Medial Crossovers for Genetic Programming

Krzysztof Krawiec

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

Abstract. We propose a class of crossover operators for genetic programming that aim at making offspring programs semantically intermediate (medial) with respect to parent programs by modifying short fragments of code (subprograms). The approach is applicable to problems that define fitness as a distance between program output and the desired output. Based on that metric, we define two measures of semantic 'mediality', which we employ to design two crossover operators: one aimed at making the semantic of offsprings geometric with respect to the semantic of parents, and the other aimed at making them equidistant to parents' semantics. The operators act only on randomly selected fragments of parents' code, which makes them computationally efficient. When compared experimentally with four other crossover operators, both operators lead to success ratio at least as good as for the non-semantic crossovers, and the operator based on equidistance proves superior to all others.

Keywords: Genetic programming, Program semantic, Semantic crossover.

1 Introduction

The function of crossover in evolutionary computation is to produce new candidate solutions (offspring) that fuse some of the features of existing candidate solutions (parents). This principle is typically implemented by following the biological paradigm, i.e., by recombining the genotypes of parents. An implicit assumption hidden in this line of reasoning is that recombination effects propagate to the phenotypes in an analogous way, and the offspring is expected to *behave* in a way that mixes, to some extent, the behaviors of its parents.

This renders invalid if the elements of the genotype do not map one-to-one to the elements of phenotype, which is unfortunately always the case for nontrivial problems. Most problems considered in genetic programming (GP, [1]) belong here too, as the interactions between the elements of genotype (instructions) are usually strong, and influence the output of program (and subsequently its fitness) in a complex way that is hard to predict and model. As a result, an operator that recombines parents' programs cannot be in general expected to always recombine their behaviors.

Trying to circumvent this difficulty, we propose a family of operators, *medial crossovers*, designed to produce programs that inherit not only parts of parents' code, but also some elements of parents' semantics. To this aim, we exploit the

A. Moraglio et al. (Eds.): EuroGP 2012, LNCS 7244, pp. 61–72, 2012.

fact that many genetic programming tasks define fitness using a metric that compares the actual program behavior with the desired one. In the proposed operators, that very metric is employed to alter subprograms within parents' code so that their outcome at certain execution stage become more similar to each other, which indirectly affects the semantic of entire offspring programs in a way that gives chance to exploit certain properties of the fitness landscape.

Although research on semantic in GP intensified only recently, there are a few related contributions. McPhee *et al.* were probably one of the first to study the impact of crossover on program semantics and so-called semantic building blocks [2]. In [3], Moraglio *et al.* considered properties of semantic spaces for different metrics and provided guidelines for designing semantically geometric crossovers. The semantically-aware crossover by Quang *et al.* [4] swaps a pair of subtrees in parent solutions that have similar, yet not too similar, semantics. To author's knowledge however, this is the first study on semantic crossovers that act in a semantically medial way on the level of subprograms.

2 Metric-Based Crossover Operators

We consider here the class of problems for which the objective function captures the divergence between the program output and the desired output, given as a part of problem formulation. This class embraces the prevailing part of GP applications, where individuals are typically tested on a set of fitness cases, and fitness is some form of error built upon the outcome of these tests (cf. symbolic regression and boolean function synthesis).

Formally, we assume that the (minimized) fitness of a candidate solution p is defined as $f(p) = ||p, t||$, where t is the (known) desired output (target) determined by *problem instance*, and $||\,||$ is a metric measures the distance between t and the output produced by p. The metric imposes a structure on the set of programs (potential solutions), turning it into a *space*. It is essential to emphasize that, throughout this paper, the metric operates on the phenotypic level, ignoring program code (syntax) and taking into account only its *output*, which within this paper we identify with *program semantics* [2,5,6]. We call a fitness function defined in this way *metric-based fitness function*.

Crossover is a search operator that produces a new search point (offspring) o (or a pair of offspring) based on a pair of search points p_1, p_2 (parents). In the following, we consider only *nontrivial* offspring $o \neq p_1, p_2$.

The metric allows us to express some properties of the offspring in the context of its parents. An offspring o that fulfills

$$||o, p_1|| + ||o, p_2|| = ||p_1, p_2||, \tag{1}$$

will be referred to as *geometric offspring,* and a crossover operator that produces such offspring *geometric crossover* (a.k.a. topological crossover [7]).

To demonstrate the potential that dwells in geometric operators, let us consider a special case, a geometric crossover for which the distributions of $||o, p_1||$ and $||o, p_2||$ are uniform and the city-block distance as the metric. In such

a case, the expected fitness of offspring is $(f(p_1) + f(p_2))/2$. The sketch of proof is as follows. The metric is a norm here and we switch to vector spaces, so let x^i denote the ith coordinate of point in the space. A geometric offspring o has to fulfill $o^i = \alpha^i p_1^i + (1 - \alpha^i p_2^i)$, where $\alpha^i \in (0, 1)$. Let $d^i(x)$ denote the difference between the location of x and t on coordinate i. Then, $d^i(o) = t^i - o^i = t^i - [\alpha^i p_1^i + (1 - \alpha^i p_2^i)]$. It is easy to show that, because for uniform distribution $\mathbb{E}(\alpha^i) = \mathbb{E}(1 - \alpha^i) = \frac{1}{2}$, it must hold:

$$\mathbb{E}(d^i(o)) = t^i - \mathbb{E}(\alpha^i)p_1^i - \mathbb{E}(1 - \alpha^i)p_2^i = t^i - \frac{p_1^i}{2} - \frac{p_2^i}{2} = \frac{d(p_1) + d(p_2)}{2}$$

Without loss of generality, we can ignore the sign of $\mathbb{E}(d^i(o))$, in which case it becomes the expected contribution to o's distance from t (fitness) on the ith dimension. The expected fitness of o for city-block fitness is then

$$\mathbb{E}(f(o)) = \mathbb{E}(\sum_i d^i(o)) = \sum_i \mathbb{E}(d^i(o)) = \frac{\sum_i d^i(p_1) + \sum_i d^i(p_1)}{2} = \frac{f(p_1) + f(p_2)}{2}$$

Verification of this property for other norms is beyond the scope of this paper. Nevertheless, under all metrics, the offspring cannot be worse than the worse of the parents.

As another important property, let us notice that, under all metrics, the expected fitness of a geometric offspring is minimized when it fulfills also the condition of *equidistance*:

$$||o, p_1|| = ||o, p_2|| \tag{2}$$

Without providing formal proof, let us notice that, under any finite-support distribution of targets t, given random locations p_1 and p_2, a point that is equidistant from them is expected to be the closest to t.

Producing a geometric offspring that is simultaneously equidistant can be difficult (not mentioning the challenge of designing a geometric crossover alone, which we discuss later). The reason for this is twofold: (i) for discrete spaces such a point may not exist, and (ii) a program o with the output that fulfills (2) may not exist (an equidistant semantics cannot be expressed within the assumed programming language). A possible solution to this problem is to relax condition (2) and produce a geometric offspring that is as equidistant as possible. However, a question arises: how much a program should be allowed to diverge from equidistance to be still considered a useful offspring for a particular pair of parents? As $||o, p_1||$ and $||o, p_2||$ diverge from each other, the offspring becomes more and more similar to one of the parents, which makes the search less effective. Note also that, in general, there is no guarantee of existence of even a single nontrivial geometric offspring for a pair of programs.

An analogous problem arises when one takes an alternative path, i.e., relaxing geometricity condition (1) while requiring perfect equidistance (2): again, a perfectly equidistant offspring may simply not exist at all (i.e., even if geometricity is completely ignored), and it is hard to tell what is the acceptable divergence from geometricity.

An offspring that meets even one of the above conditions may therefore not exist. This obliges us to simultaneously relax both of them to design an operator that works in practice. This can be formalized by defining analogous criteria: *divergence from geometricity* d_G and *divergence from equidistance* d_E:

$$\begin{aligned} d_G(o, p_1, p_2) &= ||o, p_1|| + ||o, p_2|| - ||p_1, p_2|| \\ d_E(o, p_1, p_2) &= \big|\, ||o, p_1|| - ||o, p_2|| \,\big| \end{aligned} \tag{3}$$

In the following, we consider operators that attempt to minimize d_G or d_E, which we refer to as (semantically) *medial* crossovers.

An exact implementation of medial crossover is in most cases technically infeasible. The primary cause is the complexity (and, typically, irreversibility) of genotype-phenotype mapping, which makes direct synthesis of an offspring that minimizes one or both of the above criteria impossible or at least computationally intractable. Apart from certain special cases [3], there is no direct way of constructing an offspring program that exhibits intermediate behavior with respect to the behaviors of parents programs (intermediate in the sense of the assumed metric $||\cdot||$). Standard crossover operators typically ignore that fact and produce offspring that are intermediate in purely syntactical terms, but this does not translate into analogous intermediacy in the space of program behaviors. For instance, tree-swapping crossover tends to produce offspring that are semantically very different from the parents [8].

In theory, one could consider *all* potential offspring (i.e., all possible programs), and pick the one that minimizes the criterion (criteria). But such procedure is computationally impractical (exponential time complexity w.r.t. program length), and can be only approximated via sampling, which we studied in past [5]. Moreover, if all programs were to be generated *and* run (the latter required to know the program output), then also the optimal solution ($o = t$) would be among them. A search algorithm that has to consider *all* solutions to proceed with a single iteration has limited usefulness, to say the least.

In general then, it is impossible to generate 'mixtures' of parents that are optimal in the sense of d_G or d_E. Can we at least approximate such behavior? We investigate this possibility in the subsequent section.

3 Partially Medial Crossover

In this section, we come up with a family of crossover operators that are based on the introduced criteria and are technically realizable. The key idea is to port the concepts from Section 2, where they applied to entire programs, to *subprograms*. Our hypothesis is that by making programs semantically more similar to each other (*medial*) at intermediate execution steps (*partially*), we have a chance of making them overall more similar.

The definition of subprogram depends on the assumed program representation. For simplicity, we represent here programs as linear sequences of *instructions*, in which case a subprogram is a continuous subsequence of instructions.

Let $p[i..j]$, $i \leq j$ denote the subprogram of program p composed of instructions from i^{th} to j^{th} inclusive. We also assume that concatenation of any programs p_1 and p_2 is a valid program and denote it as p_1p_2. Finally, we limit our considerations to domains with Markov property: the result of an instruction (the memory state it produces) depends only on the current memory state. $|p|$ is the length of program p (number of instructions).

The standard two-point crossover for this program representation has the natural interpretation of swapping parents' subprograms located between two randomly drawn loci i and j. The *partially medial crossover* (PMX) we propose is also homologous, and also affects subprograms in the parent solutions, however, the pieces of code pasted into the offspring result from analysis of semantic properties of the corresponding parents' subprograms. The choice of the code to be pasted between loci i and j can follow a simple principle: paste the subprogram that makes the resulting offspring possibly medial with respect to the parents *at locus* j.

More formally, the subprogram p that replaces the instructions from i to j ($|p| = j - i + 1$) in the k^{th} offspring ($k = 1, 2$) to be created from parents p_1 and p_2 is the one that minimizes:

$$\arg\min_{p} d(p_k[1..i-1]p,\, p_1[1..j],\, p_2[1..j]) \tag{4}$$

where d is one of the divergence criteria (Eq. 3). Thus, when minimizing the divergence, PMX takes into account only the semantic effects of the first j instructions. The k^{th} offspring is a program of the form $p_k[1..i-1]pp_k[j+1..|p_k|]$, where the 'head' $p_k[1..i-1]$ and the 'tail' $p_k[j+1..|p_k|]$ are copied from the k^{th} parent.

PMX considers *all* potential subprograms p that can be pasted between loci i and j. As the number of such instruction sequences is exponential in function of $|p|$, we limit the span of loci i and j: only i is drawn at random, and j is set to $i + l - 1$, where l is a parameter that limits the length of considered subprograms. The number of instructions we need to execute in order to calculate (4) is still exponential in function of l, but can be significantly reduced, so that the computational overhead for small l is reasonable (see analysis of computational complexity in Section 5).

The ties that may occur when minimizing (4) are resolved at random. In this way, we avoid unnecessary bias, and the outcome of crossover becomes partially indeterministic.

4 The Experiment

The objective of the experiment is to compare the partially medial crossover in its two variants, minimizing the divergence from geometricity d_G (PMX_G) and minimizing the divergence from equidistance d_E (PMX_E). As control approach we use macromutation (MM), which overwrites the affected instructions with randomly generated instructions, and two-point crossover (2PX), that swaps

the affected subprograms between parents. To make comparison fair, they both affect a randomly selected continuous subsequence of instructions of length l, so the fraction of code they are allowed to modify is the same as for PMX.

We employ also one-point crossover (1PX) that draws a locus at random, splits each parent into head and tail, and swaps the tails, and a 'reset' operator RND that produces a random offspring (the *entire* offspring's code is randomized). All operators produce two offspring when applied to a pair of parents.

We conduct two experiments, one to quantitatively characterize the semantic impact of considered operators (Section 4.2), and one to assess the performance of evolutionary search (Section 4.3).

4.1 The Puzzle World

We adopt the task of solving the sliding puzzle as an experimental framework. Consider the 3×3 sliding puzzle with 8 movable pieces. The puzzle can be in one of $9! = 362,880$ states, which can change as a result of four possible moves L, R, U, D, where we assume the moves to shift the empty space (and thus also a piece). Any finite sequence composed of moves can be considered as a program, with moves playing roles of instructions, and the state of the puzzle corresponding to a memory state of a virtual machine that executes the program.

We define a *puzzle task* as follows: given a starting state s_0 and a target state t, find a program of length m that transforms the former one into the latter. Formally, let $s(p)$ denote the final memory state produced by program p that started execution from memory state s_0. The puzzle task is then to find p, $|p| = m$ such that $s(p) = t$. Although insisting on finding a program of length *exactly* m may sound too specific, let us note that almost every shorter program that solves a task can be extended to length m by inserting ineffective instructions that shift the space back and forth. Thus, a task can be solved by a program which effective length is less than m.

To solve the sliding puzzle task with an evolutionary approach, we evolve individuals that encode programs as fixed-length sequences (vectors) of instructions. Evolution will be driven by a (minimized) fitness function f defined as the total city-block distance between the locations of the 8 pieces and the empty space (_) in $s(p)$ and locations of corresponding elements in t, which is always an even number. Such definition of f is not accidental, it is used as a heuristic path cost estimate to solve this type of problems with exact algorithms like A*.

Formally, $f(p) = ||s(p), t||$, and f is thus a metric-based fitness function in the sense introduced in Section 2. For instance, if t = (1,2,3,4,5,6,7,8,_), then

$$f\left(\begin{array}{|c|c|c|}\hline 1&2&3\\\hline 4&6&\\\hline 7&8&5\\\hline\end{array}\right) = \left\|\begin{array}{|c|c|c|}\hline 1&2&3\\\hline 4&6&\\\hline 7&8&5\\\hline\end{array}\,,\begin{array}{|c|c|c|}\hline 1&2&3\\\hline 4&5&6\\\hline 7&8&\\\hline\end{array}\right\| = 0+0+0+0+1+1+0+0+2 = 4$$

Despite apparent simplicity, sliding puzzle captures the important features of programming task. It is *contextual*, i.e., the effect of an instruction depends on the current memory state. In particular, an instruction or instruction sequence can be ineffective when applied to a specific state (like the move R applied to

Table 1. The statistics describing average spatial relationships between the semantics of parents and offspring for different types of crossover operators

	Statistics	RND	MM	2PX	1PX	PMX_E	PMX_G
1	$\overline{\lvert\lvert p_1, p_2\rvert\rvert - \lvert\lvert o_1, o_2\rvert\rvert}$	0.00	0.00	0.00	0.00	0.83	0.78
2	$\overline{d_G(o, p_1, p_2)}$	11.53	3.83	3.76	4.61	4.91	2.83
3	$\overline{d_E(o, p_1, p_2)}$	3.54	7.94	7.88	5.07	5.87	7.98
4	$\overline{\lvert\lvert p, o\rvert\rvert}$	11.52	3.83	3.83	8.71	5.51	3.31
5	$\Pr(d_G(o, p_1, p_2) = 0 \lvert o \neq p_1, p_2)$	0.02	0.05	0.06	0.11	0.08	0.09

the state evaluated above). It is *compositional*: new programs can be created by composing (concatenating) other programs. Finally, the memory is composed of *multiple* elements and a single instruction changes only some of them.

Apart from these features, sliding puzzle exhibits also some features characteristic for *genetic* programming tasks. Firstly, programs are evaluated by *running* them (testing) on input data. Secondly, the performance of a program is a function of a *distance* between its output and the desired output.

4.2 Experiment 1: Properties of Search Operators

In this experiment, we analyze properties of the considered crossover operators by applying them to random programs. For each operator, we repeat $1,000,000$ times the following steps: (i) generate a random starting state s_0, (ii) generate two random parents p_1, p_2, (iii) apply the operator to p_1 and p_2, producing offspring o_1 and o_2, (iv) run the parent and offspring programs, starting with memory state s_0, and (v) measure the spatial relationships between the outputs (semantics) of parent and offspring programs. Note that this analysis abstracts from the target t, and that all these measurements concern *entire* programs (the final program outcomes).

For brevity we report the results only for program length $m = 40$ and $l = 3$, (the number of instructions affected by the PMX, MM, and 2PX), but other lengths led to consistent conclusions. Table 1 shows the averaged statistics obtained in step (v) of the above procedure, including 1) the reduction of distance between offspring compared to the distance between parents $\overline{||p_1, p_2|| - ||o_1, o_2||}$, 2) mean divergence from geometricity d_G, 3) mean divergence from equidistance d_E, 4) mean distance between the parent and the offspring $\overline{||p, o||}$, and 5) probability of generating a perfectly geometric offspring, excluding the trivial cases, i.e., such that the offspring is a copy of one of the parents.

We start with noting that the non-semantic operators (RND, MM, 2PX, 1PX) are incapable to produce offspring that is more semantically similar than the parents (row 1). The offspring of PMX operators, on the other hand, is typically substantially more similar to each other.

Some non-semantic operators, despite their simplicity, turn out to be quite good at producing medial solutions (rows 2 and 3). MM, 1PX, and 2PX diverge

from geometricity (d_G) more than PMX_G, which is the best in that respect, but less than PMX_E. However, considering the rate of non-trivial offspring that are perfectly geometric (row 5), only 1PX turns out to be better than PMX_G.

The offspring of PMX_E is not the most equidistant from parents, yielding to 1PX and RND. The fact that RND attains the lowest d_E may be surprising at first, but can be explained by the structure of the space. For the considered sample of programs, the maximum distance between any pair of parents is 24, and the median of distance is 12. As the metric assumes only even values, a completely random offspring produced by RND is then quite likely to be equidistant. However, such offspring is typically very different from the parents, having mean parent-offspring distance almost twice as high as for PMX_E (row 4), so it is unlikely to inherit much of their behavior.

The main conclusion we draw from this experiment is that, despite the fact that our medial crossover operators are partial, i.e., affect only short subprograms of parents' code, their effects propagate to the end of program and affect their semantic in expected way (i.e., consistent with the used criterion). Because they also clearly produce offspring that is more similar than the parents (row 1) and semantically not too distant from them (row 4), they are quite likely to prove useful in evolutionary search, which we verify in subsequent section.

One might argue that, from time to time, PMX happens to affect the very last 3 of 40 instructions, and the observed values of indicators are mainly due to such cases. Such events are however rare: there are 37 possible crossover points, so less than 3% of cases fall into this category. Also, we conducted an analogous analysis, not reported here for brevity, where the first crossover point was constrained to the first half of the genotype. The resulting values of indicators, though less extreme, confirmed the above conclusions.

4.3 Experiment 2: Performance in Evolutionary Search

To evaluate usefulness of PMX operators as search tools, we carried out a evolutionary experiment for different program lengths ($m = 20, 40, 60, 80$) and various length of the affected code fragment ($l = 3, 4, 5$).

Each evolutionary run is to find a solution to a specific, randomly generated puzzle task (s_0,t), where s_0 and t are random permutations of the canonical state (1,2,3,4,5,6,7,8,_). For each combination of m and l, 50 independent runs were carried out, each solving a different puzzle task. Note that, as only half of the 9! permutations of pieces are reachable from any given puzzle state [9], not all such tasks are solvable, i.e., there does not exist a sequence of moves of any length that leads from s_0 to t. Moreover, as the worst configuration of 8-puzzle requires 31 moves to solve (see the OEIS integer sequence A087725, [10]), some of the tasks are not solvable for the smallest considered program length $l = 20$. These difficulties however affect equally all the considered methods, so the comparison remains fair.

We use generational evolution driven by the fitness function defined in Section 4.1, i.e., as overall city-block distance between the locations of pieces in the state reached by the program from the locations of pieces in the goal state t. Each run

Table 2. Success ratio [%] for the 3×3 puzzle, for various length of code fragment affected by crossover (l) and total program length (m)

m	20			40			60			80		
l	3	4	5	3	4	5	3	4	5	3	4	5
RND	0	0	0	2	2	2	10	10	10	10	10	10
MM	2	0	0	22	30	28	40	42	48	46	50	50
2PX	0	0	0	2	4	2	6	6	6	22	16	20
1PX	0	0	0	2	8	6	0	10	10	18	18	16
PMX_E	4	2	6	30	48	46	46	62	60	58	60	62
PMX_G	0	0	0	6	6	6	18	10	20	40	28	34

evolves a population of $1,000$ individuals for $1,000$ generations, unless a solution is found earlier. The solutions are selected using tournament of size 7, after which they either undergo crossover using one of the aforementioned operators (with probability 0.9), or one-point mutation (with probability 0.1, and probability of affecting a single instruction 0.03).

Note that for RND evolution is effectively a random memoryless search.

Table 2 presents the success rate for different settings. Before comparing the operators, we should notice that all of them perform better when operating on longer programs. As the average task difficulty remains the same, this suggests that finding a program that reaches the target state from the starting state in, say, 40 instructions can be more difficult than finding a program that does the same using 80 instructions.

MM performs remarkably well, especially when confronted with other non-semantic operators. Apparently, introducing completely random modifications in the code is on average more profitable than purely syntactic swapping of code fragments implemented by 1PX and 2PX. However, the semantic-aware manipulation provided by PMX clearly pays off. In particular, for all considered parameter settings, PMX_E finds the optimum more frequently than any other operator. The efficiency of PMX_G as a search tool is much worse, though almost always not worse than 2PX and 1PX. This suggests that, at least for the considered domain of sliding puzzle, it is more important to generate solutions that inherit roughly the same 'fraction' of behavior from both parents, even if that share is low. Minimizing the divergence from geometricity is less effective, which may be due to the slower pace at which the PMX_G traverses the search space (see row 4 of Table 1).

The high performance of MM suggests that the task we consider here is relatively easy. This observation inclined us to consider the harder 4×4, 15-piece puzzle. As the 4×4 puzzle has 20,922,789,888,000 possible states, this time we consider longer programs of length 100, 200, and 300. All other settings, including the method of task generation, remain the same.

The results, presented in Table 3, support earlier conclusions. PMX_E is again superior, and its relative outperformance over MM is even larger than for the 3×3 puzzle: probability of finding the optimum is now often several times greater than for MM. PMX_G is much worse again, but still comparable to MM. Other operators fail completely.

Table 3. Success ratio [%] for the 4×4 puzzle, for various length of code fragment affected by crossover (l) and total program length (m)

m	100			200			300		
l	3	4	5	3	4	5	3	4	5
RND	0	0	0	0	0	0	0	0	0
MM	0	0	0	4	2	4	10	8	4
2PX	0	0	0	0	0	0	0	0	0
1PX	0	0	0	0	0	0	0	0	0
PMX_E	0	2	2	4	12	20	26	18	22
PMX_G	0	0	2	0	2	2	4	4	6

5 Discussion

Good performance of PMX suggests that generating programs that share some elements of behavior (semantic) with the earlier visited solutions is an important and desired feature of search operator. In particular, this turns to be more important here than inheriting the genetic material.

However, PMX is incapable to explicitly modify the semantic of programs and operates on short subprograms only. Although, as we have shown in Table 1, such modifications tend to affect the output of entire programs in the expected way, it is the affected loci (instructions from i to j) where the semantic effect of PMX is the strongest. Why then generating programs that exhibit medial semantics at *intermediate* stages of execution should be profitable?

Our working explanation is the modularity of the puzzle task. To solve a task, i.e., to reach the memory state t, one has first to solve a certain subtask, thus reach a certain intermediate memory state s' (subgoal), which is typically unknown prior to solving the task. The partially medial crossovers, by making more similar the memory states visited by the parent programs, may promote convergence to such subgoals. However, how effective this process is can depend on many factors, including the number of solutions that exist for a given puzzle task, the number of such subgoals for a given task, and the structure of the fitness landscape (e.g., how does attaining such a subgoal pay off in terms of fitness). Thus, this hypothesis remains to be verified, possibly with help of the concept of interdependency (see, e.g., [11,12]).

An important difference between PMX and the non-semantic operators is that the former involve partial execution of the considered program fragments. This involves an extra computational overhead, which is not reflected in Tables 2 and 3. PMX needs to run (i) the heads of the parent programs and then (ii) the subprograms to be pasted (Formula 4). The former part requires execution of $i - 1$ instructions per parent, but this has to be done only once: we store the result of head execution (the final memory state), and use it then as the starting memory state for considered subprograms. For a programming language comprising n instructions, PMX considers in the latter part n^l subprograms of length l, which apparently requires executing lm^l instructions. Note however that these subprograms partially overlap and can be represented by a n-ary tree

of depth l, which comprises only $n^l - 1$ nodes (instructions). For instance, for $l = 3$ and $n = 4$ this means reduction from 64 to 15 instructions.

The overall number of instructions executed when crossing over at locus i is then $2(i+n^l-2)$, where factor 2 is due to producing two offspring. The worst-case cost, for crossing over at the very end of parents, amounts to $2(m - l + n^l - 1)$. For short subprograms and small instructions sets this is a moderate computational overhead. Should this overhead become unacceptable, some form of sampling from the space of subprograms may be considered. Given that evolutionary search is stochastic anyway, exact minimization of d_G or d_E in (4) may be unnecessary, and some performance improvements should be possible to attain with approximate approach at lower computational cost.

6 Conclusion

The overall conclusion of this study is that partially medial crossovers, search operators that make the *subprograms* of parent programs *semantically intermediate* with respect to parents, can positively contribute to effectiveness of genetic programming algorithms. The extent of this contribution is undoubtedly problem-dependent, and will vary with instruction set and metric definition. Nevertheless, we hyptothesize that a significant fraction of real-world genetic programming problems can potentially benefit from this approach. Our rationale for this claim is that real-world programming tasks tend to be modular.

The puzzle world seems to constitute a convenient demonstrator for the concept of PMX for several reasons. Firstly, because the programs are intended to solve a specific instance of puzzle task, for a *single* input (starting memory state s_0), the semantic of a (sub)program can be identified with the memory state it produces. For problems that require testing a (sub)program on a set of fitness cases (e.g., symbolic regression), the notion of semantic and the associated metric would have to be adapted (cf. [2,4,5,13]). Secondly, puzzle manipulating has the natural interpretation of *sequential* programs, which implies convenient correspondence of loci in the parent programs. For less regular program representations like trees, PMX would need some extra means to first decide which subprograms (subtrees) to operate on. Such extensions and the above hypothesis concerning modular problems remain to be verified in future research.

Acknowledgment. This work has been supported by NSC grant no. DEC-2011/01/B/ST6/07318.

References

1. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
2. McPhee, N.F., Ohs, B., Hutchison, T.: Semantic Building Blocks in Genetic Programming. In: O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcázar, A.I., De Falco, I., Della Cioppa, A., Tarantino, E. (eds.) EuroGP 2008. LNCS, vol. 4971, pp. 134–145. Springer, Heidelberg (2008)

3. Moraglio, A., Krawiec, K., Johnson, C.: Geometric semantic genetic programming. In: Igel, C., Lehre, P.K., Witt, C. (eds.) The 5th Workshop on Theory of Randomized Search Heuristics, ThRaSH 2011, Copenhagen, Denmark (2011)
4. Nguyen, Q.U., Nguyen, X.H., O'Neill, M.: Semantic Aware Crossover for Genetic Programming: The Case for Real-Valued Function Regression. In: Vanneschi, L., Gustafson, S., Moraglio, A., De Falco, I., Ebner, M. (eds.) EuroGP 2009. LNCS, vol. 5481, pp. 292–302. Springer, Heidelberg (2009)
5. Krawiec, K., Lichocki, P.: Approximating geometric crossover in semantic space. In: Raidl, G., Rothlauf, F., Squillero, G., Drechsler, R., Stuetzle, T., Birattari, M., Congdon, C.B., Middendorf, M., Blum, C., Cotta, C., Bosman, P., Grahl, J., Knowles, J., Corne, D., Beyer, H.G., Stanley, K., Miller, J.F., van Hemert, J., Lenaerts, T., Ebner, M., Bacardit, J., O'Neill, M., Di Penta, M., Doerr, B., Jansen, T., Poli, R., Alba, E. (eds.) GECCO 2009: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, Montreal, pp. 987–994. ACM (2009)
6. Krawiec, K., Wieloch, B.: Analysis of semantic modularity for genetic programming. Foundations of Computing and Decision Sciences 34(4), 265–285 (2009)
7. Moraglio, A., Poli, R.: Topological Interpretation of Crossover. In: Deb, K., Poli, R., Banzhaf, W., Beyer, H.G., Burke, E., Darwen, P., Dasgupta, D., Floreano, D., Foster, J., Harman, M., Holland, O., Lanzi, P.L., Spector, L., Tettamanzi, A., Thierens, D., Tyrrell, A. (eds.) GECCO 2004, Part I. LNCS, vol. 3102, pp. 1377–1388. Springer, Heidelberg (2004)
8. Johnson, C.G.: Genetic Programming Crossover: Does It Cross over? In: Vanneschi, L., Gustafson, S., Moraglio, A., De Falco, I., Ebner, M. (eds.) EuroGP 2009. LNCS, vol. 5481, pp. 97–108. Springer, Heidelberg (2009)
9. Archer, A.F.: A modern treatment of the 15 puzzle. American Mathematical Monthly 106, 793–799 (1999)
10. The On-line Encyclopedia of Integer Sequences, `http://oeis.org`
11. Altenberg, L.: Modularity in evolution: Some low-level questions. In: Rasskin-Gutman, D., Callebaut, W. (eds.) Modularity: Understanding the Development and Evolution of Complex Natural Systems, pp. 99–128. MIT Press, Cambridge (2005)
12. Watson, R.A.: Compositional Evolution: The impact of Sex, Symbiosis and Modularity on the Gradualist Framework of Evolution, NA. Vienna series in theoretical biology. MIT Press (February 2006)
13. Krawiec, K.: Semantically embedded genetic programming: automated design of abstract program representations. In: Krasnogor, N., et al. (eds.) GECCO 2011: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, Dublin, Ireland, pp. 1379–1386. ACM (2011)