Memetic Semantic Genetic Programming

Robyn Ffrancon, Marc Schoenauer

Memetic Semantic Genetic Programming

To appear in GECCO 2015

Robyn Ffrancon
TAO, INRIA Saclay
Univ. Paris-Sud
Orsay, France
rffrancon@gmail.com

Marc Schoenauer
TAO, INRIA Saclay
Univ. Paris-Sud
Orsay, France
marc.schoenauer@inria.fr

ABSTRACT

Semantic Backpropagation (SB) was introduced in GP so as to take into account the semantics of a GP tree at all intermediate states of the program execution, i.e., at each node of the tree. The idea is to compute the optimal "should-be" values each subtree should return, whilst assuming that the rest of the tree is unchanged, so as to minimize the fitness of the tree. To this end, the Random Desired Output (RDO) mutation operator, proposed in [17], uses SB in choosing, from a given library, a tree whose semantics are preferred to the semantics of a randomly selected subtree from the parent tree. Pushing this idea one step further, this paper introduces the Local Tree Improvement (LTI) operator, which selects from the parent tree the overall best subtree for applying RDO, using a small randomly drawn static library. Used within a simple Iterated Local Search framework, LTI can find the exact solution of many popular Boolean benchmarks in reasonable time whilst keeping solution trees small, thus paving the road for truly memetic GP algorithms.

1. INTRODUCTION

Memetic Algorithms [9] have become increasingly popular recently (see e.g., [11]): the hybridization between evolutionary algorithms and non-evolutionary (generally local) search has resulted in highly efficient algorithms in practice, mainly in the field of combinatorial optimization.

There have been, however, very few works proposing memetic approaches in Genetic Programming (e.g., the term "GP" does not even appear in [11]). The main reason is certainly the lack of recognized efficient local optimization procedures in the usual GP search spaces (trees, linear-coded GP, Cartesian GP, ...).

However, from the EC point of view, a stochastic local search procedure in a given search space can be viewed as a specific mutation operator in which choices are biased, using domain-specific knowledge, toward improving the fitness of the parent individual. Clearly, the boundary between memetic and genetic operators is far from being crisp (a well-known example is the record-winning Evolutionary TSP solver [10]).

Historical Genetic Programming (GP) [5] evolves trees by manipulating subtrees in a syntactical way, blind to any possible bias toward fitness improvement, as is the rule in 'pure' Evolutionary Algorithms: Subtree crossover selects nodes at random from both parents and swaps them, along with their rooted subtrees. Similarly, point mutation randomly selects one subtree and replaces it with a randomly constructed other subtree. Subtrees are probabilistically (most often randomly) selected from the parents to which they belong, as opposed to their own usefulness as functions.

More recently, several works have addressed this issue, gradually building up the field that is now called *Semantic GP*. For a given set of values of the problem variables, the *semantics* of a subtree within a given tree is defined as the vector of values computed by this subtree for each set of input values in turn. In Semantic GP, as the name implies, the semantics of all subtrees are considered as well as the semantics of the context in which a subtree is inserted (i.e., the semantics of the its siblings), as first proposed and described in detail in [8] (see also [15] for a recent survey). Several variation operators have been proposed for use within the framework of Evolutionary Computation (EC) which take semantics into account when choosing and modifying subtrees.

One such semantically oriented framework is Behavioural Programming GP [6]. The framework facilitates the use of Machine Learning techniques in analyzing the internal semantics of individual trees so as to explicitly identify potentially useful subtrees. It constitutes a step towards archiving and reusing potentially useful subtrees based on the merits of their functionality rather than solely on the fitnesses of the full trees from which they derive. However, the usefulness of these subtrees is assessed globally, independent of the context to which they are to be inserted.

Semantic Backpropagation (SB) [17, 7, 12] addresses this issue: given a set of fitness cases, SB computes, for each subtree of a target tree, the desired outputs which they should return so as to minimize the fitness of the tree, assuming that the rest of the tree is unchanged. A small number of specialised operators have been proposed which exploit SB, the *Random Desired Output* (RDO) mutation operator is one example [17]. This operator firstly randomly picks a target node in the parent tree, then replaces this target node with a tree from a given *library* of trees whose outputs best match the desired values of the target node [17, 12]. Because it

replaces, if possible, the tree rooted at the selected node of the parent tree with a tree that matches the local semantics of that node, RDO can also be viewed as a first step towards a memetic operator.

Building on RDO, the present work proposes *Local Tree Improvement* (LTI), a local search procedure in the space of GP trees which extends RDO with another bias toward a better fitness: Rather than selecting the target node in the parent tree at random, Local Tree Improvement selects the best possible semantic match between all possible nodes in the parent tree and all trees in the library. The resulting variation operator in the space of trees is then used within a standard Iterated Local Search procedure: LTI is repeatedly applied to one single tree with the hope of gradually improving the tree fitness – whereas a single application of LTI is not guaranteed to do so.

The prerequisites for the LTI procedure are those of Semantic Backpropagation[1]: i) a fitness defined by aggregation of some error on several fitness cases ii) a way to compute from the current context (as defined in [8]), at each node and for each fitness case, the optimal values which each node should return so that the the whole tree evaluates to the exact expected values. This is indeed possible in the case for Boolean, Categorical, and Symbolic Regression problems. However, only Boolean problems will be addressed in this work with Categorical and Regression problems left for future work (Section 7).

The rest of the paper firstly recalls (Section 2), in the interest of completeness, the general principles of Semantic Backpropagation, though instantiated in the Boolean context, and thus adopting a slightly different point of view (and notations) to [12]. Section 3 then details LTI, the main contribution of this work, and how it is used within the Iterated Local Search procedure ILTI. Experiments conducted with ILTI are presented in Section 4: The first goal of these experiments is to demonstrate the efficiency of ILTI as a stand-alone optimization procedure; the second goal is to study the sensitivity of the results with respect to the most important parameter of ILTI, the size of the library. Similarities and differences with previous works dealing with memetic GP are discussed in Sections 6, while links with previous Semantic Backpropagation works are highlighted and discussed in Section 7, together with related possible further research paths.

2. SEMANTIC BACKPROPAGATION

The powerful idea underlying Semantic Backpropagation is that, for a given tree, it is very often possible to calculate the optimal outputs of each node such that the final tree outputs are optimized. Each node (and rooted subtree) is analyzed under the assumption that the functionality of all the other tree nodes are optimal. In effect, for each node, the following question should be asked: What are the optimal outputs for this node (and rooted subtree) such that its combined use with the other tree nodes produce the optimal final tree outputs? Note that for any given node, its optimal outputs do not depend on its semantics (actual outputs). Instead, they depend on the final tree target outputs, and the actual output values (semantics) of the other nodes within the tree.

In utilizing the results of this analysis, it is possible to produce local fitness values for each node by comparing their actual outputs with their optimal outputs. Similarly, a fitness value can be calculated for any external subtree by comparing its actual outputs to the optimal outputs of the node which it might replace. If this fitness value indicates that the external subtree would perform better than the current one, then the replacement operation should improve the tree as a whole.

2.1 Hypotheses and notations

We suppose that the problem at hand comprises n fitness cases, were each case i is a pair (x_i, f_i). Given a loss function ℓ, the goal is to find the program ($tree$) that minimizes the global error

$$Err(tree) = \sum_{i=1}^{i=n} \ell(tree(x_i), f_i) \quad (1)$$

where $tree(x_i)$ is the output produced by the tree when fed with values x_i.

In the Boolean framework for instance, each input x_i is a vector of Boolean variables, and each output f_i is a Boolean value. A trivial loss function is the Hamming distance between Boolean values, and the global error of a tree is the number of errors of that tree.

In the following, we will be dealing with a *target tree* T and a *subtree library* $\mathcal{L}$. We will now describe how a subtree (node location) s is chosen in T **together with** a subtree s^* in $\mathcal{L}$ to try to improve the global fitness of T (aggregation of the error measures on all fitness cases) when replacing, in T, s with s^*.

2.2 Tree Analysis

For each node in T, the LTI algorithm maintains an *output vector* and an *optimal vector*. The i^{th} component of the output vector is the actual output of the node when the tree is executed on the i^{th} fitness case; the i^{th} component of the optimal vector is the value that the node should take so that its propagation upward would lead T to produce the correct answer for this fitness case, all other nodes being unchanged.

The idea of storing the output values is one major component of BPGP [6], which is used in the form of a trace table. In their definition, the last column of the table contained target output values of the full tree – a feature which is not needed here as they are stored in the optimal vector of the root node.

Let us now detail how these vectors are computed. The output vector is simply filled during the execution of T on the fitness cases. The computation of the optimal vectors is done in a top-down manner. The optimal values for the top node (the root node of T) are the target values of the problem. Consider now a given fitness case, and a simple tree with top node A. Denote by a, b and c their output values, and by $\hat{a}$, $\hat{b}$ and $\hat{c}$ their optimal values (or set of optimal values, see below)[2]. Assuming now that we know $\hat{a}$, we want to compute $\hat{b}$ and $\hat{c}$ (top-down computation of optimal values).

[1]though the Approximate Geometric Crossover (AGX) can be defined in a more general context by artificially creating surrogate target semantics for the root node [7, 12].

[2]The same notation will be implicit in the rest of the paper, whatever the nodes A, B and C.

If node A represents operator F, then, by definition

$$a = F(b, c) \tag{2}$$

and we want $\hat{b}$ and $\hat{c}$ to satisfy

$$\hat{a} = F(\hat{b}, c) \text{ and } \hat{a} = F(b, \hat{c}) \tag{3}$$

i.e., to find the values such that A will take a value $\hat{a}$, assuming the actual value of the other child node is correct. This leads to

$$\hat{b} = F^{-1}(\hat{a}, c) \text{ and } \hat{c} = F^{-1}(\hat{a}, b) \tag{4}$$

where F^{-1} is the pseudo-inverse operator of F. In the Boolean case, however, this pseudo-inverse operator is ill-defined. For instance, for the AND operator, if $\hat{a} = 0$ and b=0, any value for $\hat{c}$ is a solution: this leads to set $\hat{c} = \#$, the "don't care" symbol, representing the set $\{0, 1\}$. On the other hand, if $\hat{a} = 1$ and b=0, no solution exists for $\hat{c}$. In this case, $\hat{c}$ is set to the value that does not propagate the impossibility (here for instance, $\hat{c} = 1$). Note that this is an important difference with the Invert function used in [12], where the backpropagation stops whenever either "don't care" or "impossible" are encountered. See the discussion in Section 7.

Function tables for the Boolean operators AND^{-1}, OR^{-1}, NAND^{-1}, and NOR^{-1} are given in Fig. 1. A "starred" value indicates that $\hat{a}$ is impossible to reach: in this case, the 'optimal' value is set for $\hat{c}$ as discussed above.

For each fitness case, we can compute the optimal vector for all nodes of T, starting from the root node and computing, for each node in turn, the optimal values for its two children as described above, until reaching the terminals.

2.3 Local Error

The local error of each node in T is defined as the discrepancy between its output vector and its optimal vector. The loss function ℓ that defines the global error from the different fitness cases (see Eq. 1) can be reused, provided that it is extended to handle sets of values. For instance, the Hamming distance can be easily extended to handle the "don't care" symbol # (for example: $\ell(0, \#) = \ell(1, \#) = 0$). We will denote the output and optimal values for node A on fitness case i as a_i and $\hat{a_i}$ respectively. The local error $Err(A)$ of node A is defined as

$$Err(A) = \sum_i \ell(a_i, \hat{a_i}) \tag{5}$$

2.4 Subtree Library

Given a node A in T that is candidate for replacement (see next Section 3.1 for possible strategies for choosing it), we need to select a subtree in the library $\mathcal{L}$ that would likely improve the global fitness of T if it were to replace A. Because the effect of a replacement on the global fitness are in general beyond this investigation, we have chosen to use the local error of A as a proxy. Therefore, we need to compute the *substitution error* $Err(B, A)$ of any node B in the library, i.e. the local error of node B if it were inserted in lieu of node A. Such error can obviously be written as

$$Err(B, A) = \sum_i \ell(b_i, \hat{a_i}) \tag{6}$$

Then, for a given node A in T, we can find $best(A)$, the set subtrees in $\mathcal{L}$ with minimal substitution error,

$$best(A) = \{B \in \mathcal{L}; Err(B, A) = min_{C \in \mathcal{L}}(Err(C, A)\}) \tag{7}$$

and then define the *Expected Local Improvement* $I(A)$ as

$$I(A) = Err(A) - Err(B, A) \text{ for some } B \in best(A) \tag{8}$$

If $I(A)$ is positive, then replacing A with any node in $best(A)$ will improve the local fitness of A. Note however that this does not imply that the global fitness of T will improve. Indeed, even though the local error will decrease, the cases in error might be different, and this could badly affect the whole tree. Furthermore, even if B is a perfect subtree, resulting in no more error at this level (i.e., $Err(B, A) = 0$), there could remain some impossible values in the tree (the "starred" values in Fig. 1 in the Boolean case) that would indeed give an error when propagated to the parent of A.

On the other hand, if $I(A)$ is negative, no subtree in $\mathcal{L}$ can improve the global fitness when inserted in lieu of A.

Furthermore, trees in $\mathcal{L}$ are unique in terms of semantics (output vectors). In the process of generating the library (whatever design procedure is used), if two candidate subtrees have exactly the same outputs, only the one with fewer nodes is kept. In this way, the most concise generating tree is stored for each output vector. Also, $\mathcal{L}$ is ordered based on tree size, from smallest to largest, hence so is $best(A)$.

3. TREE IMPROVEMENT PROCEDURES

3.1 Local Tree Improvement

Everything is now in place to describe the full LTI algorithm. Its pseudo-code can be found in Algorithm 1. The obvious approach is to choose the node S in T with the smallest error that can be improved, and to choose in $best(S)$ the smallest tree, to limit the bloat.

However, the selection process when no node can be improved is less obvious. Hence, the algorithm starts by ordering the nodes in T by increasing error (line 1). A secondary technical criterion is used here, the number of # (don't care) symbols in the optimal vector of the node, as nodes with many # symbols are more likely to be improved. The nodes are then processed one by one and selected using a rank-based selection from the order defined above. Note that all selections in the algorithm are done stochastically rather than deterministically, to avoid overly greedy behaviors [4].

Choosing the best improvement implies that if there exists a tree $B \in \mathcal{L}$ whose output vector perfectly matches the optimal vector of a given node $A \in \mathcal{T}$ (i.e., $Err(B, A) = 0$, Eq. 6), there is no need to look further in $\mathcal{L}$. Therefore, A is replaced with B and the algorithm returns (line 11). Otherwise, the algorithm proceeds by computing $best(A)$ for the node current A (lines 19-23). Importantly, the fact that there has been at least an improvement (resp. a decrease of local fitness) is recorded, line 14 (resp. 16). At the end of the loop over the library, if some improvement is possible for the node at hand (line 24), then a tree is selected in its best matches (rank-selection on the sizes, line 25), and the algorithm returns. Otherwise, the next node on the ordered list $\mathcal{L}$ is processed.

If no improvement whatsoever could be found, but some decrease of local fitness is possible (line 28), then a node should be chosen among the ones with smallest decrease. However, it turned out that this strategy could severely

$\hat{c} = \text{AND}^{-1}(\hat{a}, b)$		
$\hat{a}$	b	$\hat{c}$
0	0	#
0	1	0
1	0	1^*
1	1	1
#	0	#
#	1	#

$\hat{c} = \text{OR}^{-1}(\hat{a}, b)$		
$\hat{a}$	b	$\hat{c}$
0	0	0
0	1	0^*
1	0	1
1	1	#
#	0	#
#	1	#

$\hat{c} = \text{NAND}^{-1}(\hat{a}, b)$		
$\hat{a}$	b	$\hat{c}$
0	0	1^*
0	1	1
1	0	#
1	1	0
#	0	#
#	1	#

$\hat{c} = \text{NOR}^{-1}(\hat{a}, b)$		
$\hat{a}$	b	$\hat{c}$
0	0	1
0	1	#
1	0	0
1	1	0^*
#	0	#
#	1	#

Figure 1: Function tables for the AND^{-1}, OR^{-1}, NAND^{-1}, and NOR^{-1}.

Algorithm 1 Procedure LTI(Tree T, library $\mathcal{L}$)

Require: $Err(A)$ (Eq. 5), $Err(B, A)$ (Eq. 6), $A \in T$, $B \in \mathcal{L}$
1: $\mathcal{T} \leftarrow$ all T nodes ordered by $Err\uparrow$, then number of #s$\downarrow$
2: Improvement $\leftarrow False$
3: OneDecrease $\leftarrow False$
4: **for** $A \in \mathcal{T}$ **do** ▷ Loop over nodes in set $\mathcal{T}$
5: Decrease(A) $\leftarrow False$
6: **while** $\mathcal{T}$ not empty **do**
7: $A \leftarrow$ RankSelect($\mathcal{T}$) ▷ Select and remove from $\mathcal{T}$
8: $Best(A) \leftarrow \emptyset$
9: $minErr \leftarrow +\infty$
10: **for** $B \in \mathcal{L}$ **do** ▷ Loop over trees in library
11: **if** $Err(B, A) = 0$ **then** ▷ Perfect match
12: Replace A with B
13: **return**
14: **if** $Err(B, A) < Err(A)$ **then**
15: Improvement $\leftarrow True$
16: **else if** $Err(B, A) > Err(A)$ **then**
17: OneDecrease $\leftarrow True$
18: Decrease(A) $\leftarrow True$
19: **if** $Err(B, A) < minErr$ **then** ▷ Better best
20: $Best(A) = \{B\}$
21: $minErr \leftarrow Err(B, A)$
22: **if** $Err(B, A) = minErr$ **then** ▷ Equally good
23: $Best(A) \leftarrow Best(A) + \{B\}$
24: **if** Improvement **then**
25: $B \leftarrow$ RankSelect($Best(A)$) ▷ Order: size$\uparrow$
26: Replace A with B
27: **return**
28: **if** OneDecrease **then** ▷ At least one decrease
29: $\mathcal{M} \leftarrow \{A \in T$; Decrease(A)$\}$
30: $\mathcal{M} \leftarrow$ top κ from $\mathcal{M}$ ordered by depth$\downarrow$
31: $A \leftarrow$ RankSelect($\mathcal{M}$) ▷ Order: $Err\uparrow$
32: $B \leftarrow$ RankSelect($Best(A)$) ▷ Order: size$\uparrow$
33: Replace A with B
34: **return**
35: $A \leftarrow$ uniformSelection(T) ▷ Random move
36: $B \leftarrow$ randomTree()
37: Replace A with B
38: **return**

damage the current tree (see Fig. 3 and the discussion in Section 5) when the replacement occurred high in the tree. This is why depth was chosen as the main criterion in this case: all nodes A with non-empty $best(A)$ (the nodes with the same errors as A are discarded, to avoid possible loops in the algorithm) are ordered by increasing depth, and a rank-selection is made upon the top κ nodes of that list (line 31). The tree from the library is then chosen based on size (line 32). User-defined parameter κ tunes the relative weights of depth and error in the choice of target node, and was set to 3 in all experiments presented in Section 5 (i.e. depth is the main criterion). Finally, in the event that no improvement nor any decrease can be achieved, a random tree replaces a random node (line 37).

Complexity Suppose that the library $\mathcal{L}$ is of size o. The computation of the output vectors of all trees in $\mathcal{L}$ is done once and for all. Hence the overhead of one iteration of LTI is dominated, in the worst case, by the comparisons of the optimal vectors of all nodes in T with the output vectors of all trees in $\mathcal{L}$, with complexity $n \times m \times o$.

3.2 Iterated LTI

In the previous section, we have defined the LTI procedure that, given a target tree T and a library of subtrees $\mathcal{L}$, selects a node S in T and a subtree S^* in $\mathcal{L}$ to insert in lieu of node S so as to minimize some local error over a sequence of fitness cases. In this section we will address how T and $\mathcal{L}$ are chosen, and how one or several LTI iterations are used for global optimization.

LTI can be viewed as the basic step of some local search, and as such, it can be used within any Evolutionary Algorithm evolving trees (e.g., GP), either as a mutation operators, or as a local search that is applied on some individual in the population at every generation. Such use of local search is very general and common in Memetic Algorithm (see e.g., Chapter 4 in [11]). Because LTI involves a target tree and a library of subtrees, it could be used, too, to design an original crossover operator, in which one of the parents would be the target tree, and the library would be the set of subtrees of the other parents. However, because LTI is an original local search procedure that, to the best of our knowledge, has never been used before, a natural first step should be devoted to its analysis alone, without interference from any other mechanism.

This is why this work is devoted to the study of a (Local) Search procedure termed ILTI, that repeatedly applies LTI to the same tree, picking up subtrees from a fixed library, without any selection whatsoever. This procedure can also be viewed as a (1,1)-EA, or as a random walk with move operator LTI.

One advantage of starting simple is that ILTI does not have many parameters to tune, and will hopefully allow us to get some insights about how LTI actually works. The parameters of ILTI are the method used to create the initial tree (and its parameters, e.g., the depth), the method (and, again, its parameters) used to create the subtrees in the library, the parameter κ for node selection (see previous Section 3.1), and the size of the library. The end of the paper is devoted to some experimental validation of ILTI, and the study of the sensitivity of the results w.r.t. its most important parameter, the library size.

Table 1: Results of the ILTI algorithm for Boolean benchmarks: 30 runs were conducted for each benchmark, always finding a perfect solution. A library size of 450 trees was used. BP columns are the results of the best performing algorithm (BP4A) of [6] (* indicates that not all runs found a perfect solution). The RDO column is taken from [17].

	Run time [seconds]				Program size [nodes]					Number of iterations		
	max	**mean**	min	BP	max	**mean**	min	BP	RDO	max	**mean**	min
Cmp06	9.9	**8.6 ± 0.5**	7.8	15	77	**59.1 ± 7.2**	47	156	185	189	**63.9 ± 34.4**	19
Cmp08	54.8	**19.8 ± 7.8**	14.3	220	191	**140.1 ± 23.2**	81	242	538	2370	**459.1 ± 466.1**	95
Maj06	10.9	**9.5 ± 0.8**	8.4	36	87	**71.2 ± 10.1**	53	280	123	183	**89.1 ± 42.2**	27
Maj08	44.1	**26.7 ± 7.0**	18.2	2019*	303	**235.9 ± 29.8**	185	563*	-	2316	**938.6 ± 519.0**	307
Mux06	11.2	**9.4 ± 0.8**	8.5	10	79	**47.1 ± 11.2**	31	117	215	34	**20.0 ± 6.8**	11
Mux11	239.1	**100.2 ± 40.2**	59.0	9780	289	**152.9 ± 59.0**	75	303	3063	1124	**302.9 ± 216.4**	79
Par06	25.2	**16.7 ± 2.4**	12.5	233	513	**435.3 ± 33.2**	347	356	1601	2199	**814.8 ± 356.6**	326
Par08	854	**622 ± 113.6**	386	3792*	2115	**1972 ± 94**	1765	581*	-	22114	**12752 ± 3603**	6074
Par09	8682	**5850 ± 1250**	4104	-	4523	**4066 ± 186**	3621	-	-	142200	**54423 ± 24919**	31230

4. EXPERIMENTAL CONDITIONS

The benchmark problems used for these experiments are classical Boolean problems that have been widely studied, and are not exclusive to the GP community (see Section 6 also). We have chosen this particular benchmarks because they are used in [12, 6] (among other types of benchmarks). For the sake of completeness, we reiterate their definitions as stated in [6]: The solution to the *v-bit Comparator problem Cmp-v* must return *true* if the $\frac{v}{2}$ least significant input bits encode a number that is smaller than the number represented by the $\frac{v}{2}$ most significant bits. For the *Majority problem Maj-v*, *true* should be returned if more that half of the input variables are true. For the *Multiplexer problem Mul-v*, the state of the addressed input should be returned (6-bit multiplexer uses two inputs to address the remaining four inputs, 11-bit multiplexer uses three inputs to address the remaining eight inputs). In the *Parity problem Par-v*, *true* should be returned only for an odd number of true inputs.

All results have been obtained using an AMD Opteron(tm) Processor 6174 @ 2.2GHz. All of the code was written in Python[3].

In all experiments presented here, the library $\mathcal{L}$ was made of full trees of depth 2: there are hence a possible $64 \times \#variables$ different trees. Experiments regarding other initializations of $\mathcal{L}$ (e.g., other depths, or using the popular ramp-half-and-half method instead of building full trees) are left for further studies. Similarly, the target tree T was initialized as a full tree of depth 2. Several depths have been tested without any significant modification of the results. The only other parameter of LTI (and hence of ILTI) is parameter κ that balances the selection criterion of the target node for replacement in cases where no improvement could be found for any node (Section 3.1, line 30 of Algorithm 1). As mentioned, it was set to 3 here, and the study of its sensitivity is also left for further work.

After a few preliminary experiments, the runs were given strict run-time upper limits: 60s for all easy runs (xxx06), 100s for the more difficult Cmp08 and Maj08, 1000s for the much more difficult Mux11 and Par08, and 10800 (3 hours) for the very difficult Par09 (the only benchmark investigated in this paper that was not reported in [6]). If a run did not return a perfect solution within the time given, it was considered a failure.

[3]The entire code base is freely available at robynffrancon.com/LTI.html

5. EXPERIMENTAL RESULTS

Figure 2 plots standard boxplots of the actual run-time to solution of ILTI for library sizes in {50, 100, 200, 300, 400, 450, 500, 550} for all benchmarks. Note that the (red) numbers on top of some columns indicate the number of runs (out of 30) for library sizes which failed to find an exact solution.

On these plots, a general tendency clearly appears: the run-time increases with the size of the library for easy problems (Cmp06, Mux6, and Maj06, even though sizes 50 and 100 fail once out of 30 runs). Then, as the problem becomes more difficult (see Par06), the results of small library sizes start to degrade, while their variance increases, but all runs still find the perfect solution. For more difficult problems (Cmp08, Mux11, and Maj08) more and more runs fail, from small to medium sizes. Finally, for the very hard Par08 and Par09 problems, almost no run with size 50 or 100 can find the solution, while sizes 200 and 300 still have occasional difficulties to exactly solve the problem. These results strongly suggest that for each problem, there exists an optimal library size, which is highly problem dependent, with a clear tendency: the more difficult the problem the larger the optimal size. A method for determining the optimal size a priori, from some problem characteristics, is left for further work.

Regarding the tree sizes of the solutions, the results (not shown here) are, on the opposite, remarkably stable: all library sizes give approximately the same statistics on the tree sizes (when a solution is found) - hence similar to the results in Table 1, discussed next.

Based on the above, the comparative results of Table 1 use a library of size 450. They are compared with the Boolean benchmark results of the best performing Behavior Programming GP (BPGP) scheme, *BP4A* [6], and when possible with the results of RDO given in [17] or in [12].

We will first consider the success rates. All algorithms seem to solve all problems which they reported on. Note however that results for Par09 are not reported for BP, and results for Par08, Par09 and Maj08 are not reported for RDO. Furthermore, there are some small exceptions to this rule, the "*" in Table 1 for BP, and Par06 for RDO (success rate of 0.99 according to [17]).

Even though it is difficult to compare the run-times of ILTI and BPGP [6] because of the different computing environments, some comments can nevertheless be made regarding this part of Table 1: Firstly, both *BP4A* and ILTI have reported run-times of the same order of magnitude . . . for the

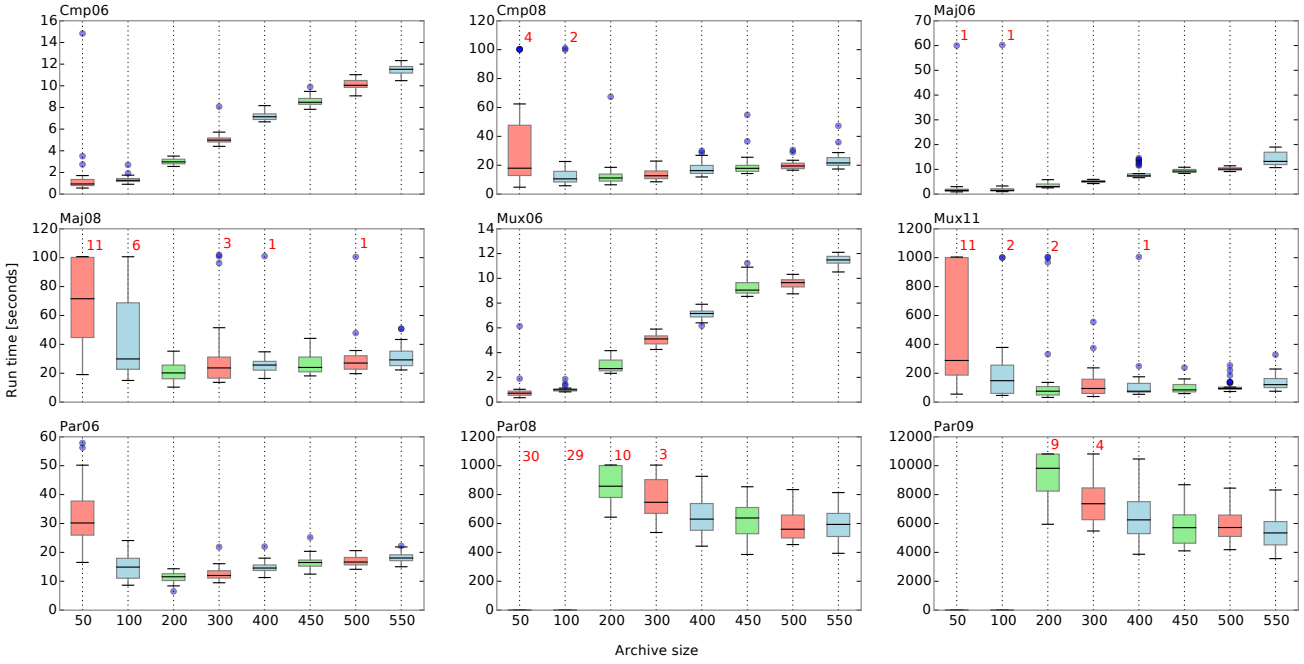

Figure 2: Comparative results: time to solution for different library sizes. The benchmark problems appear in the order of Table 1: Cmp06, Cmp08, Maj06; Maj08, Mux06, Mux11; Par06, Par08 and Par09.

easiest problems. But, when considering the results of more difficult problems, they also suggest that ILTI scales much more smoothly than *BP4A* with problem size/difficulty. ILTI clearly stands out when comparing the run-times of Cmp06 vs Cmp08, Maj06 vs Maj08, Mux06 vs Mux11 and, to a lesser extend, Par06 vs Par08. On the other hand, even though the runtimes were not directly available for the RDO results, the measure of "number of successes per hour" given in [17] suggests that obtaining the exact solution with RDO is one or two orders of magnitude faster than with ILTI or BP.

When considering the average tree size, ILTI seems more parsimonious than *BP4A* by a factor of around 2, with the exception of the parity problems. In particular, for Par08, the average size for *BP4A* is smaller than that of ILTI by a factor of 3. It is clear that the fine details of the parity problems deserve further investigations. On the other hand, RDO results [17] report much larger tree sizes, from twice to 20 times larger than those of ILTI.

Finally, Fig. 3 displays the evolution of the global error (of the whole tree T) during the 30 runs on the difficult problem Par08 for a library size of 450. It compares the behavior of ILTI using two different strategies for selecting the node to be replaced when no improvement could be found. On the left, the straightforward strategy, similar to the one adopted in case an improvement was found, which simply chooses the node with smallest error. On the right, the strategy actually described in Algorithm 1 which first favors depth, and then chooses among the deepest κ nodes based on their substitution error (lines 28-32 of Algorithm 1). As can be seen, the error-based strategy generates far more disastrous increases of the global error. A detailed analysis revealed

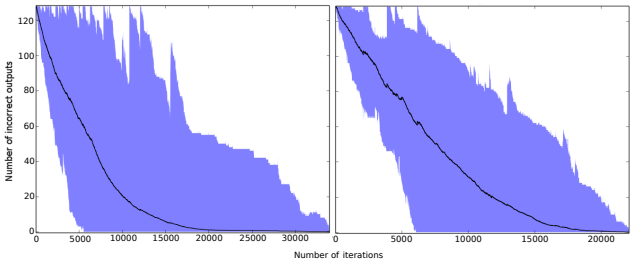

Figure 3: Evolution of the global error during 30 runs of ILTI on Par08 problem: average and range. Left: Node selection by error only; Right: node selection by depth first.

that this happens when nodes close to the root are chosen. Even if they have a small local error, the effect of modifying them might destroy many other good building blocks of the target tree. This was confirmed on the Par09 problem, for which the first strategy simply repeatedly failed – and this motivated the design of the depth-based strategy.

6. RELATED MEMETIC WORK

As mentioned in the introduction, there are very few works at the crossroad between GP and Memetic algorithms. Some papers claim to perform local search by doing offspring filtering, i.e., generating several offspring from the same parents and keeping only the best ones to include in the surviving pool [1]. Though this is indeed some sort of local search (the usual GP variation operators being used as elemen-

tary moves for the local search), it does not really involve any external search procedure with distinct search abilities from those of the underlying GP itself, and could in fact be presented as another parametrization of the GP algorithm. Furthermore, such procedure really makes sense only when the offspring can be approximately evaluated using some very fast proxy for the true fitness evaluation (see the 'informed operators' proposed in the context of surrogate modeling [13]).

Some real memetic search within GP has been proposed in the context of decision trees for classification [16]: the very specific representation of (Stochastic) Decision Trees is used, together with problem-specific local search procedures, that directly act on the subspaces defining the different classes of the classification. Though efficient in the proposed context, this procedure is by no way generic.

Probably the most similar work to LTI have been proposed in [3]: the authors introduce the so-called *memetic crossover*, that records the behavior of all trees during the execution of the programs represented by the trees (though the word 'semantic' is not present in that work), and choose the second parent after randomly choosing a crossover point in the first one, and selecting in the second parent a node that is complementary to the one of the first parent. However, this approach requires that the user manually has splitted the problem into several sub-problems, and has identified what are the positive and negative contributions to the different subproblems for a given node. This is a severe restriction to its generic use, and this approach can hence hardly be applied to other problems than the ones presented.

On the other hand, the generality of LTI has been shown here for the class of Boolean problems, and can be extended to regression problems very easily (on-going work). In particular, the high performances obtained on several very different hard benchmarks (like Maj08, Par09 and Mux11 demonstrate that this generality is not obtained by decreasing the performance. This needs to be confirmed on other domain (e.g., regression problems).

Regarding performances on Boolean problems, the very specific BDD (Binary Decision Diagrams) representation allowed some authors to obtain outstanding results using GP [18] and was first to solve the 20-multiplexer (Mux20); and 10 years later these results were consolidated on parity problems up to size 17 [2]. However, handling BDDs with GP implies to diverge from standard GP, in order to meet the constraints of BDDs during the search [14, 2], thus strictly limiting application domains to Boolean problems. It is hoped that the LTI-based approach is more general, and can be ported to other domains, like regression, as already suggested.

7. DISCUSSION AND FURTHER WORK

Let us finally discuss the main differences between LTI and previous work based on the idea of Semantic Backpropagation, namely RDO [17, 12], and some possible future research directions that naturally arise from this comparison.

The main difference lies in the choice of the target node for replacement in the parent tree: uniformly random for RDO, and biased toward local fitness improvement for LTI that looks for the best possible semantic match between the target node and the replacing tree. On the one hand, such exhaustive search explains that LTI seems much slower than the original RDO, though competitive with BP [6]. On the other hand, it is only possible for rather small libraries (see below). However, the computational costs seem to scale up more smoothly with the problem dimension for LTI than for RDO or BP (see e.g., problem Par09, that none of the other semantic-based methods was reported to solve. Nevertheless, the cost of this exhaustive search will become unbearable when addressing larger problems, and some trade-off between LTI exhaustive search and RDO random choice might be a way to overcome the curse of dimensionality (e.g., some tournament selection of the target node).

The other crucial difference is that the results of LTI have been obtained here by embedding it into a standard Iterated Local Search, i.e., outside any Evolutionary framework. In particular, ILTI evolves a single tree, without even any fitness-based selection, similar to a (1+10)-EA. However, though without any drive toward improving the fitness at the level of the whole tree itself, ILTI can reliably solve to optimality several classical Boolean problems that have been intensively used in the GP community (and beyond), resulting in solutions of reasonable size compared to other GP approaches. Hence it is clear that ILTI somehow achieves a good balance between exploitation and exploration. It would be interesting to discover how, and also to investigate whether this implicit trade-off could or should be more explicitly controlled. In particular, would some selection pressure at the tree level help the global search (i.e., replacing the current (1,1)-EA by some (1+1)-EA and varying the selection pressure)? Similar investigations should also be made at the level of LTI itself, which is the only component which drives a tree towards better fitness. Different node selection mechanisms could be investigated for the choice of a target node for replacement.

Finally, the way ILTI escapes local optima should also be investigated. Indeed, it was empirically demonstrated that even though it is necessary to allow LTI to decrease the global fitness of the tree by accepting some replacement that degrade the local performance (and hence the global fitness of the tree at hand), too much of that behaviour is detrimental on complex problems (though beneficial for easy ones) – see the discussion around Fig. 3 in Section 5.

Several other implementation differences should also be highlighted. First, regarding the library, LTI currently uses a small static library made of full trees of depth 2, whereas the libraries in [12] are either the (static) complete set of full trees up to a given depth (3 for Boolean problems, resulting in libraries from size 2524 for 6-bits problems to 38194 for Mux11), or the dynamic set of all subtrees gathered from the current population (of variable size, however reported to be larger than the static libraries of depth 3). Such large sizes probably explain why the perfect match with the semantics of the target node is found most of the time. On the other hand, it could be also be the case that having too many perfect matches is in fact detrimental in the framework of Iterated Local Search, making the algorithm too greedy. This is yet another possible parameter of the exploitation versus exploration trade-off that should be investigated.

Second, RDO implements a systematic test of the ephemeral constants that is chosen as replacement subtree if it improves over the best match found in the library. Such mechanism certainly decreases the bloat, and increases the diversity of replacing subtrees, and its effect within LTI should be investigated.

Also, LTI and RDO handle the "don't care" and "impos-

sible" cases very differently ... maybe due to the fact that, at the moment, LTI has only been applied to Boolean problems. Indeed, as discussed in Section 2.2, the backpropagation procedure in RDO stops whenever an "impossible" value is encountered, whereas it continues in LTI, using the value that is least prone to impossibility as optimal value. But such strategy will not be possible anymore in the Symbolic Regression context: the extension of LTI to regression problems might not be as easy as to Categorical problems (such as the ones experimented with in [6]), which appears straightforward (on-going work).

Tackling continuous Symbolic Regression problems also raise the issue of generalization: How should we ensure that the learned model behaves well on the unseen fitness cases? Standard Machine Learning approaches will of course be required, i.e., using a training set, a test set, and a validation set. In order to avoid over-fitting the training set, the LTI procedure should not be run until it finds a perfect solution on the current training set, and several different training sets might be needed in turn. Interestingly, solving very large Boolean problems will raise similar issues, as it will rapidly become intractable to use all existing fitness cases together, for obvious memory requirement reasons.

Last but not least, further work should investigate the use of LTI within a standard GP evolutionary algorithm, and not as a standalone iterated procedure. All the differences highlighted above between LTI and RDO might impact the behavior of both RDO and AGX: Each implementation difference should be considered as a possible parameter of a more general procedure, and its sensitivity should be checked. Along the same line, LTI could also be used within the initialization procedure of any GP algorithm. However, again, a careful tuning of LTI will then be required, as it should probably not be applied at full strength. Finally, a reverse hybridization between LTI and GP should also be tested: when no improvement can be found in the library during a LTI iteration, a GP run could be launched with the goal of finding such an improving subtree, thus dynamically extending the library. However, beside the huge CPU cost this could induce, it is not clear that decreases of the local fitness are not the only way toward global successes, as discussed above.

Overall, we are convinced that there are many more potential uses of Semantic Backpropagation, and we hope to have contributed to opening some useful research directions with the present work[4].

8. REFERENCES

[1] A. Bhardwaj and A. Tiwari. A Novel GP Based Classifier Design Using a New Constructive Crossover Operator with a Local Search Technique. In D. Huang et al., editor, *Intelligent Computing Theories*, volume 7995 of *LNCS*, pages 86–95. Springer Verlag, 2013.

[2] R. M. Downing. Evolving Binary Decision Diagrams using Implicit Neutrality. In *Proc. IEEE CEC*, pages 2107–2113 Vol. 3, 2005.

[3] B. E. Eskridge and D. F. Hougen. Memetic Crossover for Genetic Programming: Evolution Through Imitation. In K. Deb, editor, *Proc. ACM-GECCO*, pages 459–470. LNCS 3103, Springer Verlag, 2004.

[4] D. E. Goldberg. Zen and the Art of Genetic Algorithms. In J. D. Schaffer, editor, *Proc. 3rd ICGA*, pages 80–85, 1989.

[5] John R Koza. *Genetic Programming: on the Programming of Computers by means of Natural Selection*, volume 1. MIT press, 1992.

[6] K. Krawiec and U.-M. O'Reilly. Behavioral Programming: A Broader and More Detailed Take on Semantic GP. In Dirk Arnold et al., editor, *Proc. GECCO*, pages 935–942. ACM Press, 2014.

[7] K. Krawiec and T. Pawlak. Approximating Geometric Crossover by Semantic Backpropagation. In Ch. Blum and E. Alba, editors, *Proc. 15th GECCO*, pages 941–948. ACM, 2013.

[8] N. F. McPhee, B. Ohs, and T. Hutchison. Semantic Building Blocks in Genetic Programming. In M. O'Neill et al., editor, *Proc. 11th EuroGP*, volume 4971 of *LNCS*, pages 134–145. Springer Verlag, 2008.

[9] P. Moscato. On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms. *Caltech concurrent computation program, C3P Report*, 826:1989, 1989.

[10] Y. Nagata. New EAX Crossover for Large TSP Instances. In Th. P. Runarsson et al., editor, *Proc. PPSN IX*, pages 372–381. LNCS 4193, Springer Verlag, 2006.

[11] F. Neri, C. Cotta, and P. Moscato, editors. *Handbook of Memetic Algorithms*. Number 379 in Studies in Computational Intelligence. Springer Verlag, 2012.

[12] T. Pawlak, B. Wieloch, and K. Krawiec. Semantic Backpropagation for Designing Search Operators in Genetic Programming. *Evolutionary Computation, IEEE Transactions on*, PP(99):1–1, 2014.

[13] K. Rasheed and H. Hirsh. Informed operators: Speeding up genetic-algorithm-based design optimization using reduced models. In D. Whitley et al., editor, *Proc. ACM-GECCO*, pages 628–635, 2000.

[14] H. Sakanashi, T. Higuchi, H. Iba, and Y. Kakazu. Evolution of Binary Decision Diagrams for Digital Circuit Design using Genetic Programming. In T. Higuchi, M. Iwata, and W. Liu, editors, *Evolvable Systems: From Biology to Hardware*, pages 470–481. LNCS 1259, Springer Verlag, 1997.

[15] L. Vanneschi, M. Castelli, and S. Silva. A Survey of Semantic Methods in GP. *Genetic Programming and Evolvable Machines*, 15(2):195–214, 2014.

[16] P. Wang, K. Tang, E.P.K. Tsang, and X. Yao. A Memetic Genetic Programming with Decision Tree-based Local Search for Classification Problems. In *Proc. IEEE-CEC*, pages 917–924, June 2011.

[17] B. Wieloch and K. Krawiec. Running Programs Backwards: Instruction Inversion for Effective Search in Semantic Spaces. In Ch. Blum and E. Alba, editors, *Proc. 15th GECCO*, pages 1013–1020. ACM, 2013.

[18] M. Yanagiya. Efficient Genetic Programming based on Binary Decision Diagrams. In *Proc. IEEE-CEC*, volume 1, pages 234–239, 1995.

[4]More recent results in the same research direction, including the handling of categorical context, will be presented during the SMGP workshop at the same GECCO 2015 conference (see Companion proceedings).