Greedy Semantic Local Search for Small Solutions

Robyn Ffrancon, Marc Schoenauer

▶ To cite this version:

Robyn Ffrancon, Marc Schoenauer. Greedy Semantic Local Search for Small Solutions. Companion Proceedings (workshops) of the Genetic and Evolutionary Computation COnference, Jul 2015, TAO, INRIA Saclay, France. pp.1293-1300. hal-01169075

HAL Id: hal-01169075

https://inria.hal.science/hal-01169075v1

Submitted on 30 Jun 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Greedy Semantic Local Search for Small Solutions

To appear in the Semantic Methods in Genetic Programming Workshop at 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, and to choose a subtree that matches as well as possible these target values. A single tree is evolved by iteratively replacing one of its nodes with the best subtree from a static library according to this local fitness, with tree size as a secondary criterion. Previous results for standard Boolean GP benchmarks that have been obtained by the authors with another variant of SB are improved in term of tree size. SB is then applied for the first time to categorical GP benchmarks, and outperforms all known results to date for three variable finite algebras.

1. INTRODUCTION

Local search algorithms are generally the most straightforward optimization methods that can be designed on any search space that has some neighborhood structure. Given a starting point (usually initialized using some randomized procedure), the search proceeds by selecting the next point, from the neighborhood of the current point, which improves the value of the objective function, with several possible variants (e.g., first improvement, best improvement, etc). When the selection is deterministic, the resulting *Hill Climbing* algorithms generally perform poorly, and rapidly become intractable on large search spaces. Stochasticity must be added, either to escape local minima (e.g. through restart procedures from different random initializations, or by sometimes allowing the selection of points with worse objective value than the current point), or to tackle very large search spaces (e.g., by considering only a small part of the neighborhood of the current point). The resulting algorithms, so-called *Stochastic Local Search* algorithms (SLS) [2], are today the state-of-the-art methods in many domains of optimization.

.

The concept of a neighborhood can be equivalently considered from the point of view of some *move* operators in the search space: the neighborhood of a point is the set of points which can be reached by application of that *move* operator. This perspective encourages the use of stochasticity in a more flexible way by randomizing the *move* operator, thus dimming the boundary between local and global search. It also allows the programmer to introduce domain specific knowledge in the operator design.

All $(1 \dagger \lambda)$-EAs can be viewed as Local Search Algorithms, as the mutation operator acts exactly like the *move* operator mentioned above. The benefit of EAs in general is the concept of population, which permits the transfer of more information from one iteration to the next. However in most domains, due to their simplicity, SLS algorithms have been introduced and used long before more sophisticated metaheuristics like Evolutionary Algorithms (EAs). But this is not the case in the domain of Program Synthesis [1] where Genetic Programming (GP) was the first algorithm related to Stochastic Search which took off and gave meaningful results [3]. The main reason for that is probably the fact that performing random moves on a tree structure rarely result in improvement of the objective value (aka fitness, in EA/GP terminology).

Things have begun to change with the introduction of domain-specific approaches to GP, under the generic name of *Semantic GP*. For a given set of problem variable values, 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 its siblings), as first proposed and described in detail in [6] (see also [7] for the practical design of semantic geometric operators, and [11] 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. In particular, *Semantic Backpropagation* (SB) [12, 5, 8] were the first works to take into account not only the semantic of a subtree to measure its potential usefulness, but also the semantics of the target node where it might be planted. The idea of SB was pushed further in a paper published by the authors in this very conference [1], where the

[1] see also [9] for a survey on recent program synthesis techniques from formal methods and inductive logic programming, to GP.

first (to the best of our knowledge) Local Search algorithm, called Iterated Local Tree Improvement (ILTI), was proposed and experimented with on standard Boolean benchmark problems for GP. Its efficiency favorably compared to previous works (including Behavioural Programming GP [4], another successful approach to learn the usefulness of subtrees from their semantics using Machine Learning).

The present work builds on [1] in several ways. Firstly, Semantic Backprogation is extended from Boolean to categorical problems. Second, and maybe more importantly, the algorithm itself is deeply modified and becomes Iterated Greedy Local Tree Improvements (IGLTI): On one hand, the library from which replacing subtrees are selected usually contains all possible depth-k subtrees ($k = 2$ or $k = 3$), hence the greediness. On the other hand, during each step of the algorithm, a strong emphasis is put on trying to minimize the total size of the resulting tree. Indeed, a modern interpretation of the Occam's razor principle states that small solutions should always be preferred to larger ones – the more so in Machine Learning in general, where large solutions tend to learn "by heart" the training set, with poor generalization properties. And this is even more true when trying to find an exact solution to a (Boolean or categorical) problem with GP. For instance in the categorical domain of finite algebras (proposed in [10]), there exists proven exact methods for generating the target terms. However these methods generate solutions with millions of terms that are of little use to mathematicians.

Assuming that the reader will have access to the companion paper [1], the paper is organized as follows: Section 2 recalls the basic idea of Semantic Backpropagation, illustrated in the categorical case here. Section 3 then describes in detail the new algorithm IGLTI. Section 4 introduces the benchmark problems, again concentrating on the categorical ones, and Section 5 presents the experimental results of IGLTI, comparing them with those of the literature as well as those obtained by ILTI [1]. Finally Section 6 concludes the paper, discussing the results and sketching further directions of research.

2. SEMANTIC BACKPROPAGATION

2.1 Hypotheses and notations

The context is that of supervised learning: The problem at hand comprises n fitness cases, were each case i is a pair (x_i, f_i), x_i being a vector of values for the problem variables, and f_i the corresponding desired tree output. For a 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.

2.2 Rationale

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.

In the following, we will be dealing with a *master 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^*.

2.3 Tree Analysis

For each node in T, the GLTI algorithm maintains an *output vector* and an *optimal vector*. The ith component of the output vector is the actual output of the node when the tree is executed on the ith fitness case; the ith 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 [4], 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 simple tree with top node A and child nodes B and C. For a given fitness case, denote by a, b and c their respective returned 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).

If node A represents operator F, then, by definition

$$a = F(b, c) \quad (2)$$

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

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

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

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

$$\hat{b} = F_b^{-1}(\hat{a}, c) \text{ and } \hat{c} = F_c^{-1}(\hat{a}, b) \quad (4)$$

where F_k^{-1} is the pseudo-inverse operator of F which must be used to obtain the optimum $\hat{k}$ of variable k. The definition of the pseudo-inverse operators in the Boolean case is simpler than that in the categorical case. Only the latter will be discussed now – see [1] for the Boolean case.

In the Boolean case, all operators are symmetrical - hence F_b^{-1} and F_c^{-1} are identical. However, in the categorical problems considered here, the (unique) operator is not commutative (i.e., the tables in Fig. 1 are not symmetrical), hence F_b^{-1} and F_c^{-1} are different.

The pseudo-inverse operator is multivalued: for example, from inspecting the finite algebra $A4$ (Fig. 1-left), it is clear to see that if $\hat{a} = 1$ and $b = 0$ then $\hat{c}$ must equal 0 or 2. In which case we write $\hat{c} = (0, 2)$. That is to say, if $c \in \hat{c}$ and $b = 0$ then $a = 1$. For this example, the pseudo-inverse operator is written as $F_c^{-1}(1, 0) = (0, 2)$. On the other hand, from Fig. 1-right, it comes that $F_b^{-1}(1, 0) = 0$.

Now, consider a second example where the inverse operator is ill-defined. Suppose $\hat{a} = 1$, $b = 1$, and we wish to obtain the value of $\hat{c} = F_c^{-1}(1, 1)$. From inspecting row $b = 1$ of $A4$ we can see that it is impossible to obtain $\hat{a} = 1$ regardless of the value of c. Further inspection reveals that $\hat{a} = 1$ when $b = 0$ and $c = (0, 2)$, or when $b = 2$ and $c = 1$.

Therefore, in order to chose $\hat{c}$ for $\hat{a} = 1$ and $b = 1$, we must assume that $b = 0$ or that $b = 2$. If we assume that $b = 2$ we then have $\hat{c} = 1$. Similarly, if we assume that $b = 0$ we will have $\hat{c} = (0, 2)$. The latter assumption is preferable because we assume that it is less likely for c to satisfy $\hat{c} = 1$ than $\hat{c} = (0, 2)$. In the latter case, c must be one of two different values (namely $c = 0$ or $c = 2$) where as in the former case there is only one value which satisfies $\hat{c}$ (namely $c = 1$). We therefore choose $F_c^{-1}(1, 1) = (0, 2)$. However, as a result, we must also have $F_b^{-1}(1, 0) = 0$ and $F_b^{-1}(1, 2) = 0$.

Of course, for the sake of propagation, the pseudo-inverse operator should also be defined when $\hat{a}$ is a tuple of values. For example, consider the case when $\hat{a} = (1, 2)$, $c = 0$, and $\hat{b}$ is unknown. Inspecting column $c = 0$ in $A4$ will reveal that the only a value that will satisfy $\hat{a}$ (namely $a = 1$ satisfies $\hat{a} = (1, 2)$) is found at row $b = 1$. Therefore, in this case $\hat{b} = F_b^{-1}((1, 2), 0) = 1$.

Using the methodologies outlined by these examples it is possible to derive pseudo-inverse function tables for all finite algebras considered in this paper. As an example, Fig. 2 gives the complete pseudo-inverse table for finite algebra $A4$.

Having defined the pseudo-inverse operators, we can compute, for each fitness case, 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.

2.4 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. A straightforward extension in the categorical context (there is no intrinsic distance between different values) is the following.

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

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

were

$$\ell(a_i, \hat{a_i}) = \begin{cases} 0, & \text{if } a_i \in \hat{a_i} \\ 1, & \text{otherwise.} \end{cases} \quad (6)$$

2.5 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 replacement on the global fitness is, in general, beyond the scope of 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

$A4$		c = 0	c = 1	c = 2
b	0	1	0	1
	1	0	2	0
	2	0	1	0

$B1$		c = 0	c = 1	c = 2	c = 3
b	0	1	3	1	0
	1	3	2	0	1
	2	0	1	3	1
	3	1	0	2	0

Figure 1: Function tables for the primary algebra operators $A4$ and $B1$.

$\hat{a}$	b	$\hat{c}$
0	0	1
	1	(0,2)
	2	(0,2)
1	0	(0,2)
	1	(0,2)
	2	1
2	0	1
	1	1
	2	1
(0,1)	0	(0,1,2)
	1	(0,2)
	2	(0,1,2)
(0,2)	0	1
	1	(0,1,2)
	2	(0,2)
(1,2)	0	(0,2)
	1	1
	2	1
(0,1,2)	0	(0,1,2)
	1	(0,1,2)
	2	(0,1,2)

$\hat{a}$	c	$\hat{b}$
0	0	(1,2)
	1	0
	2	(1,2)
1	0	0
	1	2
	2	0
2	0	1
	1	1
	2	1
(0,1)	0	(0,1,2)
	1	(0,2)
	2	(0,1,2)
(0,2)	0	(1,2)
	1	(0,1)
	2	(1,2)
(1,2)	0	0
	1	(1,2)
	2	0
(0,1,2)	0	(0,1,2)
	1	(0,1,2)
	2	(0,1,2)

Figure 2: Pseudo-inverse operator function tables for the $A4$ categorical benchmark.

$$Err(B, A) = \sum_i \ell(b_i, \hat{a_i}) \quad (7)$$

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

$$best(A) = \{B \in \mathcal{L}; Err(B, A) = min_{C \in \mathcal{L}}(Err(C, A)\} \quad (8)$$

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

$$I(A) = Err(A) - Err(B, A) \text{ for some } B \in best(A) \quad (9)$$

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 erroneous fitness cases may differ, which could adversely affect the whole tree. 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.

Two different IGLTI schemes were tested on the categorical benchmarks which we will refer to as: IGLTI depth 2 and IGLTI depth 3. In the IGLTI depth 2 scheme the library consisted of all semantically unique trees from depth 0 to depth 2 inclusive. Similarly, in the IGLTI depth 3 scheme all semantically unique trees from depth 0 to depth 3 were included. Only the IGLTI depth 3 scheme was tested on the Boolean benchmarks. In this case, the library size was limited to a maximum of 40000 trees.

The library for the ILTI algorithm was constructed from all possible semantically unique subtrees of 2500 randomly generated full trees of depth 2. In this case the library had a strict upper size limit of 450 trees and the library generating procedure immediately finished when this limit was met. Note that for the categorical benchmarks, the size of the library was always below 450 trees. For the Boolean benchmarks on the other hand, the library size was always 450 trees.

In the process of generating the library (whatever design procedure is used), if two candidate subtrees have exactly the same outputs, only the tree with fewer nodes is kept. In this way, the most concise generating tree is stored for each output vector. The library $\mathcal{L}$ is ordered by tree size, from smallest to largest, hence so is $best(A)$. Table 1 gives library sizes for each categorical benchmarks.

Table 1: Library sizes for each categorical benchmark.

Benchmark	Library size		
	IGLTI depth 3	IGLTI depth 2	ILTI depth 2
D.A1	16945	138	72
D.A2	19369	144	78
D.A3	18032	145	81
D.A4	14963	133	69
D.A5	20591	145	81
M.A1	12476	134	68
M.A2	16244	144	78
M.A3	10387	145	81
M.A4	11424	130	66
M.A5	19766	145	81
M.B	21549	-	81

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

Require: $Err(A)$ (Eq. 5), $Err(B, A)$ (Eq. 7), $A \in T$, $B \in \mathcal{L}$

1: $\mathcal{T} \leftarrow \{A \in T;$ **if** $Err(A) \neq 0\}$

2: $bestErr \leftarrow +\infty$
3: $bestReduce \leftarrow +\infty$
4: $bestANodes \leftarrow \{\}$

5: **for** $A \in \mathcal{T}$ **do** ▷ Loop over nodes which could be improved
6: $A.minErr \leftarrow +\infty$
7: $A.minReduce \leftarrow +\infty$
8: $A.libraryTrees \leftarrow \{\}$
9: $indexA \leftarrow$ index position of A in tree T

10: **for** $B \in \mathcal{L}$ **do** ▷ Loop over trees in library
11: **if** $B \in T.bannedBTrees(indexA)$ **then**
12: **continue**

13: $BReduce \leftarrow$ size(B) − size(A)

14: **if** $Err(B, A) < A.minErr$ **then**
15: $A.minErr \leftarrow Err(B, A)$
16: $A.minReduce \leftarrow BReduce$
17: $A.libraryTrees \leftarrow \{B\}$

18: **if** $Err(B, A) = 0$ **then**
19: **break** ▷ Stop library search for current A

20: **else if** $Err(B, A) = A.minErr$ **then**
21: **if** $BReduce < A.minReduce$ **then**
22: $A.minReduce \leftarrow BReduce$
23: $A.libraryTrees \leftarrow \{B\}$

24: **else if** $BReduce = A.minReduce$ **then**
25: $A.libraryTrees$.append(B)

26: **if** $A.minErr < bestErr$ **then**
27: $bestErr \leftarrow A.minErr$
28: $bestReduce \leftarrow A.minReduce$
29: $bestANodes \leftarrow \{A\}$

30: **else if** $A.minErr = bestErr$ **then**
31: **if** $A.minReduce < bestReduce$ **then**
32: $bestReduce \leftarrow A.minReduce$
33: $bestANodes \leftarrow \{A\}$

34: **else if** $A.minReduce = bestReduce$ **then**
35: $bestANodes.append(A)$

36: $chosenA \leftarrow random(bestANodes)$
37: $chosenB \leftarrow random(chosenA.libraryTrees)$

38: $indexA \leftarrow$ index position of $chosenA$ in T
39: $T.bannedBTrees(indexA).append(chosenB)$

40: **return** $chosenA$, $chosenB$, T

3. TREE IMPROVEMENT PROCEDURES

3.1 Greedy Local Tree Improvement

Everything is now in place to describe the full GLTI algorithm, its pseudo-code can be found in algorithm 1. The algorithm starts (line 1) by storing all nodes $A \in T$ where $Err(A) \neq 0$ in the set $\mathcal{T}$. Then, the nodes in $\mathcal{T}$ are each examined individually (line 5).

The library $\mathcal{L}$ is inspected (lines 14 - 25) for each node $A \in \mathcal{T}$ with the aim of recording each associated library tree B which firstly minimises $Err(B, A)$ and secondly minimises $BReduce = \text{size}(B) - \text{size}(A)$. In the worst case, for each node A, every tree B within the library $\mathcal{L}$ is inspected. However, the worst case is avoided, and the inspection of the library is aborted, if a tree $B \in \mathcal{L}$ is found which satisfies $Err(B, A) = 0$.

The master tree T can effectively be seen as an array where each element corresponds to a node in the tree. When a library tree B replaces a node and rooted subtree in T a record is kept of the index position at which B was inserted. For a node A in the master tree, at line 11 the algorithm ensures that the library trees which have previously been inserted at the T array index position of node A are not considered for insertion again at that index position. This ensures that the algorithm does not become stuck in repeatedly inserting the same B trees to the same array index positions of the master tree T.

After inspecting the library for a given node A, the values $A.minErr$ and $A.minReduce$ are used to determine the set of the very best $A \in \mathcal{T}$ nodes, $bestANodes \subseteq \mathcal{T}$ (lines 26 - 35).

Next, the algorithm (line 36) randomly chooses a node $chosenA \in bestANodes$ and randomly chooses an associated tree from its best library tree set
$chosenB \in chosenA.libraryTrees$.

Finally, the algorithm records the chosen library tree $chosenB$ as having been inserted at the array index position of $chosenA$ in T.

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 GLTI is dominated, in the worst case, by the comparisons of the optimal vectors of all m nodes in T with the output vectors of all trees in $\mathcal{L}$, with complexity $n \times m \times o$.

3.2 Iterated GLTI

In the previous section, we have defined the GLTI procedure that, given a master tree T and a library of subtrees $\mathcal{L}$, selects a node $chosenA$ in T and a subtree $chosenB$ in $\mathcal{L}$ to insert in lieu of node $chosenA$ so as to minimize some local error over a sequence of fitness cases as primary criterion, and the full tree size as secondary criterion. In this section we will turn GLTI into a full Stochastic Search Algorithm.

As discussed in [1], or as done in [8], GLTI could be used within some GP algorithm to improve it with some local search, "Ãă la" memetic. However, following the road paved in [1], we are mainly interested here in experimenting with GLTI a full search algorithm that repeatedly applies GLTI to the same tree. Note that the same tree and the same library will be used over and over, so the meaning of "iterated" here does not involve random restarts. On the other hand, the only pressure toward improving the global fitness will be put on the local fitness defined by Eq. 9. In particular, there are cases where none of the library trees can improve the local error: the smallest decrease is nevertheless chosen, hopefully helping to escape some local optimum.

The parameters of IGLTI are the choice of the initial tree, the method (and its parameters) used to create the library, and the size of the library. The end of the paper is devoted to some experimental validation of IGLTI and the study of the sensitivity of the results w.r.t. its most important parameter, the depth of the library trees.

3.3 Modified ILTI

The ILTI scheme (first introduced in [1]) was modified for use in this paper. In the IGLTI algorithm, a record is kept of which library trees were inserted at each array index positions of the master tree. This feature ensured that the same library tree was not inserted at the same array index positions of the master tree more than once. A typical situation where this proved necessary is when a single-node subtree achieves very small number of errors when put at the root position. Without the modification, this single-node tree is inserted at the root every second iteration. Similar situations can also take place at other positions, resulting in endless loops. This modification was particularly useful for problem D.A4[3]. This feature was also implemented in the modified ILTI scheme. Note that for the rest of this paper the modified ILTI scheme will simply be referred to as the ILTI scheme.

4. EXPERIMENTAL CONDITIONS

The benchmark problems used for these experiments are classical Boolean problems and some of the finite algebra categorical problems which have been proposed in [10] and recently studied in [4, 8]. For the sake of completeness, we reiterate their definitions as stated in [4].

"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.

The categorical problems deal with evolving algebraic terms and dwell in the ternary (or quaternary) domain: the admissible values of program inputs and outputs are $\{0, 1, 2\}$ (or $\{0, 1, 2, 3\}$. The peculiarity of these problems consists of using only one binary instruction in the programming language, which defines the underlying algebra. For instance, for the A4 and B1 algebras, the semantics of that instruction are given in Figure 1.

For each of the five algebras considered here, we consider two tasks. In the discriminator term tasks, the goal is to synthesize an expression (using only the one given instruction) that accepts three inputs x, y, z and returns x if $x \neq y$ and z if $x = y$. In ternary domain, this gives rise to $3^3 = 27$ fitness cases.

The second task defined for each of algebras consists in

[3]Notice that Behavioral Programming [4] performed rather poorly on problem D.A4 with a success rate of only 23%.

evolving a so-called *MalâĂŹcev* term, i.e., a ternary term that satisfies $m(x, x, y) = m(y, x, x) = y$. Hence there are only 15 fitness cases for ternary algebras, as the desired value for $m(x, y, z)$, where x, y, and z are all distinct, is not determined."

In the ILTI algorithm a master tree is initialised as a random full tree of depth 2. For the IGLTI algorithm, the initial master tree is chosen as the best performing subtree from the subtree library. If there are multiple library trees with the same performance, the smallest tree is chosen.

Hard run time limits of 5000 seconds were set for each experiment. A run was considered a failure if a solution was not found within this time.

All results were obtained using an 64bits Intel(R) Xeon(R) CPU X5650 @ 2.67GHz. All of the code was written in Python[4].

5. EXPERIMENTAL RESULTS

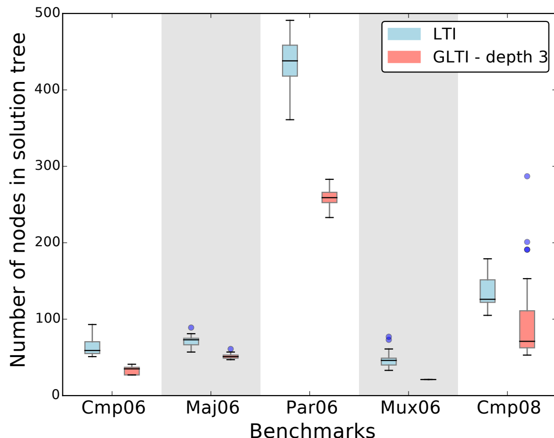

Figure 3: Standard box-plots for the program solution tree sizes (number of nodes) for the ILTI algorithm and IGLTI depth 3 algorithm tested on the Boolean benchmarks. Each algorithm performed 20 runs for each benchmark. Perfect solutions were found from each run except for the Cmp06 benchmark where the IGLTI algorithm failed 4 times (as indicated by the red number four).

Figure 3 shows standard box-plots for solution tree sizes obtained while testing the ILTI and IGLTI depth 3 algorithms on the 6 bit and Cmp08 Boolean benchmarks. It shows how the IGLTI algorithm finds solution trees which are smaller (number of nodes) than those found by the ILTI algorithm. Four failed runs are reported in this figure which occurred when the IGLTI depth 3 algorithm was tested on the Cmp08 benchmark.

The figure also shows how the spread of solution sizes are generally narrower for IGLTI depth 3 than for ILTI. The only exception to this generality is the results of the Cmp08 benchmark. Additional supporting data for this figure is given in Table 3. From inspecting the figure and table together, it is clear that the 20 solution trees obtained from testing IGLTI depth 3 on the Mux06 benchmark were all of the same size.

Figure 4 shows standard box-plots for the number of operators used in the categorical benchmark solutions which were found using the ILTI, IGLTI depth 3, and IGLTI depth 2 schemes. Supporting data for this figure can also be seen in Table 3. However, note that this table measures tree sizes by the number of nodes and not by the number of operators.

The figure shows how the IGLTI depth 3 scheme found the smallest solutions on the $D.A2$, $D.A4$, $D.A5$, $M.A1$, and $M.A2$ benchmarks. For all other three variable categorical benchmarks, the IGLTI depth 2 scheme found the smallest solutions. In all cases, the spread of solution sizes (number of operators) were smallest for IGLTI depth 3 and largest for the ILTI scheme. Reminiscent of the Mux06 benchmark results, the IGLTI depth 3 scheme found twenty solutions which were all of the same size when tested on the $M.A3$ benchmark.

Table 2 gives the algorithm runtimes for each benchmark. The ILTI algorithm is the best performing algorithm for this measure. However, note that the IGLTI depth 2 scheme showed similar average runtimes (but larger spreads) for the three variable *MalâĂŹcev* term benchmarks.

Nine runs of the ILTI algorithm failed to find a solution within the 5000 second time limit when testing on the $M.B$ benchmark. An average of 387.2 ± 283.0 operators were used per correct solution. The IGLTI depth 3 scheme failed to find a solution once when testing on the $M.B$ benchmark. An average of 88.4 ± 21.4 operators were used by the correct solutions found in this case.

6. DISCUSSION AND FURTHER WORK

The results presented in this paper show that SB can be successfully used to solve standard categorical benchmarks when the pseudo-inverse functions are carefully defined. Furthermore, the IGLTI algorithm can be used to find solutions for the three variable categorical benchmarks, which are small enough to be handled by a human mathematician (approximately 45 operators), faster than any other known method.

Interestingly, the results suggest that using a larger library can sometimes lead to worse results (compare the IGLTI depth 2 and IGLTI depth 3 algorithms on the $D.A3$ benchmark for instance). This is likely as a result of the very greedy nature of the IGLTI algorithm. A larger library probably provided immediately better improvements which lead the algorithm away from the very best solutions.

Future work should entail making modification to the IGLTI algorithm so that it is less greedy. In principle, these modifications should be easy to implement by simply adding a greater degree of stochasticity so that slightly worst intermediate results can be accepted. Furthermore, the pseudo-inverse functions should be tested as part of schemes similar to those which feature in [8] with dynamic libraries and a population of potential solutions.

7. REFERENCES

[1] Robyn Ffrancon and Marc Schoenauer. Memetic semantic genetic programming. In S. Silva and A. Esparcia, editors, *Proc. GECCO*. ACM, 2015. To appear.

[4]The entire code base is freely available at robynffrancon.com/GLTI.html

Table 2: Run time (seconds) results for the ILTI algorithm and IGLTI algorithm (library tree maximum depths 2 and 3) tested on the 6bits Boolean benchmarks and the categorical benchmarks. 20 runs were conducted for each benchmark, always finding a perfect solution. The best average results from each row are in bold. The BP4A column is the results of the best performing algorithm of [4] (* indicates that not all runs found a perfect solution).

	Run time [seconds]									
	GLTI - library depth 3			GLTI - library depth 2			modified LTI			BP4A
	max	**mean**	min	max	**mean**	min	max	**mean**	min	mean
D.A1	325.6	298.1 ± 15.2	272.3	24.3	9.8 ± 6.8	2.3	5.9	**2.6 ± 1.4**	1.1	136*
D.A2	366.5	315.5 ± 18.9	289.6	12.3	4.7 ± 2.2	2.0	2.1	**1.3 ± 0.3**	0.8	95*
D.A3	357.1	302.2 ± 23.0	276.7	11.6	4.0 ± 3.0	1.0	2.2	**1.2 ± 0.4**	0.7	36*
D.A4	373.9	308.0 ± 24.4	268.9	320.8	53.5 ± 69.6	4.0	17.6	**5.3 ± 3.3**	2.7	180*
D.A5	421.4	349.2 ± 39.7	282.6	45.8	23.8 ± 9.0	11.9	6.1	**3.1 ± 1.6**	0.9	96*
M.A1	213.2	191.3 ± 15.7	162.5	3.4	1.1 ± 0.6	0.4	1.6	**1.0 ± 0.3**	0.5	41*
M.A2	252.4	241.2 ± 8.6	230.8	2.0	1.0 ± 0.4	0.5	1.1	**0.8 ± 0.2**	0.4	21*
M.A3	172.0	161.7 ± 7.4	148.0	1.7	**0.8 ± 0.3**	0.4	1.1	0.9 ± 0.2	0.5	27*
M.A4	182.3	171.1 ± 5.7	160.7	5.4	3.2 ± 1.3	1.3	1.8	**1.0 ± 0.3**	0.5	9
M.A5	327.5	298.1 ± 20.8	263.9	4.7	1.7 ± 1.1	0.4	1.4	**0.9 ± 0.2**	0.5	14
M.B	6749*	2772.9* ± 1943.0	432*	-	-	-	1815*	**843.6* ± 876.2**	4*	-
Cmp06	165.9	111.3 ± 23.3	61.4	-	-	-	5.3	**4.1 ± 0.6**	2.9	15
Maj06	129.6	95.7 ± 13.0	70.7	-	-	-	5.1	**4.1 ± 0.5**	2.9	36
Par06	396.3	258.2 ± 53.2	164.7	-	-	-	20.1	**13.2 ± 2.5**	7.9	233
Mux06	73.7	66.1 ± 6.4	48.8	-	-	-	4.9	**4.1 ± 0.8**	2.6	10

[2] Holger H. Hoos and Thomas StÃijtzle. *Stochastic Local Search*. Morgan Kaufmann, 2005.

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

[4] 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.

[5] 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.

[6] 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.

[7] A. Moraglio, K. Krawiec, and C. G. Johnson. Geometric Semantic Genetic Programming. In Coello, Carlos A. Coello et al., editor, *Parallel Problem Solving from Nature - PPSN XII*, volume 7491 of *LNCS*, pages 21–31. Springer Verlag, 2012.

[8] 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.

[9] Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, editors. *Inductive Programming: A Survey of Program Synthesis Techniques*, volume 5812 of *Lecture Notes in Computer Science*. Springer Verlag, 2010.

[10] Lee Spector, David M Clark, Ian Lindsay, Bradford Barr, and Jon Klein. Genetic Programming for Finite Algebras. In C. Ryan and M. Keijzer, editors, *Proc. 10th GECCO*, pages 1291–1298. ACM, 2008.

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

[12] 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.

Table 3: Solution program size (number of nodes) results for the ILTI algorithm and IGLTI algorithm (library tree maximum depths 2 and 3) tested on the 6bits Boolean benchmarks and the categorical benchmarks. 20 runs were conducted for each benchmark, always finding a perfect solution. The best average results from each row are in bold. The BP4A column is the results of the best performing algorithm of [4] (* indicates that not all runs found a perfect solution).

	Program size [nodes]									
	GLTI - library depth 3			GLTI - library depth 2			modified LTI			BP4A
	max	**mean**	min	max	**mean**	min	max	**mean**	min	mean
D.A1	107	95.3 ± 4.4	91	107	**80.7** ± **14.1**	55	575	260.5 ± 122.0	137	134*
D.A2	87	**65.7** ± **15.9**	41	121	92.0 ± 18.7	43	295	144.5 ± 48.1	81	202*
D.A3	71	65.1 ± 4.4	61	71	**54.7** ± **6.6**	45	241	146.1 ± 46.4	79	152*
D.A4	103	**84.9** ± **10.4**	67	115	92.6 ± 12.4	67	549	320.9 ± 84.8	187	196*
D.A5	87	**64.6** ± **10.8**	47	173	98.0 ± 23.1	57	465	238.0 ± 100.1	89	168*
M.A1	45	**37.8** ± **2.4**	37	71	46.9 ± 7.9	33	191	104.4 ± 41.9	43	142*
M.A2	49	44.8 ± 3.2	33	59	**44.3** ± **7.7**	33	107	70.8 ± 18.2	45	160*
M.A3	49	49.0 ± 0.0	49	41	**34.8** ± **3.2**	31	259	143.1 ± 51.0	75	104*
M.A4	53	**47.9** ± **2.9**	41	71	49.8 ± 10.9	33	211	119.5 ± 35.6	61	115
M.A5	39	37.8 ± 1.8	35	59	**31.7** ± **13.1**	21	123	77.1 ± 26.2	33	74
M.B	243*	**179.4*** ± **42.3**	95*	-	-	-	3409*	1591.4* ± 1078.6	353*	-
Cmp06	41	**32.9** ± **5.2**	27	-	-	-	93	64.1 ± 11.9	51	156
Maj06	61	**51.2** ± **3.3**	47	-	-	-	89	71.4 ± 7.6	57	280
Par06	283	**260.0** ± **12.1**	233	-	-	-	491	436.0 ± 29.3	361	356
Mux06	21	**21.0** ± **0.0**	21	-	-	-	77	46.3 ± 11.8	33	117

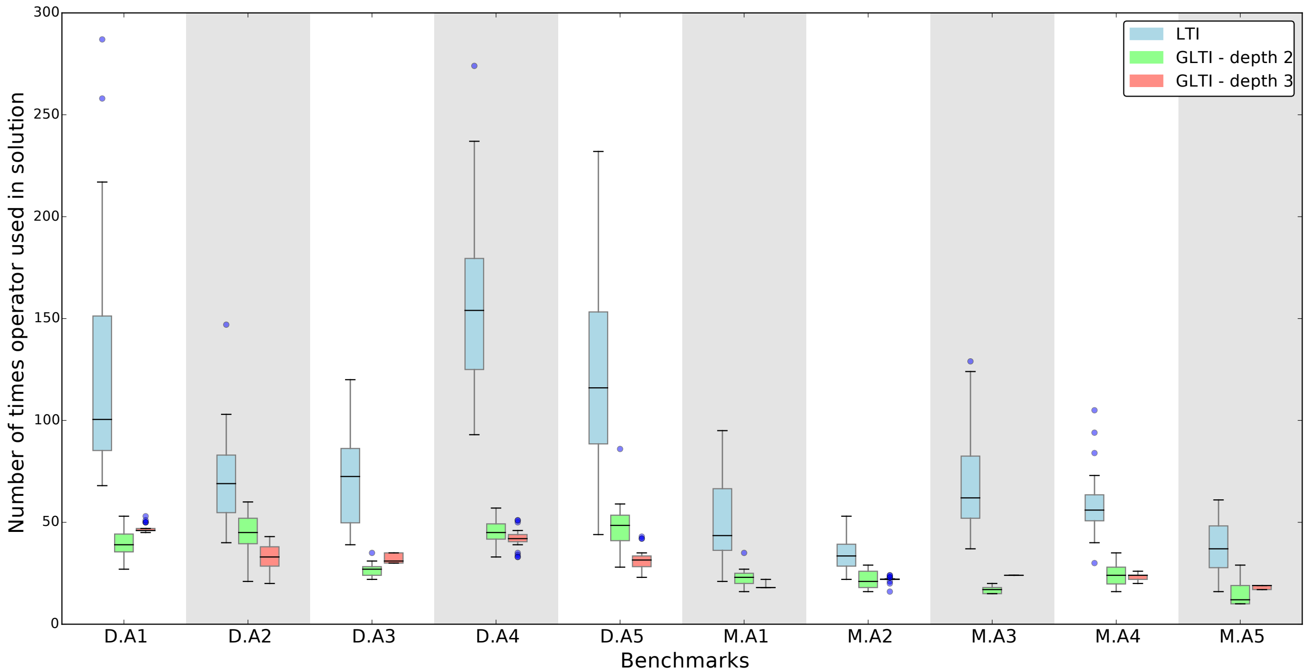

Figure 4: Standard box-plots for the number of operators in program solutions for the ILTI algorithm and IGLTI algorithm (library tree maximum depths 2 and 3) tested on the categorical benchmarks: For each problem, from left to right, ILTI, IGLTI-depth 2, and IGLTI-depth 3. Each algorithm performed 20 runs for each benchmark. Perfect solutions were found from each run.