Potential Fitness for Genetic Programming

Krzysztof Krawiec and Przemysław Polewski
Institute of Computing Science, Poznan University of Technology
Piotrowo 2, 60965 Poznań, Poland
kkrawiec@cs.put.poznan.pl, przemyslaw.polewski@gmail.com

ABSTRACT

We introduce *potential fitness*, a variant of fitness function that operates in the space of schemata and is applicable to tree-based genetic programing. The proposed evaluation algorithm estimates the maximum possible gain in fitness of an individual's direct offspring. The value of the potential fitness is calculated by analyzing the context semantics and subtree semantics for all contexts (schemata) of the evaluated tree. The key feature of the proposed approach is that a tree is rewarded for the correctly classified fitness cases, but it is not penalized for the incorrectly classified ones, provided that such errors are recoverable by substitution of an appropriate subtree (which is however not explicitly considered by the algorithm). The experimental evaluation on a set of seven boolean benchmarks shows that the use of potential fitness may lead to better convergence and higher success rate of the evolutionary run.

Categories and Subject Descriptors: I.2.8 [Problem Solving, Control Methods, and Search]: Heuristic methods

General Terms: Algorithms

Keywords: Evolutionary Computation, Genetic programming, Context, Semantics

1. INTRODUCTION

Most variants of fitness functions used in evolutionary computation (EC) analyze individual's actual behavior (phenotype) and usually does not explicitly consider its potential offspring. The rationale for this is at least twofold. Firstly, this is consistent with the biological evolution, which operates here and now, and, to our current knowledge, has no means of predicting the performance of individual's offspring. Secondly, most of genotypic representations used in EC make it difficult to tell in advance what are the odds for a specific individual to give rise to a successful lineage.

We hypothesize that having such a foresight would be profitable for the convergence of the evolutionary run. One can argue that the fitness function as we know it already does this job — if it didn't promote the individuals that are likely to produce well-performing offspring, we would not observe any convergence at all. In other words, an individual performs well due to some fragments of its genetic code. That performance makes him being promoted by the evolutionary process so that some portions its code may be reused in its offspring. However, in regular EC, such code portions are never examined explicitly by the fitness function, neither in syntactic nor in semantic sense.

The approach prestented in this paper is more explicit, meaning that it semantically investigates the code of the evaluated individual. It does so in order to predict the chance of success for the individual's potential offspring in a more precise way than the regular fitness function. More precisely, it estimates the fitness of the best possible child of the individual, assuming that such a child is obtained from the evaluated individual by modifying it by means of a single-point mutation or a single-point crossover. In other terms, an individual is rewarded for its *potential fitness* rather than its *actual fitness*; hence the name of the approach. The potential fitness may be calculated at a reasonable computational cost for the tree-based genetic programming (GP) applied to boolean-valued problems.

This paper is inspired by the work of McPhee *et al.* on subtree semantics and context semantics [7] and indirectly on Poli's and Page's study on subsymbolic node representations for GP [8]. In another related paper, Majeed *et al.* [6] generalize the evolved trees to schemata and explicitly consider their contributions to the fitness of the tree. The authors claim that their method may be helpful for module discovery and identification of introns, however, they do not provide for theoretical or empirical evidence for that hypothesis. In the technical implementation, our method exhibits also some similarity to the variant of the context-aware crossover proposed by Majeed and Ryan [5], which finds the best possible context for a randomly chosen subtree in the child. Their crossover operator selects randomly a subtree in the first parent and tries to substitute it at every possible position in the second parent. Each such tree is evaluated, and the best of them is appointed as an offspring. The context-aware crossover leads to significantly better performance than the standard GP on the problems of symbolic regression, multiplexer, and lawnmower. Nevertheless, Majeed and Ryan do not refer to context semantics and focus exclusively on crossover; the approach presented here, on the contrary, abstracts from crossover and influences only the fitness function.

GECCO'08, July 12–16, 2008, Atlanta, Georgia, USA.
ACM 978-1-60558-131-6/08/07.

The explicit analysis of the context semantics used in our approach may be linked to the study by Lones and Tyrrel [2], who introduce the notion of implicit context. They use it within their enzyme GP model, where each program component (part of the solution), biological enzymes alike, has a set of behavioral (phenotypic) properties expressed as a 'shape'. Using shape, a program component expresses the properties of the other component(s) it would like to receive input from. In other words, the shape declares the component's expected role within a program. The conducted experiments confirm the authors' hypothesis that the use of implicit context promotes evolvability.

2. THE POTENTIAL FITNESS

In this section, we introduce the potential fitness as an extension of the standard tree-based GP paradigm [1]. Two assumptions are necessary at this point: (i) the task to be solved has to be defined in the boolean domain; (ii) the *desired value* to be returned by GP trees has to be known for each fitness case. Most of the popular boolean GP benchmarks like parity and multiplexers fulfill these conditions. However, the tasks that require running a simulation to calculate fitness do not fulfill condition (ii).

Following [8] and [7], we define a *context* as an expression tree with a temporarily removed subtree marked by the # character, referred to as *insertion point*. Given a GP tree of n nodes, it is possible to build $n-1$ nontrivial contexts from it by removing particular nodes (we discard the trivial case of removing the entire tree). For instance, the three-node tree `(or x y)` gives rise to two contexts: `(or # y)` and `(or x #)`. As pointed out in [7], a context is essentially equivalent to a schemata, with the important difference that it is not intended to represent a *set* of trees.

Context semantics, or *semantics* for short, is a string of zeros, ones, plus signs, and minus signs ($\{0,1,+,-\}$) [7]. A position in the context semantics corresponds to a specific fitness case in the fitness set, so that the string length equals the number of fitness cases. For a specific context, the character at the ith position in the corresponding context semantics reflects the dependency of the tree outcome on the value substituted for #, assuming the remaining part of the context uses the ith fitness case for calculation. If the value returned by the tree does not depend on the value substituted for # when processing the ith fitness case, the ith position in the context semantics contains that value (0 or 1). Otherwise, two cases are possible: the tree either returns the same value as the value substituted for # or the negation of the value substituted for #. These scenarios are marked by the presence of '+' or '−' characters, respectively, at the ith position of the context semantics.

Let us demonstrate the calculation of the context semantics on the tree `(or x y)`, assuming the following ordered list of fitness cases $(x, y) = (0,0),(0,1),(1,0),(1,1)$. Our goal is to compute the context semantics of the context `(or # y)`. For the first and the third fitness case, the tree returns the same value as the value substituted for #. For the second and the fourth fitness case, the tree returns 1 no matter what we substitute for #. Thus, the context semantics of the context `(or # y)` is +1+1. Analogously, the context semantics of the context `(or x #)` is 00+-.

Let us note that, in general, the deeper the location of the # symbol in the tree, the more *fixed* the semantics, i.e., the more 0's and 1's it contains. In particular, the semantics of a context with # replacing an intron — a subtree that has no impact on the tree output — is completely fixed, i.e., contains exclusively 0's and 1's. For instance, assuming the same fitness set as above, the context semantics of the context `(or x (or # true))` is 0011, because # does not influence the tree output. Moreover, *all* the nodes in a subtree comprising an intron have *the same* fixed semantics. This property significantly speeds up our algorithm, which needs to calculate the context semantics for all possible insertion points of a tree.

The distinction of the fixed and non-fixed elements in the context semantics is essential for our idea of potential fitness. We start with the observation that context semantics clearly tells apart the fitness cases for which the semantics is fixed (0's and 1's) from those for which the semantics is non-fixed (pluses and minuses). In other words, the fixed part of the semantics indicates the fitness cases for which the return value of the context cannot be changed (improved or deteriorated), no matter what we substitute in place of #. The non-fixed part of the semantics, on the contrary, indicates the fitness cases for which the return value of the context may be changed by substituting an appropriate subtree in place of #.

The potential fitness function estimates the maximum fitness that may be obtained by appropriately substituting a single insertion point. For this purpose, we first define the *score* of a context in the following way. We start with score=0 and iterate over the corresponding context semantics. For a fixed value of semantics (0 or 1) the score is incremented if that value is consistent with the tree's desired output, and decremented otherwise. The non-fixed elements of the context semantics are ignored.

As a demonstration, let us assume that the formerly considered tree `(or x y)` belongs to the population that tries to solve the task given by the following sequence of desired return values: 1100. We showed above that the semantics of the context `(or # y)` is +1+1. As the non-fixed fixed elements are ignored during score calculation, only the second and the fourth elements of this semantics are relevant. In the former case, the semantics is consistent with the tree's desired output, so the score is incremented and amounts momentarily to 1. However, for the last, fourth fitness case, the semantics is different from the tree's desired output. This causes the score to be decreased and brings its back to its original value of zero. Thus, the overall score of this context amounts to 0.

The score function rewards a context for hits and penalizes it for misses. However, it does not care about the remaining fitness cases, i.e., those ones that could be correctly classified provided an appropriate tree substituted in place of #. Two contexts that are fixed to a different extent (i.e., one of them being fixed at 2 positions and the other one having 4 fixed positions) may have the same score. Clearly, for n fitness cases, the worst possible score is $-n$ and the best possible score (ideal) is n.

The *potential fitness* (PF) of a tree is simply the maximum of scores of all its contexts[1]. In other words, the potential fitness finds the best possible context with respect to evolvability, i.e., such a location in the tree that would result in

[1]Technically, to avoid negative fitness values, the potential fitness is the maximum score *plus* n. However, we do not show that offset in the text to preserve better correspondence between these two quantities for the reader.

the best fitness when substituted by an appropriate subtree. Let us emphasize that we take into account only one such context. Thus, even if the tree has many contexts with the maximal score, individual's fitness will be the same as if it had just one such context.

We hypothesize that potential fitness will promote individuals that offer significant chance of producing well-performing offspring. Many of such individuals remain undiscovered when evaluated by a regular fitness function, because their actual performance at the root node is only modest and does not reflect the possibility of improvement.

3. THE ALGORITHM

To calculate the potential fitness (PF), it is convenient to compute first the semantics of all subtrees of the tree being evaluated. By *subtree semantics* we mean a string of zeros and ones returned by the subtree for the consecutive fitness cases [7]. Similarly to context semantics, the position in that string corresponds to a specific fitness case. For instance, given the aforementioned fitness set, the semantics of the tree `(or x y)` is 0111. Obviously, calculating the semantics of all subtrees of a tree does not increase the computational expense when compared to the regular fitness function, as all tree nodes have to be anyway evaluated.

After computing the semantics of all subtrees, the calculation of context semantics is a straightforward top-down process that starts at the root node. McPhee *et al.* have shown that the semantics of a particular context may be easily determined based on (i) the function implemented by the parent node of the insertion point, (ii) the context semantics of the parent node, and (iii) the subtree semantics of the remaining arguments (siblings) of the parent node [7]. For a given two-argument parent node, there are only $4 \times 2 = 8$ combinations of the parent's context semantics $\{0, 1, +, -\}$ and the subtree semantics of the sibling branch $\{0, 1\}$, which may be conveniently tabelarized (see Table 2 in [7]).

To assess the computational expense of PF calculation, it is convenient to resort to the unit of Genetic Programming Operation (GPO). We assume that 1 GPO is equivalent to the cost of evaluation of a single tree node on a single fitness case. For simplicity, in the following we limit our interest to a single fitness case, unless otherwise stated.

In GP, the cost of calculating the value returned by a n-node tree amounts to n GPO. Assuming that the cost of comparing the returned value with the desired value is approximately the same as a single GPO, the overall cost of evaluating an individual on a single fitness case is $n+1$ GPO.

For PF, the tree must be first calculated in a regular way, at the cost of $n + 1$ GPO. Then, we need to propagate the context semantics in a top-down manner described earlier. To this aim, at each tree node, we calculate the value of context semantics based on the function realized by the parent node of the insertion point, the semantics of the parent node, and the semantics of the remaining arguments of the parent node. This boils down to a simple table lookup operation, so it is reasonable to assume that it costs us 1 GPO per node, hence n GPO for the entire tree. Finally, at each node, we have to calculate the score based on the context semantics and the desired output, which, again, may be assumed to require an effort of 1 GPO per node. Thus, the overall cost of PF for a tree of size n totals to $3n + 1$ GPO, i.e., approximately three times more than the regular GP.

At first sight, tripling the computational cost of standard GP may seem a high price to pay for the potential increase of success rate. The computational effort required by PF may be however easily reduced based on two observations that concern the top-down traversing of the tree when calculating the scores. Firstly, as mentioned before, as soon as we reach a node N with a completely fixed context semantics (only 0's and 1's in the context semantics string), we can immediately conclude that all the nodes in the subtree below N may be skipped, as their context semantics is the same. Secondly, at any node N, it is easily to calculate what is the best possible score s that may be attained in the subtree below N. This allows us to apply the Branch & Bound principle: if s is not greater than the best score found so far, the subtree may be discarded from the search.

4. THE EXPERIMENT

In the following experiment, we analyze the impact of the potential fitness on the performance of genetic programming applied to popular boolean problems. Our baseline is genetic programming with standard Koza-I-style settings (without ADFs) [1], referred simply to as GP in the following. However, we found out that using the standard GP as a yardstick for PF would flaw the comparison, as the way we define the score is not the only factor that may potentially boost the convergence of a PF run. The other one is the fact that, by maximizing the score over *all* the contexts (all tree nodes), the potential fitness considers many more trees than the regular GP. Thus, the differences to be observed between PF and GP are the joint effect of using the score *and* a more intense exploration of the search space.

To make the comparison between GP and PF fair, we come up with another control approach, termed GPS (GP + subtrees). Effectively, GPS is a hybrid of GP and PF: similarly to PF, it considers all tree nodes, but evaluates them using the fitness function instead of the score. The fitness of an individual is the maximum of the fitnesses of its subtrees. Thanks to that, GPS has the chance of examining the number of solutions that is comparable to the number of solutions visited by PF. The theoretical computational effort of GPS amounts to 2 GPO, i.e., in between of the efforts of GP ($n + 1$ GPO) and PF ($3n + 1$ GPO).

The experiment involved seven instances of the following problems: odd-parity, multiplexer, and comparators (see Table 1). The former two problems are well-known in the GP community; the latter one is an n-bit problem and consists in evolving an expression that returns 1 if the binary number encoded in the least significant $n/2$ bits of the input vector is greater than the number encoded in the most significant $n/2$ bits of the input vector, and 0 otherwise.

For each problem, 400 evolutionary runs have been carried out. All the methods have been implemented by extending the ECJ library written in Java [3]. The evolutionary settings were the same for all three methods and followed exactly those specified in ECJ, including discarding the mutation operator to preserve the compatibility with Koza's work [1]: generational GP, population size 1024, 100 generations, max. tree depth 17, tree-swapping crossover, crossover probability 0.9, reproduction probability 0.1. For GP and GPS, the fitness is simply the number of hits (the correctly classified fitness cases). To speed up the processing, we packed the desired output values and subtree semantics into integers *a la* sub-machine-code GP proposed in [8].

Table 1: The performance of GP, GPS, and PF, averaged over 400 evolutionary runs. Statistically significant differences in hit rate marked in bold (t-test, significance level 0.01).

	Success rate			Ideal found in generation			Best-of-run hit rate			Best-of-run tree size		
	GP	GPS	PF	GP	GPS	PF	GP	GPS	PF	GP	GPS	PF
Odd-4-parity	0.755	0.755	0.918	33.1	33.1	23.8	15.67	15.69	**15.90**	243.8	260.8	242.7
Odd-5-parity	0.048	0.055	0.220	74.3	67.1	69.9	28.39	28.49	**30.05**	402.8	416.4	452.2
Odd-6-parity	0.000	0.000	0.000	—	—	—	50.43	50.88	**53.45**	455.8	461.8	523.3
Mux-6	0.995	0.998	1.000	9.2	11.4	11.1	63.99	63.99	64.00	49.8	107.7	83.8
Mux-11	0.283	0.145	0.530	61.1	68.4	63.0	1950.87	1919.51	**1996.79**	275.2	324.2	357.4
Cmp-4	0.990	1.000	1.000	7.6	8.6	6.2	15.99	16.00	16.00	67.1	87.7	69.6
Cmp-6	0.390	0.178	0.740	53.8	57.6	44.3	62.97	62.46	**63.68**	230.1	245.2	256.9

Table 2: The performance of GP, GPS, and PF, averaged over 400 runs with lexicographic parsimony pressure. Statistically significant differences in hit rate marked in bold (t-test, significance level 0.01).

	Success rate			Ideal found in generation			Best-of-run hit rate			Best-of-run tree size		
	GP	GPS	PF	GP	GPS	PF	GP	GPS	PF	GP	GPS	PF
Odd-4-parity	0.830	0.878	0.928	26.4	27.2	23.1	15.73	15.82	**15.91**	117.6	116.3	157.8
Odd-5-parity	0.103	0.140	0.295	69.9	70.3	73.1	28.99	29.32	**30.37**	174.0	179.4	254.1
Odd-6-parity	0.005	0.010	0.018	71.0	72.0	85.4	51.70	52.83	**54.45**	239.0	234.5	338.4
Mux-6	0.998	0.998	1.000	8.1	7.5	10.5	63.98	63.99	64.00	26.7	33.2	60.6
Mux-11	0.328	0.278	0.535	54.0	48.8	61.3	1950.05	1922.34	**1988.01**	132.7	115.1	217.9
Cmp-4	0.910	0.843	0.998	7.2	7.3	6.3	15.91	15.82	**15.99**	45.3	43.5	53.3
Cmp-6	0.250	0.153	0.738	52.7	52.3	43.9	61.95	61.12	**63.56**	76.2	64.6	137.8

Table 1 compares the outcomes of the three methods averaged over 400 runs. The table reports the success rate, the mean generation in which the ideal was found (calculated only over those runs that found the ideal), the mean hit rate of the best-of-run individual (calculated over both ideals and non-ideals), and the mean size (number of tree nodes) of the best-of-run individual. For each problem, the ideal hit rate is two to the power of number of bits involved, e.g., $2^6 = 64$ for the Odd-6-parity, Mux-6, and Cmp-6 problems.

In most cases, PF performs significantly better than GP and GPS in terms of success rate and best-of-run hit rate. For the difficult problems (Odd-5-parity, Mux-11, Cmp-6), it is approximately two (or more) times more likely to find the ideal. It often needs also fewer generations to reach the global optimum.

On the other hand, GPS does only slightly better than GP. This clearly indicates that the superiority of PF should be attributed to the concept of score, and not to the fact that all tree nodes are tested against the desired values.

Table 1 demonstrates that all considered algorithms produce similarly sized trees and suffer from the code bloat to a similar extent. To alleviate the negative consequences of such behavior (increased computational effort and memory occupancy), we run another series of experiments, applying the lexicographic parsimony pressure [4]: each time a tournament ends in a draw, the smaller individual is preferred. If there is still draw, a randomly selected contestant wins the tournament.

Table 2 presents the performance of particular methods with lexicographic parsimony pressure turned on. Apart from reducing the average tree size and thus shortening the average run time, parsimony pressure usually increases the success rate and the best-of-run's hit rate. In particular, all the methods become able to occasionally find the optimal solution for the Odd-6-parity problem. PF is still superior in terms of success rate and best-of-run hit rate, though on average it benefits less from the parsimony pressure than GP and GPS: its trees are smaller only about 35% than those evolved without parsimony pressure, whereas the best trees from the GP and GPS runs are reduced by 50% and 60%, respectively. As a result, PF trees are now approximately 60% larger than those produced by the other methods. We hypothesize that this is a side-effect of the potential fitness function that promotes individuals, which, when evaluated by means of the regular fitness function, would get worse evaluation and lose tournaments to equally-performing yet smaller trees.

As it appears, applying parsimony pressure is not always beneficial to the evolutionary process. This is clearly the case with both comparator problems, where the best-of run hit rate deteriorated for both GP and GPS. Interestingly enough, the PF variant seems exempt from this degenerative effect, and we have yet to observe a case where its performance under parsimony pressure is significantly worse than without it. This is also a way to improve the execution times of the PF-driven algorithm compared to the standard GP version, as the smaller tree sizes enable PF to make up for some of the time lost due to more complex calculations.

Figures 1-7 present the mean fitness graphs of GP, PF,

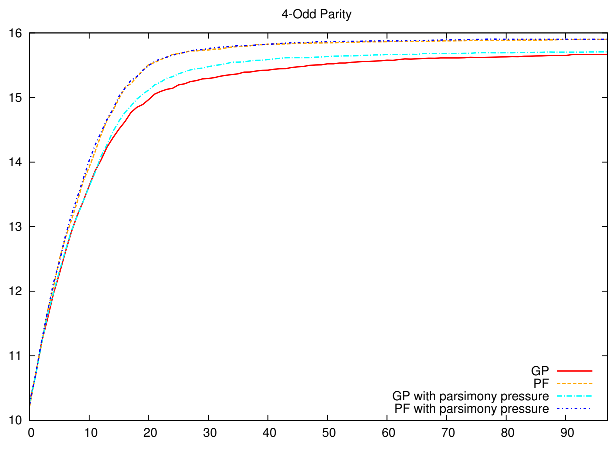

Figure 1: Fitness graph for Odd-4-parity.

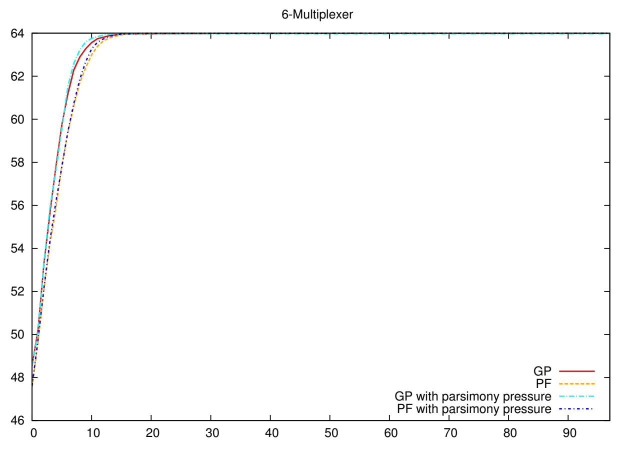

Figure 4: Fitness graph for Mux-6.

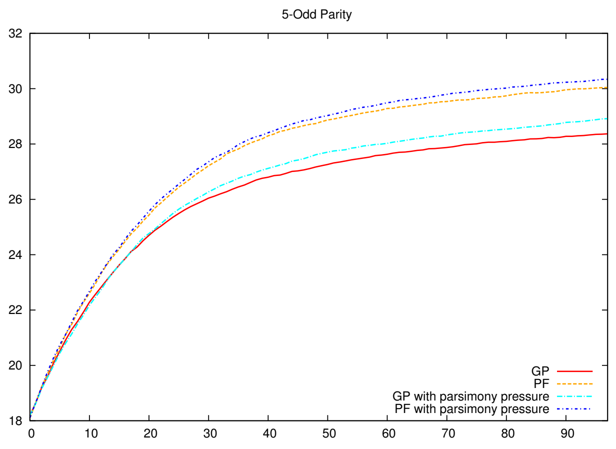

Figure 2: Fitness graph for Odd-5-parity.

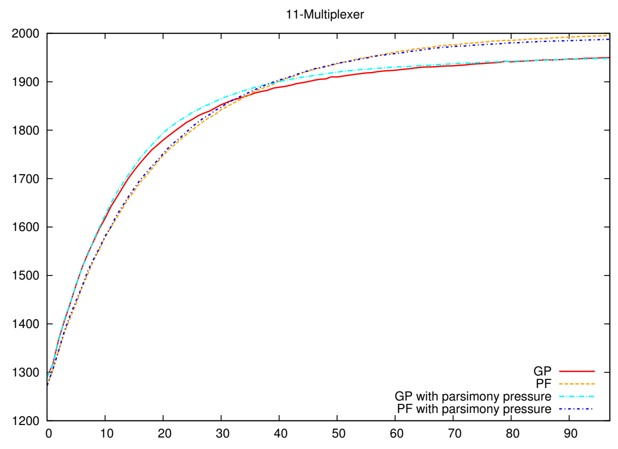

Figure 5: Fitness graph for Mux-11.

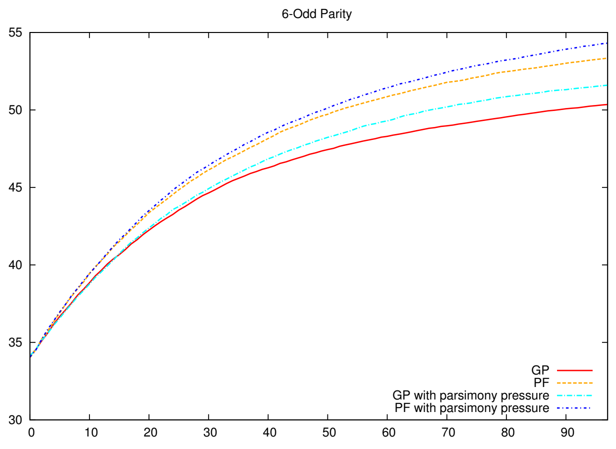

Figure 3: Fitness graph for Odd-6-parity.

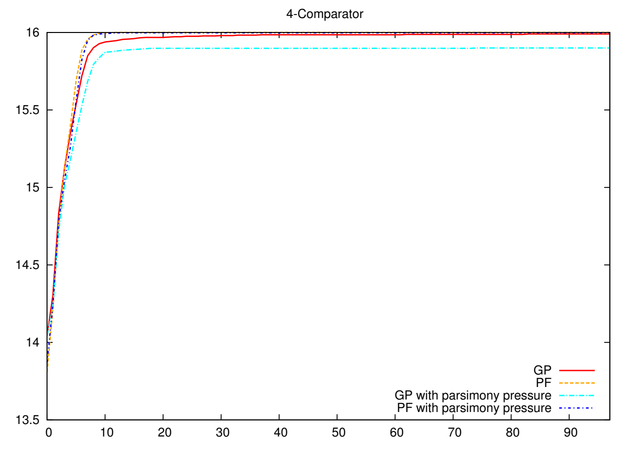

Figure 6: Fitness graph for Cmp-4.

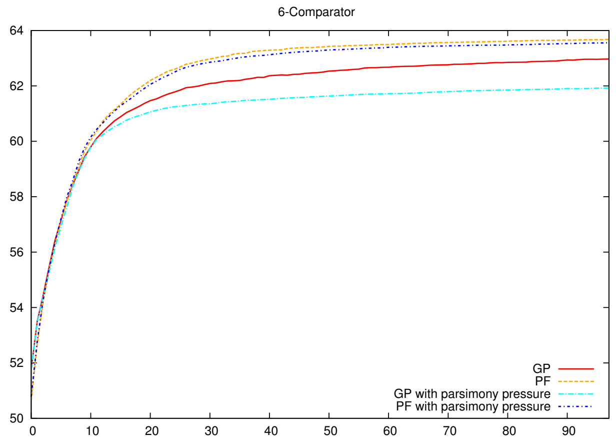

Figure 7: Fitness graph for Cmp-6.

and their variants equipped with parsimony pressure (mean number of hits as a function of generation number). In most cases, PF shows faster convergence from the very beginning of the evolutionary run. Only for Mux-11 (see Fig. 5), PF initially lags behind GP, but around 35th generation starts to overtake it. Of course, for a very easy problem like Mux-6 (Fig. 4), all methods almost always converge quickly to the ideal solution and their comparison is inconclusive.

5. CONCLUSIONS

The potential fitness function is a relatively simple mechanism that estimates the maximum possible gain in fitness of individual's direct offspring. The experimental verification on popular benchmark problems demonstrates that potential fitness provides better convergence and chance of success than the regular GP. The price we pay for that is an extra computational cost, which currently amounts to approx. 200% of the GP runtime. This overhead is the major drawback of our method and should be addressed in the future research. Nevertheless, the encouraging results in terms of success rate and hit rate give rise to other extensions of this idea.

McPhee *et al.* pointed out in [7] that context semantics may be a useful means for building intelligent recombination operators that promote combining subtrees and contexts with compatible semantics. It would be interesting to make use of this option together with the potential fitness. This is especially appealing taking into account that all context semantics of all individuals are computed anyway by the potential fitness function, so the extra overhead of introducing an intelligent operator would be probably low.

The potential fitness algorithm in its current form performs a kind of one-step look-ahead. A natural extension of this procedure would be to search for another promising context (insertion point) after the first one has been found. However, this would be probably computationally expensive. Even more importantly, the probability that *two* such insertion points would be successfully exploited by the individual's offspring is much smaller than the probability of exploiting just one of them, as it was the case in this paper.

The encouraging outcomes reported here may be partially attributed to the ease of defining the notions of subtree semantics and context semantics for the boolean problems. Further research could aim at extending the approach beyond this problem category. However, this seems to be a serious challenge, as it requires a complete re-definition of the notions of semantics and score. One may expect them to become more complicated in the domain of, e.g., real-valued functions and symbolic regression. Unfortunately, this may imply also higher computational overhead and render the whole idea prohibitively expensive in terms of the computational effort.

Acknowledgment

This research has been supported by the Ministry of Science and Higher Education grants # N N519 3505 33 and DS 91-452.

6. REFERENCES

[1] Koza, J. R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[2] Lones, M. A., and Tyrrell, A. M. Modelling biological evolvability: Implicit context and variation filtering in enzyme genetic programming. *BioSystems 76*, 1–3 (Aug.–Oct. 2004), 229–238.

[3] Luke, S. ECJ evolutionary computation system, 2002. (http://cs.gmu.edu/ eclab/projects/ecj/).

[4] Luke, S., and Panait, L. Lexicographic parsimony pressure. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference* (New York, 9-13 July 2002), 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, Eds., Morgan Kaufmann Publishers, pp. 829–836.

[5] Majeed, H., and Ryan, C. A less destructive, context-aware crossover operator for GP. In *Proceedings of the 9th European Conference on Genetic Programming* (Budapest, Hungary, 10 - 12 Apr. 2006), P. Collet, M. Tomassini, M. Ebner, S. Gustafson, and A. Ekárt, Eds., vol. 3905 of *Lecture Notes in Computer Science*, Springer, pp. 36–48.

[6] Majeed, H., Ryan, C., and Azad, R. M. A. Evaluating GP schema in context. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation* (Washington DC, USA, 25-29 June 2005), H.-G. Beyer, U.-M. O'Reilly, D. V. Arnold, W. Banzhaf, C. Blum, E. W. Bonabeau, E. Cantu-Paz, D. Dasgupta, K. Deb, J. A. Foster, E. D. de Jong, H. Lipson, X. Llora, S. Mancoridis, M. Pelikan, G. R. Raidl, T. Soule, A. M. Tyrrell, J.-P. Watson, and E. Zitzler, Eds., vol. 2, ACM Press, pp. 1773–1774.

[7] McPhee, N. F., Ohs, B., and Hutchison, T. Semantic building blocks in genetic programming. In *Genetic Programming* (2008), M. O'Neill, L. Vanneschi, S. Gustafson, A. I. E. Alcázar, I. D. Falco, A. D. Cioppa, and E. Tarantino, Eds., vol. 4971 of *LNCS*, Springer, pp. 134–145.

[8] Poli, R., and Page, J. Solving high-order boolean parity problems with smooth uniform crossover, sub-machine code GP and demes. *Genetic Programming and Evolvable Machines 1*, 1/2 (Apr. 2000), 37–56.