program for particular tests. From the viewpoint of SFIMX, a single row in an interaction matrix is the outcome of confronting program's semantics with the vector of desired outputs. Recent years have seen a large number of contributions that employ this characterization of program behavior to design new initialization, search, and selection operators [23]. However, those methods are in general not designed to redefine search objectives, which is the primary goal of SFIMX.

6 Experimental Verification

We examine the performance of SFIMX in the domain of tree-based GP. All compared methods implement generational evolutionary algorithm and share the same parameter settings, with initial population of size $|P| = 1000$ 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 using (1). Search lasts up to 200 generations and stops when the assumed number of generation elapses or an ideal program is found; the latter case is considered a success.

Compared Algorithms. We are interested in verifying whether SFIMX is a viable method for reducing the computational cost incurred by evaluation. For that aim, we control the fraction of interactions to be calculated by the parameter $\alpha \in \{0.1, 0.2, \ldots, 1.0\}$ in SFIMX algorithm in Sect. 4. By reducing in each generation the number of interactions by a factor of $(1 - \alpha)$, we spare $(1-\alpha)|P||T|$ interactions per run. We investigate what can be gained by investing these savings in increased population size: we increase the population size by the factor of $(1 - \alpha)$, so that it holds $|P| + (1 - \alpha)|P| = (2 - \alpha)|P|$ individuals. Therefore, population size does not change at all when $\alpha = 1.0$, while for α close to 0 it is almost doubled. Nevertheless, the overall computational budget is the same for all configurations and amounts to $1,000|T|$ interactions per generation and thus $200,000|T|$ interactions per run. This holds for all of the compared algorithms, including the control setups.

We consider three settings of factorization rank k that controls the degree to which the interaction outcomes are being compressed by factorization. The configuration dubbed **SFIMX-full** uses $k = \min(|P|, |T|)$, which is equivalent here to $k = |T|$, because for the considered benchmarks $|P| > |T|$. This value should be considered large, as NMF can then perfectly reproduce the known interaction outcomes, because the rank of G can be at most $\min(|P|, |T|)$.

The **SFIMX-half** configuration uses $k = |T|/2$, which forces the interaction outcomes to be compressed in half the number of weights in matrix W and features in matrix H. However, this number can be still considered quite high, given that we expect the interaction outcomes to be mutually correlated between program and tests.

Finally, the configuration **SFIMX-log** uses the smallest rank $k = \lceil \log_2 |T| \rceil$. In this case, k is in the order of the number of input variables; for instance, for the Mux6 problem $k = \log_2 2^6 = 6$.

The factorization is realized by the WNMF algorithm ((9) and (10)). The regularization factor λ is set to 0.01, as suggested by the common practice. When invoked for a given sparse interaction matrix G, we let WNMF perform up to 50 iterations, each involving both steps, i.e., (9) and (10). If the approximation error (the left-hand size of (5)) drops below 10^{-5}, we stop the optimization earlier. The computational overhead of running WNMF is on average 19 percent of the time spent evaluating programs in the population.

We confront SFIMX with several control setups. The first baseline is the conventional Koza-style **GP** [13]. The second control configuration, dubbed RSS, calculates fitness using $\alpha|T|$ randomly selected tests. The subset of tests is drawn anew in every generation of evolutionary run. We refer to this method as Random

Subset Selection (**RSS**), based on its similarity to an evaluation scheme known in coevolutionary algorithms [3].

Benchmark Problems. SFIMX and the multiplicative update algorithm it involves can in principle factor an arbitrary non-negative interaction matrix G and then reconstruct its approximation $\hat{G}$. However, obtaining good reconstructions for arbitrarily large interaction outcomes might be difficult, and such unconstrained outcomes can be expected for symbolic regression, where they are based on arbitrarily large errors committed by programs on test (not mentioning the possibility of programs returning infinity). Also, the raw interaction outcomes for 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. For these reasons, in this study we limit our interest to problems with discrete interaction outcomes.

The first group are Boolean benchmarks, which employ 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. 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). In the parity $Par\,v$ problems, *true* should be returned only for an odd number of *true* inputs.

The second group of benchmarks are the algebra problems originating from Spector *et al.*'s work on evolving algebraic terms [27]. These problems dwell in a 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 is defined as follows:

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

In the following, the employed algebra is indicated by the suffix the name of term to be evolved. See [27] for the definitions of the remaining four algebras. For each of the five algebras considered here, we consider two tasks (of four discussed in [27]). 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:

$$t^A(x, y, z) = \begin{cases} x & if\, x \neq y \\ z & if\, x = y \end{cases} \tag{11}$$

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:

$$m(x, x, y) = m(y, x, x) = y \tag{12}$$

This condition specifies the desired program output only for some combinations of inputs: 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* tasks, the lowest of all considered benchmarks. The motivation for the discriminator and Mal'cev term problems is originally that they're of interest to mathematicians [4]. In this paper, however, we chose them as benchmarks because of their difficulty and formal elegance.

Table 1. Success rate (×100) of best-of-run individuals, averaged over 30 evolutionary runs. Bold marks the best result for each benchmark.

α	Cmp6	Cmp8	Maj6	Mux6	Par5	Dsc1	Dsc2	Dsc3	Dsc4	Dsc5	Mal1	Mal2	Mal3	Mal4	Mal5
	SFIMX-full ($k = \lvert T\rvert$)														
1.0	50	0	62	**100**	0	7	3	33	0	3	90	63	87	23	93
0.9	37	3	50	**100**	3	0	13	30	0	17	97	63	77	33	93
0.8	50	7	70	**100**	10	3	17	50	0	10	90	77	83	33	**100**
0.7	70	0	73	**100**	7	3	7	57	0	3	87	90	97	53	87
0.6	77	3	80	**100**	**17**	3	13	67	0	17	93	83	97	40	97
0.5	73	3	**83**	**100**	0	10	3	67	0	13	97	77	97	60	93
0.4	**83**	7	80	**100**	3	7	10	**80**	0	17	90	93	77	53	97
0.3	73	0	77	**100**	0	**17**	13	67	0	**23**	90	**100**	**100**	53	**100**
0.2	70	0	57	**100**	0	13	3	73	0	13	73	93	93	53	**100**
0.1	25	0	8	**100**	0	10	7	40	0	0	60	57	77	33	87
	SFIMX-half ($k = \lvert T\rvert/2$)														
1.0	40	0	40	**100**	0	0	10	20	0	10	**100**	50	80	0	90
0.9	53	3	60	**100**	3	3	3	50	0	0	87	63	73	17	87
0.8	57	**10**	67	**100**	3	0	3	40	0	7	83	63	83	30	90
0.7	77	0	80	**100**	7	3	7	57	0	0	83	70	87	30	90
0.6	80	7	80	**100**	3	0	0	63	0	3	87	90	97	47	**100**
0.5	77	7	77	**100**	0	3	3	70	0	13	83	90	90	**67**	93
0.4	77	**10**	63	**100**	3	7	13	77	**3**	13	97	80	93	60	93
0.3	63	0	60	**100**	0	7	**33**	67	0	10	73	80	90	40	93
0.2	50	3	37	**100**	0	7	7	43	0	3	63	77	83	53	97
0.1	17	0	3	73	0	0	0	13	0	0	50	50	63	20	90
	SFIMX-log ($k = log_2\lvert T\rvert$)														
1.0	27	0	47	**100**	0	0	3	13	0	3	83	47	70	7	73
0.9	47	3	70	**100**	0	0	3	27	0	0	90	40	67	13	83
0.8	50	0	60	**100**	0	0	3	23	0	3	83	53	67	7	77
0.7	70	0	47	**100**	7	7	7	37	0	3	80	50	80	17	93
0.6	57	0	60	**100**	3	7	7	30	0	7	87	67	87	40	97
0.5	70	0	73	**100**	0	0	0	30	0	3	87	63	97	30	90
0.4	73	3	77	**100**	0	3	3	60	0	17	90	80	90	47	97
0.3	67	0	70	**100**	0	7	13	73	0	10	93	83	90	50	**100**
0.2	43	0	43	**100**	0	3	7	57	0	3	70	83	83	57	97
0.1	0	0	0	93	0	0	0	3	0	0	27	20	37	7	73
	RSS														
0.9	65	0	55	95	5	0	0	32	0	0	88	58	62	12	85
0.8	68	0	55	95	0	0	2	35	0	0	82	58	78	15	88
0.7	62	0	65	95	0	0	0	42	0	0	85	55	85	18	88
0.6	68	2	52	92	2	0	0	32	0	5	68	65	82	8	88
0.5	65	0	45	95	0	0	0	45	0	0	68	48	78	8	85
0.4	52	0	55	95	0	0	0	45	0	0	68	48	78	25	85
0.3	58	0	65	92	0	0	0	35	0	0	85	65	82	22	95
0.2	42	0	45	95	0	0	0	35	0	0	62	55	58	22	82
0.1	15	0	0	95	0	0	0	15	0	0	62	45	55	8	85
	GP														
	63	3	63	100	3	0	0	30	0	3	93	54	63	27	93

Performance. Table 1 reports the success rates of particular algorithms, resulting from 30 runs of each configuration on every benchmark. To provide an aggregated perspective on performance, we employ the Friedman's test for multiple achievements of multiple subjects [11]. We first determine the best performing configuration within each method. For SFIMX-full and SFIMX-half, the configurations with $\alpha = 0.4$ fare the best, while for SFIMX-log and for RSS $\alpha = 0.3$ is most advantageous. The Friedman test applied to those configurations leads to the following ranking:

SFIMX-half-04	SFIMX-full-04	SFIMX-log-03	GP	RSS-03
2.07	2.13	2.67	**3.90**	**4.23**

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 [9]: bold font marks the methods that are outranked at 0.05 significance level by SFIMX-half-04.

For additional insight, we also ranked all considered configurations for all individual values of α. The best overall average rank of 7.57 was achieved by SFIMX-half-04. Eight out of ten SFIMX-half configurations ranked before any of the control configurations; only SFIMX-half with $\alpha = 0.1$ and $\alpha = 1.0$ ranked behind GP and some RSS setups (average ranks 32.47 and 24.53, respectively). GP attained average rank 21.43.

SFIMX clearly outperforms the other methods. Its average ranks are better than the ranks of control configurations, albeit not so for the logarithmic variant SFIMX-log. That last fact is not surprising, given that SFIMX-log uses roughly an order of magnitude fewer weights and features than SFIMX-full and SFIMX-half. Nevertheless, SFIMX-log still delivers decent performance and for its preferred setting $\alpha = 0.3$ surpasses GP and RSS on most benchmarks. This corroborates our hypothesis that the interaction outcomes are significantly correlated and lend themselves to high compression without affecting the overall performance of the method. This result is particularly appealing, as low k implies low computational overhead of factorization: for SFIMX-log, it amounts only to approximately 6 percent of the total cost of calculating the $1,000|T|$ program-test interactions.

On the other hand, there are no significant differences in performance between SFIMX-full and SFIMX-half. Apparently the relatively high rank of the resulting matrices makes it possible to model the interaction outcomes sufficiently well in both these cases.

The success rates of SFIMX for individual benchmarks are always the best among the considered methods – see the values marked in bold in Table 1. For SFIMX-half, the SFIMX variant that overall fares the best, for three problems (Mux6, Mal1, and Mal5) there is at least one setting of α that makes SFIMX succeed systematically, i.e., in every run (success rate 100). In that respect, it is equaled only by GP and only on the Mal5 problem.

SFIMX performs also well in qualitative terms. It manages to produce solutions for all problems, while GP never solves Cmp8, Dsc1, Dsc2, Dsc4 and Dsc5, and RSS never solves Dsc1 and Dsc4, and hardly ever solves Cmp8, Dsc2 and Dsc5. On those hard problems, SFIMX is in most cases remarkably resistant to the setting of α: for Cmp8 and Dsc1, it succeeds for most values of α in the range $[0.2, 0.9]$, and for Dsc5 for $\alpha \in [0.2, 1.0]$. The only exception is Dsc4 where it managed to solve the problem only for $\alpha = 0.4$, and only once in 30 runs.

As a rule of thumb, we may say that setting α in $[0.3, 0.7]$ is favorable. However, using other values is not very detrimental. For many problems SFIMX maintains decent success rates even for very low setting of this parameter; for instance SFIMX-half is better than or comparable to GP for $\alpha = 0.2$ on Cmp8, Mux6, Dsc1, Dsc2, Dsc3, Dsc5, Mal2, Mal3, Mal4, and Mal5. For some benchmark, it still works quite well even for $\alpha = 0.1$. This is impressive, given that the interaction matrix is reconstructed there from only 10 percent of actual outcomes of program-test interactions.

7 Conclusions and Future Work

In conclusion, we find the idea of reconstructing interaction outcomes via factorization of sparse interaction matrix both conceptually appealing and useful in practice. SFIMX is straightforward, founded on solid mathematical underpinnings, and performs well for a broad range of values of parameters α and k. We assumed here that the algorithm spends the spared evaluation cycles on additional programs in extended population. Obviously, nothing precludes other designs, i.e., extending evolution with additional generations or simply completing a run in a shorter time.

Applicability of SFIMX reaches beyond GP. In general, interaction matrices produced in any test-based problems can be subject to the proposed processing. This applies in particular to interactive domains typically solved with competitive coevolution algorithms. Examples include two-player games, evolution of robot controllers, and abstract problems like density classification task, a classical problem in cellular automata.

In the form presented in this paper, SFIMX deliberately discards certain interactions. However, it might be used in scenarios where G is sparse due to other, more objective and external reasons. The most obvious example are the problems with an infinite or very large number of tests. Many control problems belong to this category. Even in the discrete domains like artificial ant or density classification task, the numbers of possible environments (or initial conditions) are often astronomical, not mentioning the continuous domain with problems like inverted pendulum. In such problems, tests (environments) can be generated on demand, and the interaction function performs agent's simulation in an environment and is thus computationally costly. SFIMX's capability of filling in the missing interaction outcomes can be in such cases invaluable. This preliminary study can be extended in multiple directions. For instance, here we applied SFIMX to discrete domains only; its usefulness in continuous domains typical for symbolic regression is an open question. The required adaptation concerns mapping the – in general arbitrary large – continuous error to interaction outcomes.

We hypothesize that simple transformation with some squeezing function (e.g., sigmoidal function or hyperbolic tangent) may be appropriate for that purpose.

Acknowledgements. 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.

References

1. Berry, M.W., Browne, M., Langville, A.N., Pauca, V.P., Plemmons, R.J.: Algorithms and applications for approximate nonnegative matrix factorization. Comput. Stat. Data Anal. **52**(1), 155–173 (2007)
2. Bucci, A., Pollack, J.B., de Jong, E.: Automated extraction of problem structure. In: Deb, K., Tari, Z. (eds.) GECCO 2004. LNCS, vol. 3102, pp. 501–512. Springer, Heidelberg (2004)
3. Chong, S.Y., Tino, P., Ku, D.C., Xin, Y.: Improving generalization performance in co-evolutionary learning. IEEE Trans. Evol. Comput. **16**(1), 70–85 (2012)
4. Clark, D.M.: Evolution of algebraic terms 1: term to term operation continuity. Int. J. Algebra Comput. **23**(05), 1175–1205 (2013)
5. de Jong, E.D., Pollack, J.B.: Ideal evaluation from coevolution. Evol. Comput. **12**(2), 159–192 (2004)
6. Gonçalves, I., Silva, S., Melo, J.B., Carreiras, J.M.B.: Random sampling technique for overfitting control in genetic programming. In: Moraglio, A., Silva, S., Krawiec, K., Machado, P., Cotta, C. (eds.) EuroGP 2012. LNCS, vol. 7244, pp. 218–229. Springer, Heidelberg (2012)
7. Helmuth, T., Spector, L., Matheson, J.: Solving uncompromising problems with lexicase selection. IEEE Trans. Evol. Comput. **19**(5), 630–643 (2015)
8. Hildebrandt, T., Branke, J.: On using surrogates with genetic programming. Evol. Comput. **23**(3), 343–367 (2015)
9. Hollander, M., Wolfe, D.A., Chicken, E.: Nonparametric Statistical Methods, vol. 751. Wiley, New York (2013)
10. Jin, Y., Olhofer, M., Sendhoff, B.: A framework for evolutionary optimization with approximate fitness functions. IEEE Trans. Evol. Comput. **6**, 481–494 (2002)
11. Kanji, G.K.: 100 Statistical Tests. Sage, London (2006)
12. Koren, Y., Bell, R., Volinsky, C.: Matrix factorization techniques for recommender systems. Computer **8**, 30–37 (2009)
13. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
14. Krawiec, K.: Behavioral Program Synthesis with Genetic Programming. Springer, Switzerland (2015)
15. Krawiec, K., Liskowski, P.: Automatic derivation of search objectives for test-based genetic programming. In: Machado, P., Heywood, M.I., McDermott, J., Castelli, M., García-Sánchez, S., Sim, K. (eds.) EuroGP 2015. LNCS, vol. 9025, pp. 53–65. Springer International Publishing, Switzerland (2015)
16. Krawiec, K., O'Reilly, U.: Behavioral programming: a broader and more detailed take on semantic GP. In: Proceedings of the 2014 Conference on Genetic and Evolutionary Computation, pp. 935–942. ACM (2014)

17. Krawiec, K., Solar-Lezama, A.: Improving genetic programming with behavioral consistency measure. In: Bartz-Beielstein, T., Branke, J., Filipič, B., Smith, J. (eds.) PPSN 2014. LNCS, vol. 8672, pp. 434–443. Springer, Heidelberg (2014)
18. Lee, D.D., Seung, H.S.: Algorithms for non-negative matrix factorization. In: Advances in Neural Information Processing Systems, pp. 556–562 (2001)
19. Liskowski, P., Krawiec, K.: Discovery of implicit objectives by compression of interaction matrix in test-based problems. In: Bartz-Beielstein, T., Branke, J., Filipič, B., Smith, J. (eds.) PPSN 2014. LNCS, vol. 8672, pp. 611–620. Springer, Heidelberg (2014)
20. Mao, Y., Saul, L.K.: Modeling distances in large-scale networks by matrix factorization. In: Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement, pp. 278–287. ACM (2004)
21. McKay, R.I.B.: Committee learning of partial functions in fitness-shared genetic programming. In: 26th Annual Conference of the IEEE Third Asia-Pacific Conference on Simulated Evolution and Learning 2000, Industrial Electronics Society, IECON, 22–28 October 2000, vol. 4, pp. 2861–2866. IEEE Press, Nagoya, Japan (2000)
22. McKay, R.I.B.: Fitness sharing in genetic programming. In: Whitley, D., Goldberg, D., Cantu-Paz, E., Spector, L., Parmee, I., Beyer, H.G. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000), 10–12 July 2000, pp. 435–442. Morgan Kaufmann, Las Vegas (2000)
23. Moraglio, A., Krawiec, K.: Semantic genetic programming. In: Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference, pp. 603–627. ACM (2015)
24. Moraglio, A., Krawiec, K., Johnson, C.G.: Geometric semantic genetic programming. In: Coello, C.A.C., Cutello, V., Deb, K., Forrest, S., Nicosia, G., Pavone, M. (eds.) PPSN 2012, Part I. LNCS, vol. 7491, pp. 21–31. Springer, Heidelberg (2012)
25. Paatero, P., Tapper, U.: Positive matrix factorization: a non-negative factor model with optimal utilization of error estimates of data values. Environmetrics **5**(2), 111–126 (1994)
26. Smith, R.E., Forrest, S., Perelson, A.S.: Searching for diverse, cooperative populations with genetic algorithms. Evol. Comput. **1**(2), 127–149 (1993)
27. Spector, L., Clark, D.M., Lindsay, I., Barr, B., Klein, J.: Genetic programming for finite algebras. In: Keijzer, M. (ed.) Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation, GECCO 2008, 12–16 July 2008, pp. 1291–1298. ACM, Atlanta (2008)