Learnable Embeddings of Program Spaces

Krzysztof Krawiec

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

Abstract. We consider a class of adaptive, globally-operating, semantic-based embeddings of programs into discrete multidimensional spaces termed prespaces. In the proposed formulation, the original space of programs and its prespace are bound with a learnable mapping, where the process of learning is aimed at improving the overall locality of the new representation with respect to program semantics. To learn the mapping, which is formally a permutation of program locations in the prespace, we propose two algorithms: simple greedy heuristics and an evolutionary algorithm. To guide the learning process, we use a new definition of semantic locality. In an experimental illustration concerning four symbolic regression domains, we demonstrate that an evolutionary algorithm is able to improve the embedding designed by means of greedy search, and that the learned prespaces usually offer better search performance than the original program space.

1 Introduction

Genetic Programming (GP) is an evolution-inspired search in space of programs guided by their performance in some environment. Its inherent feature is that a program (code, syntax) and its effect (semantics, behavior) are usually separated by multiple levels of abstraction, as opposed to conventional optimization tasks where the relationship between genotype and phenotype is typically much more direct. As a result, even minute changes of code can radically alter program’s semantics and fitness, which makes GP fitness landscapes quite chaotic, often hampers scalability of search algorithms, and limits applicability of this methodology to difficult real-world problems.

This problem has been noticed in different contexts and its aftermaths identified and labeled by various terms. In the global perspective, it has been observed that fitness landscapes in GP typically have low fitness-distance correlation [13]. As a result, most crossovers are destructive and only a small fraction of them lead to increase in fitness [4]. In the local perspective, GP’s genotype-phenotype mapping has low locality [6], which makes particularly hard designing mutation operators that behave in a way they should, i.e., have only minor effect in fitness.

In the eyes of many, this is an inherent property of programming and programming languages, and the attempts aimed at circumventing it are futile. This stance is particularly common among such adversaries of operational semantics like E. Dijkstra, who famously stated that “In the discrete world of computing, there is no meaningful metric in which ‘small’ changes and ‘small’ effects go hand in hand, and there never will be” [3]. Nevertheless, a growing body of research evidence in GP, mostly concerning design of
search operators, seems to suggest otherwise. There is a repertoire of search operators
designed to be more aware of the abovementioned problems, like context-preserving
crossover [1], semantically-aware crossover [9], or geometrical crossover [5], to name
only a few. On the other end, there is a long list of alternative program representations
for GP, like Linear Genetic Programming [2] and Cartesian Genetic Programming [7],
some of which have been designed to, among others, circumvent the above problem.

The above approaches are symbolic in the sense that they manipulate symbols given
a priori with problem specification (typically instructions), and usually they do that in
abstraction from the actual meaning (semantics) of those symbols. 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, which itself does not have to be problematic, but their
redundancy is difficult to capture – for instance, reversals of instruction order can be
neutral for program outcome. Finally, they are highly epistatic – distant code fragments
often strongly interact with each other, with complex effects on program semantics.

The primary objective of this study is to investigate alternative program representa-
tion spaces, in which programs are aligned in a way that is less related to their syntax
and more to their semantics. Our main contribution is a concept of program represen-
tation space (prespace) that to some extent abstracts from the symbolic program code
and delegates the process of automated programming to another domain. In short, rather
than designing new search operators or devising new structures for program represen-
tations, we switch to prespace, embed the programs in it, and learn a mapping from that
space to the original space, a mapping with some desirable properties.

2 Programs, Semantics, and Locality of Semantic Mapping

In abstraction from any specific search algorithm, we formalize in this section the no-
tion of search space and its properties. Let \( P \) be a program space, and let \( s(p) \) de-
define the semantics of program \( p \in P \), where \( s : P \rightarrow S \) 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., \( S \equiv \text{image}(P) \)). Typically, \( s \) is also not invert-
ible and \( |P| \gg |S| \). By assuming that \( s \) is endowed with the entire knowledge required
for program execution, we are not interested here in program syntax and the process of
program interpretation.

We define problem instance as a program space augmented by a fitness function
\( f : P \rightarrow \mathbb{R} \). Multiple problem instances can be defined for a given program space \( P \). In
GP, the fitness \( f \) of a program \( p \) is often directly based on a distance between \( s(p) \) and
a predefined point in \( S \).

Let \( N(p) \) be the neighbourhood of \( p \) \( (N : P \rightarrow 2^P \setminus \{\emptyset\}, p \notin N(p)) \). Typically, \( N(p) \)
is the set of all programs that can be built form \( p \) by introducing small changes in it
(like substituting a single instruction with another). From EC perspective, it is common
to identify \( N(p) \) with the set of all mutants of \( p \). The pair \( (P, N) \) is also referred to as
configuration space, and the triple \((P, N, f)\) as fitness landscape (see, e.g., [8]).