Semantically Embedded Genetic Programming: Automated Design of Abstract Program Representations

Krzysztof Krawiec
Institute of Computing Science
Poznan University of Technology
Piotrowo 2, 60965 Poznań, Poland
krawiec@cs.put.poznan.pl

ABSTRACT

We propose an alternative program representation that relies on automatic semantic-based embedding of programs into discrete multidimensional spaces. An embedding imposes a well-structured hypercube topology on the search space, endows it with a semantic-aware neighborhood, and enables convenient search using Cartesian coordinates. The embedding algorithm consists in locality-driven optimization and operates in abstraction from a specific fitness function, improving locality of all possible fitness landscapes simultaneously. We experimentally validate the approach on a large sample of symbolic regression tasks and show that it provides better search performance than the original program space. We demonstrate also that semantic embedding of small programs can be exploited in a compositional manner to effectively search the space of compound programs.

Categories and Subject Descriptors

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

General Terms

Algorithms, Design, Experimentation

Keywords

genetic programming, genotype-phenotype mapping, locality, program representation, program semantics

1. INTRODUCTION

In Genetic Programming (GP), the relationship between a program (code, syntax) and its effect (semantics, behavior) is much more complex than in conventional tasks considered in evolutionary computation, where the phenotype typically depends on genotype in a more direct way. Even a minute change of program code can radically alter its semantics and fitness, which makes GP fitness landscapes very ragged. This has serious implications for the evolutionary search process, both in local perspective (exploitation) as well as in the global one (exploration). Low *locality* of genotype-phenotype mapping makes it particularly hard to design mutation operators that have small effect on fitness [1]. On the other hand, the lack of fitness-distance correlation causes only a small fraction of crossovers to increase fitness [14, 2]. This hampers scalability of GP search, and limit its applicability to difficult real-world problems.

Agreeing with this state of matter would mean no prospects for automated programming, so significant effort has been put into search for countermeasures. As the primary determinant of locality is the adopted neighborhood definition, most work focused on designing new search operators that implicitly redefine the notion of neighborhood in program space. In the case of mutation, operators could optimize the values of constants, either at random [13] or in a directed way [7]. For crossover, one can preserve the locations of code fragments being exchanged, e.g., context-preserving crossover or homologous crossover (see [10] for review). Thus, most of past research approaches the problem from the symbolic perspective, by redesigning the way in which the search operators manipulate the program code, and usually doing so in abstraction from the actual meaning (semantics) of those symbols. Only recently, semantic-oriented approaches gained more attention [8, 11, 4, 5].

While there are many reasons to believe that representing programs by structures of symbols is convenient for human programmers, it is not necessarily the best choice for an automated search process. Symbolic representations lack a natural and semantically coherent similarity relation. They are redundant in a way that is difficult to capture – for instance, reversals of instruction order can be neutral for program outcome. At the same time, they can be also highly epistatic – distant code fragments often strongly interact with each other, with complex effects on program semantics.

In this study, we redefine the task of automated programming by embedding programs into an abstract multidimensional space in such a way that semantically similar programs are likely to have near locations. This concept leads to a new program representation, where the original program code is replaced by its coordinates in the new space. We show the embedding be effectively optimized with respect to locality, so that the process of automated programming in the new space can be more effective. Finally, we also demonstrate how these concepts can be exploited in a compositional way, improving the scalability of GP.

GECCO'11, July 12–16, 2011, Dublin, Ireland.

2. SEMANTIC MAPPING AND LOCALITY

Let P be a *program space,* and let $s(p)$ define the *semantics* of program $p \in P$, where $s : P \to S$ is the *semantic mapping* (a.k.a. genotype-phenotype mapping) and S is a metric *semantic space*. For the rest of this paper, we assume that s is surjective (i.e., $S \equiv image(s)$). Typically, s is also not invertible and $|P| \gg |S|$, where $|S|$ can be termed as *semantic diversity* of the program space P. By assuming that s is endowed with the entire knowledge required for program execution, we abstract here from program syntax and the specifics of its interpretation.

We define *problem instance* as a program space augmented by a fitness function $f : P \to \mathbb{R}$. Multiple problem instances can be defined for a given program space P.

Let $N(p)$ be the *neighborhood* of p ($N : P \to 2^P \setminus \{\emptyset\}$, $p \notin N(p)$). Typically, $N(p)$ is the set of all programs that can be obtained by introducing small changes in p, or, in EC terms, by mutating p (e.g., substituting one instruction with another). In practice, $N(p)$ should be cheap to compute. The pair (P, N) is also referred to as *configuration space*, and the triple (P, N, f) as *fitness landscape* (see, e.g., [9]).

We propose to measure the locality in the neighborhood of program p in the following way:

$$l(N, p, s) = \frac{1}{|N(p)|} \sum_{p' \in N(p)} \frac{1}{1 + \|s(p') - s(p)\|} \qquad (1)$$

where $\|\|$ denotes a metric in S. The rationale for this definition is at least twofold. First, whatever range of values $\|\|$ assumes, the formula maps them to the interval $(0, 1]$. Second, a single neighbor cannot dominate the value of l, as its maximal contribution is $1/|N(p)|$. l reaches 1 when all neighbors of p have the same semantics as p. l close to zero implies that all neighbors of p are semantically very different from it.

The value of l for a particular program is of little importance from the viewpoint of search algorithm, given that $|P|$ is typically large. Being interested in locality of semantic mappings, we generalize this concept to the entire program space P, defining *semantic locality* (or *locality* for short):

$$L(N, P, s) = \frac{1}{|P|} \sum_{p \in P} l(N, p, s) \qquad (2)$$

Semantic locality is is a joint property of the configuration space (P, N) and semantic mapping s. Obviously, $L \in (0, 1]$, but $L = 1$ holds only if all programs in P have the same semantics, so it should not be expected in practice.

High locality is desirable, as it indicates smooth fitness landscape that is easier to explore for search algorithms. This is particularly evident when the fitness of p is based on a distance between $s(p)$ and some *target* $t \in S$, like a vector of desired program outputs in symbolic regression and logical function synthesis. In such cases, which we focus on here, one can define a (minimized) fitness as $f(p) = \|s(p) - t\|$, and the link between locality and smoothness is a direct consequence of triangle inequality. For any two programs p_1 and p_2, $s(p_1)$, $s(p_2)$, and t form a triangle in the semantic space and thus

$$\|s(p_1) - s(p_2)\| \geq |f(p_1) - f(p_2)|$$

In other words, the difference in fitness between any two solutions cannot be greater than their mutual semantic distance. This applies to any pair of programs (p_1, p_2), including such that $p_1 \in N(p_2)$ or vice versa (N does not have to be symmetric). Therefore, by improving locality, we decrease the semantic discrepancy between neighboring programs and increase smoothness of the fitness landscape. This holds simultaneously for *all* fitness landscapes, and does not depend on whether t is explicitly known.

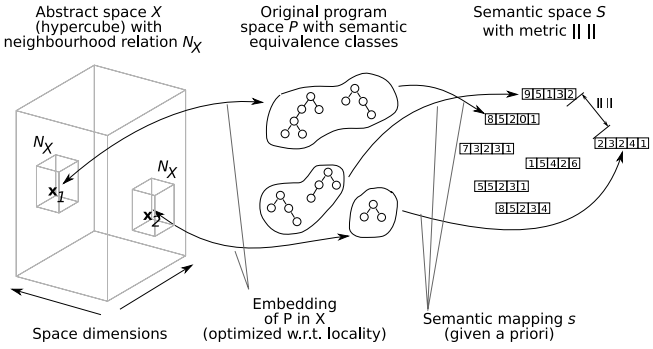

Figure 1: Conceptual sketch of semantic embedding.

Smooth fitness landscapes tend to have fewer local optima that are the main nuisance for many optimization algorithms. Though this is particularly true for local search algorithms that rely exclusively on neighborhoods, exploitation ability of global search algorithms, including evolutionary algorithms, can be also sensitive to this aspect, as the experimental part of this paper shows. On the other hand, it is worth emphasizing that high locality does not *guarantee* good search performance – easy problems typically have high locality, but the reverse is in general not true.

The notion of locality can be alternatively based directly on fitness function instead of semantics (see, e.g., [12]). In that perspective, locality is high if the neighboring programs have similar fitness values. Such formulation is not concerned with how fitness is derived from program code and, in particular, does not assume the existence of semantic space. However, by the same token it is very problem-specific: conclusions concerning locality of one problem instance cannot be generalized to other problem instances.

3. SEMANTIC EMBEDDING

Design of search operators reviewed in Introduction is a 'program-centric' methodology of locality improvement, which focuses on building a new candidate solution from one or more given solutions. In this study, we propose a different, 'space-centric' direction (Fig. 1). Rather than designing new operators for the existing space of programs P (central part of Fig. 1), we switch to another space (left part of Fig. 1) and try to build a mapping from that space onto P, such that its composition with the semantic mapping provides high locality in the semantic space (right part of Fig. 1).

More formally, our objective is to design three formal objects: (i) a space X, which we name *prespace* by analogy to 'pre-image', (ii) a neighborhood relation N_X in that space, and (iii) a mapping u from X to P, such that together they maximize $L(N_X, X, s \circ u)$. To clearly tell apart P from X, we rename P as the *original* program space.

Before we proceed, let us first note that because locality cannot distinguish semantically equivalent programs, it does not matter which of them is returned by u for a given $x \in X$. More formally, let $[p] = \{p' \in P : s(p') = s(p)\}$ be

the *semantic equivalence class* of program p. If $u(x) = p$, then as long as we redefine u in such a way that $u(x) \in [p]$, the superposition $s(u(x))$ points to the same semantics and locality remains unchanged. Thus, it is enough to consider as codomain of u the space $[P]$ of semantically unique *representatives* of programs in P, defined as

$$[P] \subseteq P : \forall_{p \in P} \exists_{p' \in [P]} s(p') = s(p) \wedge \forall_{p_1, p_2 \in [P]} s(p_1) \neq s(p_2)$$

This property is important from practical perspective, because $|[P]| = |S|$, while typically $|P| \gg |S|$, so considering only the representatives is computationally less demanding. Thus, in following $u : X \to [P]$.

As there are many ways in which X, N_X, and u can be defined, our task is clearly underconstrained. To narrow the spectrum of possible designs, we limit our considerations to bijective mappings u, which implies that $|X| = |[P]|$. In such case u is a permutation of X and can be considered as a form of *embedding* of $[P]$ in X.

Concerning the topology of X, it seems convenient to endow it with a natural neighborhood definition. For that purpose, we assume X to be a discrete d-dimensional hypercube of size $n \geq 2$, i.e., $X \equiv [1, n]^d \cap \mathbb{N}^d$, so that any $\mathbf{x} \in X$ is a vector of coordinates determining a particular location in X. The assumption $n \geq 2$ guarantees that each coordinate of the hypercube differentiates some elements. Of course, for a given d, there does not have to exist an integer n such that $n^d = |[P]|$, but we will handle this technicality in the experimental part. We define the neighborhood $N_X(\mathbf{x})$ as the set of locations in X such that their city-block distance from $\mathbf{x}$ is less than r, where we assume that the distance is calculated modulo n on each dimension, so that the hypercube is effectively toroidal.

The above choices fix X and N_X, so design of the triple (X, N_x, u) boils down to finding a permutation u that maximizes $L(N_X, X, s \circ u)$. In other words, we want to *optimize the embedding* of $[P]$ in X, finding such an optimal embedding u^* such that $s \circ u^*$ is the best locality-preserving mapping from X to S via P:

$$u^* = \arg\max_u L(N_X, X, s \circ u) \tag{3}$$

It is easy to notice that for $d = 1$ the above problem becomes equivalent to the traveling salesperson problem, where u defines a particular order of 'visiting' the elements of $[P]$. It is then NP-hard for $d = 1$, and it is quite easy to demonstrate that this holds also for $d > 1$. Finding u^* via exhaustive search is futile even for relatively small $|[P]|$, as the sheer number of possible permutations renders considering all embeddings infeasible. This is however not critical: we demonstrate in the following that even a suboptimal embedding found by a heuristic algorithm (Section 4) can improve locality and benefit the search performance.

Finally, it is worth pointing out that X and N_X implicitly define a new program representation, where each semantic equivalence class of programs is identified by a d-tuple of coordinates in the hypercube. This is the space we ultimately delegate the search process to (Section 5). In general, this representation is detached from program code, or more precisely, from the symbols (instructions) that the considered programs are composed of. Neighbors in X can correspond to syntactically very different programs in P. Program space P becomes transparent from the viewpoint of X, and serves as a proxy for reaching into the semantic space S (Fig. 1). Nevertheless, despite that abstraction, this approach preserves the, characteristic for GP, explicability, as a solution expressed as a point in the abstract space X can be always traced back to the corresponding concrete program in P (more precisely, the semantic equivalence class of programs).

4. IMPROVING LOCALITY OF EMBEDDING

In [3] we proposed two algorithms to find an approximate solution to problem (3). Here, we rely on a different iterative local search method that turns out to be much better. We propose a greedy search heuristic for optimizing locality of embedding that finds an approximate solution for problem (3). For clarity, rather than referring to mappings, we explain it in terms of embeddings, i.e., different permutations of program representatives in X.

The algorithm starts with the hypercube X filled up with randomly ordered elements of $[P]$, and then iterates over the consecutive locations in X, column by column, row by row, etc. For each location $\mathbf{x}$, it considers every $\mathbf{x}' \in N(\mathbf{x})$, temporarily swaps the programs located in $\mathbf{x}$ and $\mathbf{x}'$, records the change of locality L, and then retracts the move. After looking up the entire neighborhood, if any move increased L, then the move that led to the greatest increment is executed. As a single pass of this procedure over the entire hypercube cannot be expected to substantially improve L, we repeat it multiple times, recording in each step the total improvement of L, and terminate if it drops below an assumed floating-point precision level (here: 10^{-8}). Typically, less than 20 iterations are sufficient for the algorithm to converge.

The complexity of the algorithm depends on three parameters: the neighborhood size $|N|$, the semantic diversity $|S|$, and the number of iterations k. Because L is defined additively (Eq. (2)), each move requires updating l only for the direct neighbors of the swapped programs, and there is no need to globally recalculate L. This makes the algorithm fast for reasonably sized neighborhoods. Full scan of one neighborhood requires then $|N|^2$ calculations of semantic distance between pairs of solutions, and the overall complexity is $k|S||N|^2$. The major factor that impacts algorithm's runtime is therefore d, as it directly determines $|N|$, in a way depending on the assumed neighborhood definition. Note also that, if $|S|^2$ is reasonably small and computation of semantic distance is costly, it can pay off to cache distances between all pairs from S prior to optimization.

Figure 2 visualizes an example of a two-dimensional embedding of arithmetic programs composed of symbols $\{+, -, *, /, v, 1\}$, where v is the independent variable. We use the same setup as in the forthcoming experiment, however consider only 1024 programs represented by trees of depth up to three (3 nonterminals and 4 terminals). Given 20 values of v drawn randomly from the interval $[-1, 1]$, this set of programs produces 132 unique semantics depicted by the plots (clamped at top and bottom). The figure presents the embedding optimized using the proposed algorithm, with the shadings between the plots reflecting the semantic distance: the darker the grade, the greater the distance between the neighbors (the embedding is toroidal). Locality of this embedding amounts to 0.848, compared to 0.742 for a random embedding. The effect of optimization is clearly visible: similar plots group in close proximity.

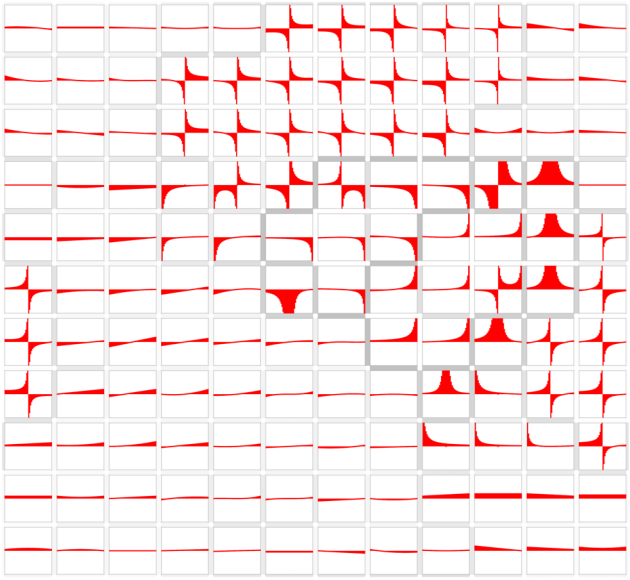

Figure 2: An exemplary optimized two-dimensional toroidal embedding of all 132 unique semantics obtained from 1024 programs of depth 3 composed of instructions $+, -, *, /, v, 1$. Each tile plots the function calculated by the corresponding representative program (its semantics) for $v \in [-1, 1]$. The darker the shading between the neighboring tiles, the greater the semantic distance. The locality of this embedding is higher than that of random embedding, which could be obtained by reshuffling the tiles randomly (see text for more details).

5. THE EXPERIMENT

In the experiment we want to verify the usefulness of semantic embedding for evolutionary search. In [3] we showed that optimized embeddings provide better performance of a local search algorithm. Here, we use semantic embedding for global evolutionary search and apply it in compositional manner to evolve larger programs.

Program space. We consider univariate symbolic regression with two terminal symbols (the constant 1.0 and the independent variable v) and four arithmetic operators $\{+, -, *, /\}$ as internal tree nodes (non-terminals). The semantics is a vector of outputs produced by a program for 20 values of the independent variable (fitness cases), drawn at random from the interval $[-10, 10]$. We use the Euclidean distance as a metric in S and round it off to 0 whenever it drops below 10^{-15}.

We anticipated the expected runtime in a series of preliminary experiments and decided to constrain our program space to binary GP trees with depth up to 4. With the assumed set of instructions, any tree smaller than a full binary tree of depth 4 can be expanded to a semantically equivalent full tree by replacing some occurrences of the terminal v with a subtree `(* 1 v)`, and occurrences of `1` with `(* 1 1)`. This feature, though not essential for our approach, allows constraining the original program space to $|P| = 4^7 \times 2^8 = 4,194,304$ syntactically unique full-tree programs (7 nodes + 8 leaves).

Table 1: The cardinality of the neighborhood $|N|$, mean hypercube size $\overline{n}$, locality L of random embedding, and locality of optimized embedding with 0.05 confidence intervals, for different prespace dimension d.

d	$\lvert N\rvert$	$\overline{n}$	Random L	Optimized L
2	8	121.8	0.799	0.910±0.008
3	18	25.0	0.781	0.888±0.009
4	32	11.8	0.783	0.881±0.005
5	50	7.0	0.790	0.871±0.002
6	72	5.0	0.793	0.862±0.009
7	98	4.0	0.795	0.857±0.010
9	162	3.0	0.795	0.854±0.010

Semantic space. The number of unique semantics (semantic variability) $|S|$ that this set of programs produces is much lower and varies from around 14,670 to 14,681, with the average of 14,673, depending on the particular set of 20 randomly generated fitness cases. This is also the number of program representatives $[P]$, which we embed in a d-dimensional toroidal hypercube X for $d = 2, 3, 4, 5, 6, 7, 9$. As $n \geq 2$ has to hold, the largest valid d is here $\lceil \lg_2 14,673 \rceil = 14$. However, for large d the neighborhood effectively ceases to be local (see Table 1) and the optimization process starts to be computationally quite demanding, so $d \geq 10$ has not been considered here.

Because the cardinality of the semantic space is quite accidental, it typically cannot be expressed as n^d for integral d and n, so that $|X| = |[P]| = |S|$ holds. To solve this difficulty, we use the smallest n such that $n^d \geq |[P]|$. For instance, given $d = 3$ and $|[P]| = 41$, we would set n to 3, as $3^3 < 41 < 4^3$. Then, we would fill the hypercube with programs column by column, row by row, etc., allowing the final $64 - 41 = 23$ locations to remain vacant. As a result, the two first two-dimensional layers would be completely filled-up ($2 \times 4^2 = 32$), the third one would be occupied by $41 - 32 = 9$ programs, and the last one would be empty. Finally, we toroidally connect the last occupied locations with the first layer. This procedure preserves the d-dimensional topology of the space, except for the degenerate case when *all* representatives turn out to fit into the first $d-1$ dimensional layer of the hypercube, i.e., when $(n-1)^d < |[P]| \leq n^d$ and $|[P]| < n^{d-1}$. In our experiment, this happens for $d = 8$, which explains its absence.

We use the city-block distance sphere of radius $r = 2$ as the neighborhood N_X. This radius is a compromise between the expected gains in locality elaborated by the optimization algorithm (the greater r, the more thorough the search), and the computational cost of optimization – neighborhood size grows exponentially with r, and even for the modest $r = 2$, the size of the neighborhood depends quadratically on d ($|N| = 2d + 2d(d-1) = 2d^2$), so for large d it may be quite numerous, as Table 1 proves.

5.1 Optimization of embeddings

We optimize embedding for $d = 2, \ldots, 7, 9$ using the local search algorithm described in Section 4. The fourth and fifth columns of Table 1 present the locality L respectively before the optimization (i.e., for a random assignment of programs to locations), and after the embedding has been optimized.

The results are averaged over 20 starting points of the algorithm. The algorithm turned out to be quite insensitive to initial conditions, typically converging to embeddings that have very similar L. Its runtime on an up-to-date PC varied from about 20 seconds for $d = 2$ to 15 minutes for $d = 9$.

The first observation is that locality prior to optimization varies only slightly with d, which is obvious knowing that the initial embedding of programs is random. But more importantly, our algorithm clearly improves locality for each considered dimensionality. Given that $L \in (0, 1]$ by definition, the increases do not look very impressive, but such judgments are pointless without knowing what is the locality of an optimal embedding for a given d. Also, the locality of optimized embedding decreases slightly with d, which is probably an effect of interplay between two factors. On one hand, larger neighborhood makes the algorithm consider more program swaps, so finding a good embedding becomes more likely. On the other, it also leads to more interactions between neighbors, all of which contribute to L (cf. Formula (1)), so in this sense the optimization process faces more constraints. It is then tempting to hypothesize that in our experiment the latter factor prevails, but this phenomena involves also the distribution of semantics in S and is in general more complex, so we leave this issue open.

5.2 Search in an optimized embedding

We assume now a more practical perspective and try to find out what is the impact of embedding optimization for the actual performance of a search algorithm. To this aim, we empirically compare the performance of evolutionary runs that use the non-optimized and optimized embeddings on a sample of different problem instances.

Methodology. In both cases, we launch a series of independent evolutionary runs driven by different fitness functions. In a single run, a random target $t \in S$ (problem instance, fitness function) is first selected, a random population is created, and then an evolutionary algorithm driven by that fitness function runs until an ideal solution is found or 100 generation elapse, whatever comes first. The former case is recorded as success, the latter as failure. An ideal found in the initial generation is not counted as a success and the run is relaunched. This process is repeated 300 times for 10 different embeddings (random or optimized, depending on the setup), and the overall success rate for the $3,000$ problem instances becomes our performance indicator.

The process of generating targets deserves comments. As we assumed that S is an *image* of s, drawing targets from S means considering only problem instances that are *solvable* with the assumed instruction set and program length limit. Secondly, targets are drawn from S generated for the set of fitness cases specific for particular evolutionary run, which is in general different than the set used by the embedding optimization algorithm. Thirdly, we use only non-trivial problem instances, i.e., such that the output of the program that produced t depends on the value of the input variable v (which we verify by checking if there is any variance in program output over fitness cases). Finally, we care for the distribution of targets, which is essential, as the distribution of semantics over programs in P is highly skewed. For instance, 50.7% of programs in our program space generate the top 100 most frequent semantics (of the total of over $14,600$), and 84.4% generate the top 1000. In this light, generating problem instances uniformly in P would bias the results by strongly favoring the 'easy' problems. To avoid that, we draw targets *uniformly* from S, assuming that this reflects better the unknown distribution of real-world problem instances.

Evolutionary setup. We employ generational evolutionary algorithm with a population of 100 individuals, each being a vector $\mathbf{x}$ of d integers that encodes a program representative by pointing to a specific location in the d-dimensional hypercube. The elements of $\mathbf{x}$ (genes) are clamped to interval $[1, n]$, where n is the hypercube size (cf. Table 1). A new individual is bred from parents selected from the the previous population using tournament of size 7. Breeding takes place in one of two alternative ways: via mutation with probability p_m, or via crossover with probability $1 - p_m$, where we consider $p_m \in \{0, 0.2, 0.4, 0.6, 0.8, 1.0\}$. Mutation operates in a twofold way: with probability 0.9 it increments or decrements by one (modulo hypercube size) a single, randomly selected gene in an individual, or with probability 0.1 it resets the entire genome by drawing new random hypercube coordinates. We introduce the latter mode because, in absence of crossover (the case $p_m = 1$), there would be no way to escape the local optima. The crossover is geometric: given two parents representing locations (coordinates) $\mathbf{x}_1$ and $\mathbf{x}_2$ in the hypercube, the offspring is a randomly selected location on the Euclidean segment connecting $\mathbf{x}_1$ and $\mathbf{x}_2$, with coordinates rounded off to the discrete grid. Because of the toroidal topology, there are 2^d distinct segments that connect $\mathbf{x}_1$ with $\mathbf{x}_2$ (unless $\mathbf{x}_1 = \mathbf{x}_2$), of which we choose the shortest one.

This design of search operators provides for fair comparison of evolutionary runs for different d. In particular, we should emphasize that our 'unitary' mutation employs neighborhood defined as a Hamming-distance sphere of radius 1, not the much bigger N_X neighborhood that has been used exclusively for optimization of embeddings. In this sense, it performs a very conservative local search.

The results. Table 2 presents the mean success rate for the random and optimized embeddings. The figures clearly suggest that the latter ones provide better 'searchability' for all considered combinations of d and p_m, which suggests that locality is an appropriate and effective objective for optimization of embeddings. The superiority of the optimized embeddings is statistically significant for all combinations of d and p_m, except for $(2, 0.0)$. The increase of success rate with d does not correlate with the final locality of the optimized embeddings (the last column of Table 1), but this should not be expected given that the upper bound on locality depends on dimensionality.

The highest success rates are attained for the greatest considered dimensionality, so it is tempting to use an even higher d. There are however some counterarguments. The expected city-block distance of two random locations in a d-dimensional toroidal hypercube containing $|S|$ elements is $l = \frac{d}{4}\left\lceil \sqrt[d]{|S|} \right\rceil$. This is also the average number of unitary mutations that an individual from the initial population must undergo to reach the target. For $|S| \cong 14,600$, l happens to reach the global minimum of 6.75 for $d = 9$. Therefore, using $d > 9$ increases the expected distance from the target and could have negative impact on search performance. This reasoning should be however taken with a grain of salt, as it ignores the structure of the fitness landscape.

Table 2: Success rate (%) of EA evolving programs on a sample of $3,000$ problem instances for the random and optimized embeddings.

p_m	0.0	0.2	0.4	0.6	0.8	1.0
d	*Random embedding*					
2	5.7	7.1	9.4	9.2	10.5	11.0
3	3.9	5.4	7.1	10.6	12.3	12.2
4	2.6	4.0	7.6	9.1	12.0	13.9
5	2.1	4.5	7.3	11.1	12.6	19.2
6	2.1	5.0	7.9	10.5	16.4	28.1
7	1.3	4.9	6.5	11.7	18.2	32.1
9	1.6	4.1	6.9	12.1	21.5	32.9
d	*Optimized embedding*					
2	6.3	12.1	16.4	21.4	23.6	28.9
3	7.0	20.5	24.7	32.3	36.2	43.4
4	9.4	21.7	34.9	41.0	55.0	65.2
5	7.5	24.5	33.7	45.2	61.3	74.3
6	6.3	25.3	33.5	46.3	65.7	80.9
7	6.7	25.1	34.2	48.4	70.1	83.1
9	6.2	30.3	42.6	54.6	76.9	87.8

Considering the contributions of mutation and crossover, the latter one seems to be more a hindrance than a help: the success rate is the higher, the higher the mutation probability, and for $p_m = 1.0$ and $d = 9$ the search becomes most effective. This was expected, as the ability of the Euclidean crossover to produce novel solutions in a discrete space is quite limited, in particular for high d, when each gene can take on only a few possible values (note that n drops to as low as 3 for $d = 9$). Nevertheless, our motivation for using this particular operator becomes clear when analyzing the column $p_m = 0$ in Table 2: when used as the only search operator, crossover also improves the success rate (except for $d = 2$). This suggests that high locality makes it more likely for a fitness landscape to be not only smooth, but also convex. This observation pertains to other potential variants of semantic embeddings, which we discuss in Conclusions.

In a broader perspective, applying evolutionary algorithm in this particular scenario is not the best choice from practical viewpoint, because the number of possible semantics is here rather low. The $10,000$ evaluations that each of our runs carried out could be alternatively used up for systematic search, which would then reach the optimal solution with probability $10,000/14,600 \approxeq 0.68$, so not much less than the best numbers reported in Table 2 for the optimized embeddings. Nevertheless, our point here was to demonstrate the impact of optimization of embedding, which will have also some consequences for the next section.

Comparison to standard GP. Finally, we compare these results to the performance of standard tree-based GP. To this aim, we run an analogous experiment to the above one, i.e., we estimate GP's success rate on a sample of $3,000$ targets drawn uniformly from the space of semantics. A population of 100 individuals, each being a GP tree of depth exactly 4, evolves for 100 generations and undergoes only subtree-replacement mutation. If the mutated individual violates the tree depth limit, mutation is repeated up to 100 times, after which the parent individual is discarded and another individual is selected for mutation. Other parameters have the same values as in the main experiment.

With such setup, standard GP attains the success rate of 9.6%, almost an order of magnitude lower than the best figures reported in Table 2. Of course, to some extent this gap is due to the difference in cardinality of the search space, which is greater for GP by over three orders of magnitude ($4,194,304$ programs vs. $14,673$ semantics), and due to the fact that the distribution of semantics in the prespace is uniform. Nevertheless, Table 2 demonstrates that optimization makes the probability of success much higher.

5.3 Embeddings for compound programs

Optimizing an embedding to solve a single problem instance is usually pointless, as the proposed approach requires calculation of semantics of *all* programs in P, so the overall cost of optimization *and* search with the optimized embedding is far greater than the computational effort of direct search in the program space P. Nevertheless, there are at least two other scenarios in which this can pay off. The previous section followed the scenario in which there are *many* problem instances to be solved using the same instruction set. Here, we treat embedding as a means to speed up the search in the space of program *fragments* rather than entire programs. The question we ask is: can an optimized embedding be exploited to effectively solve larger problem instances, i.e., instances for which an optimal solution is a program that does not belong to P? In other words, can the increase of locality elaborated by optimization be extrapolated in a compositional manner?

To answer this question we assume that our goal is to evolve a compound program that can be expressed as a *composition* of two arbitrary programs from P, by which we mean feeding the output of the first program p_1 into the second program p_2 as the value of the input variable v. Another words, $p_2 \circ p_1$ is a tree obtained by substituting all occurrences of terminals v with p_1. This is of course a very specific form of program composition: in particular, if p_2 contains multiple references to v, all of them will be replaced by *the same* program p_1, leading to a compound program that standard GP is unlikely to produce. Nevertheless, this is the simplest non-trivial compositional setup that can be implemented in the proposed framework, and sufficient to validate our hypothesis.

Methodology. We employ an analogous experimental protocol as in Section 5.2, i.e., we estimate the success rate on a sample of $3,000$ problem instances (targets). This time, however, a single problem instance is defined by the semantics of the compound program $p_2 \circ p_1$. For the reasons explained earlier, we assume a uniform distribution of target semantics. Because enumerating all unique semantics of $4,194,304^2$ compound programs is infeasible, we ran $5,000,000$ compositions of random programs p_1 and p_2 and so created a sample of almost a million of unique semantics. This sample contains also semantics of simple programs (i.e., from P), because it may happen that $s(p_2 \circ p_1) \in S$, for instance when p_1 is a single-node program v. Because we do not want this experiment to be contaminated by the performance of methods on simple programs, we discard them from the sample, so finally it contains $972,369$ unique semantics. Then, the $3,000$ evolutionary runs optimize for targets drawn uniformly from the sample. The task for the evolution is to find a compound program (a pair of simple

programs from P) that has the same semantics as the target, using non-optimized or optimized embedding.

An individual's genome $\mathbf{x}$ is now an integer vector of length $2d$, with the first half of the genotype encoding the coordinates in X that determine p_1, and the other half encoding p_2, where X is the embedding optimized for single programs in Section 5.1. Thus, this time the search takes place in the space $X \times X$ of cardinality $14,600^2 \cong 2 \times 10^8$. Other settings remain the same as in Section 5.2.

The results. Table 3 presents performance of EA on this task. The success rate drops by more than order of magnitude compared to Table 2. For the unoptimized embeddings, it exceeds 1% only for a few combinations of d and p_m. The reason for this is not only the far greater cardinality of the search space, but also the fact that deeper trees (longer programs) are by nature harder to evolve than the shallow ones, as manipulations on deep tree fragments are unlikely to influence the semantics of a program. As shown by Langdon [6], the distribution of semantics becomes very nonuniform and approaches a limit with tree growth.

In this context, it is encouraging to observe that the optimized embeddings can help finding the solution several times more frequently, particularly for larger values of d and p_m. This suggest that, at least to some extent, the locality elaborated by the optimization algorithm for 'subprograms' (here p_1 and p_2) improves also the locality of the compound program $p_2 \circ p_1$. There are at least two reasons for which this should be considered particularly interesting. Firstly, because p_1 can produce an arbitrary output, the values fed into p_2 are often outside the range $[-10, 10]$ used for optimization of embedding. This suggests that the optimized embedding is capable to generalize beyond the training set. Secondly, our embeddings are not optimized with respect to the input variable v, but only with respect to the mutual distance between the semantics of programs. It is probable then that we benefit here also from 'natural' smoothness of programs with respect to v, which is quite common in symbolic regression, as the plots in Fig. 2 suggest.

Comparison to standard GP. We confront these results with the performance of standard GP in two ways. In the first control experiment we evolve individuals being *pairs* of GP trees, each constrained to depth 4, which, when evaluated, are being composed in a way described above. The best success rate we were able to obtain by parameter tuning for our semantically uniform sample of $3,000$ targets was 1.83% for both mutation and crossover probability equal to 0.5. The second control experiment was standard GP with parameters quite close to Koza-I setup (single tree, depth limit 17, crossover probability 0.9, mutation probability 0.1). In this case, GP found solution in 1.97% of runs. An optimized semantic embedding for $d = 9$ and $p_m = 1.0$ is thrice more likely to find an optimum.

Table 3: Success rate (%) of EA evolving *compound* programs on a sample of $3,000$ problem instances for the random and optimized embeddings.

p_m	0.0	0.2	0.4	0.6	0.8	1.0
d	*Random embedding*					
2	0.43	0.53	0.67	0.77	1.10	0.87
3	0.40	0.40	0.43	0.83	0.77	0.90
4	0.33	0.77	0.43	0.67	0.87	0.83
5	0.40	0.53	0.83	0.77	1.07	1.33
6	0.33	0.53	0.53	0.90	1.03	1.10
7	0.23	0.53	0.67	0.77	1.20	1.17
9	0.40	0.60	0.63	0.87	0.93	0.80
d	*Optimized embedding*					
2	0.20	0.90	1.50	0.80	1.07	0.73
3	0.23	0.63	0.60	0.87	1.20	0.90
4	0.67	0.73	0.60	1.00	1.50	2.60
5	0.53	0.67	0.77	0.77	1.70	3.50
6	0.30	0.50	1.37	1.13	1.77	5.43
7	0.40	0.50	1.17	1.07	1.93	3.83
9	0.63	0.80	1.27	1.67	3.10	5.80

6. SUMMARY AND CONCLUSIONS

The proposed approach of semantic embedding has three key features. The first is the switching from the original program space to the space of program representatives, which vastly reduces cardinality and provides one-to-one correspondence with the semantic space. The second is the concept of geometric embedding that imposes a well-structured, regular topology on the search space, endows it with a natural neighborhood, and enables convenient search using Cartesian coordinates. Finally, there is the locality-driven optimization which, operating in abstraction from a specific fitness function, smoothes the embedding and, as demonstrated in the experiments, improves the expected search performance on all problem instances simultaneously.

The only prerequisite for applying semantic embeddings to a given domain is the existence of a metric semantic space S. Semantics is present in virtually every application domain of GP, and does not have to form a vector as in this study. A path traversed by an agent (artificial ant, game character, robot) in an environment, a sequence of buy/sell transactions issued by a trading strategy, and an output image produced by an evolved image processing program, are other examples of semantics. Wherever the similarity of such outcomes of programs can be measured, semantic embedding is applicable. Moreover, it is reasonable to expect that for problems where the target semantics t is not known and thus fitness function is not explicitly based on metric (e.g., artificial ant problem), this approach can also bring some benefits by improving locality.

Of course, measuring semantic similarity can be a challenge in itself. Because the set of 20 fitness cases we used here cannot in general reflect the semantic richness of continuous functions, the semantic distance we used here is essentially only an estimate. However, it seems that this estimate was good enough, as the method works well despite the fact different sets of fitness cases have been used here for optimization of embedding and for evolutionary search.

We demonstrated that semantic embedding can bring substantial benefits in at least two scenarios: when there is a need of solving multiple GP problems using the same instruction set and program length limit, and for problem decomposition. For now, the most promising direction of future research seems to be better exploitation of embedding in the latter, compositional scenario. We considered here a simple scheme of composition, but we expect that more can be gained by bringing in more sophistication, in particular by explicitly taking into account the smoothness with

respect to program input v, or simultaneous use of multiple embeddings, e.g., optimized separately for p_1 and p_2.

In a broader perspective, the whole approach can be considered as an automatic induction of a new programming language. An embedding can be seen as single-input (v), single-output meta-instruction parameterized by a vector of coordinates $\mathbf{x}$ that determines its semantics. Such meta-instruction is functionally complete (i.e., it encapsulates all possible input-output transformations that can be expressed under the assumed program length limit) and can be optimized in a reasonable time, avoiding combinatorial explosion. Most importantly, the second experiment suggests that evolution of programs composed of such meta-instructions can be more effective than the search in the original program space. Alternatively, this can be viewed as a form of problem decomposition, with the goal of finding such decomposition of the original program space into subspaces, that the subspaces can be embedded at high locality.

7. ACKNOWLEDGMENTS

Work supported by grants no. N N519 441939 and 91507.

8. REFERENCES

[1] E. Galvan-Lopez, M. O'Neill, and A. Brabazon. Towards understanding the effects of locality in GP. In *Eighth Mexican International Conference on Artificial Intelligence, MICAI 2009*, pages 9–14, Nov. 2009.

[2] C. Johnson. Genetic programming crossover: Does it cross over? In L. Vanneschi, S. Gustafson, A. Moraglio, I. De Falco, and M. Ebner, editors, *Proceedings of the 12th European Conference on Genetic Programming, EuroGP 2009*, volume 5481 of *LNCS*, pages 97–108, Tuebingen, Apr. 15-17 2009. Springer.

[3] K. Krawiec. Learnable embeddings of program spaces. In S. Silva, J. A. Foster, M. Nicolau, M. Giacobini, and P. Machado, editors, *Proceedings of the 14th European Conference on Genetic Programming, EuroGP 2011*, volume 6621 of *LNCS*, pages 167–178, Turin, Italy, 27-29 Apr. 2011. Springer Verlag.

[4] K. Krawiec and P. Lichocki. Approximating geometric crossover in semantic space. In G. Raidl, F. Rothlauf, G. Squillero, R. Drechsler, T. Stuetzle, M. Birattari, C. B. Congdon, M. Middendorf, C. Blum, C. Cotta, P. Bosman, J. Grahl, J. Knowles, D. Corne, H.-G. Beyer, K. Stanley, J. F. Miller, J. van Hemert, T. Lenaerts, M. Ebner, J. Bacardit, M. O'Neill, M. Di Penta, B. Doerr, T. Jansen, R. Poli, and E. Alba, editors, *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 987–994, Montreal, 8-12 July 2009. ACM.

[5] K. Krawiec and B. Wieloch. Functional modularity for genetic programming. In G. Raidl, F. Rothlauf, G. Squillero, R. Drechsler, T. Stuetzle, M. Birattari, C. B. Congdon, M. Middendorf, C. Blum, C. Cotta, P. Bosman, J. Grahl, J. Knowles, D. Corne, H.-G. Beyer, K. Stanley, J. F. Miller, J. van Hemert, T. Lenaerts, M. Ebner, J. Bacardit, M. O'Neill, M. Di Penta, B. Doerr, T. Jansen, R. Poli, and E. Alba, editors, *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 995–1002, Montreal, 8-12 July 2009. ACM.

[6] W. B. Langdon. Convergence rates for the distribution of program outputs. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 812–819, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[7] B. McKay, M. J. Willis, and G. W. Barton. Using a tree structured genetic algorithm to perform symbolic regression. In A. M. S. Zalzala, editor, *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, volume 414, pages 487–492, Sheffield, UK, 12-14 Sept. 1995. IEE.

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

[9] A. Moraglio and R. Poli. Topological interpretation of crossover. In K. Deb, R. Poli, W. Banzhaf, H.-G. Beyer, E. Burke, P. Darwen, D. Dasgupta, D. Floreano, J. Foster, M. Harman, O. Holland, P. L. Lanzi, L. Spector, A. Tettamanzi, D. Thierens, and A. Tyrrell, editors, *Genetic and Evolutionary Computation – GECCO-2004, Part I*, volume 3102 of *Lecture Notes in Computer Science*, pages 1377–1388, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[10] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[11] U. N. Quang, X. H. Nguyen, and M. O'Neill. Semantic aware crossover for genetic programming: the case for real-valued function regression. In L. Vanneschi, S. Gustafson, A. Moraglio, I. De Falco, and M. Ebner, editors, *Proceedings of the 12th European Conference on Genetic Programming, EuroGP 2009*, volume 5481 of *LNCS*, pages 292–302, Tuebingen, Apr. 15-17 2009. Springer.

[12] F. Rothlauf. *Representations for Genetic and Evolutionary Algorithms*. Physica-Verlag, Heidelberg New York, 2002.

[13] M. Schoenauer, M. Sebag, F. Jouve, B. Lamy, and H. Maitournam. Evolutionary identification of macro-mechanical models. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 23, pages 467–488. MIT Press, Cambridge, MA, USA, 1996.

[14] L. Vanneschi and M. Tomassini. Pros and cons of fitness distance correlation in genetic programming. In A. M. Barry, editor, *GECCO 2003: Proceedings of the Bird of a Feather Workshops, Genetic and Evolutionary Computation Conference*, pages 284–287, Chigaco, 11 July 2003. AAAI.