École Polytechnique

Masters Thesis

Reversing Operators for Semantic Backpropagation

Author:
Robyn Ffrancon

Supervisor:
Marc Schoenauer

A thesis submitted in fulfillment of the requirements for the degree of Masters in

Complex Systems Science

This work was conducted during a 6 month internship at
TAO team, INRIA, Saclay, France

June 2015

ÉCOLE POLYTECHNIQUE

Abstract

Complex Systems Science

Masters

Reversing Operators for Semantic Backpropagation

by Robyn FFRANCON

Boolean function synthesis problems have served as some of the most well studied benchmarks within Genetic Programming (GP). Recently, these problems have been addressed using Semantic Backpropagation (SB) which was introduced in GP so as to take into account the semantics (outputs over all fitness cases) of a GP tree at all intermediate states of the program execution, i.e. at each node of the tree. The mappings chosen for reversing the operators used within a GP tree are crucially important to SB. This thesis describes the work done in designing and testing three novel SB algorithms for solving Boolean and Finite Algebra function synthesis problems. These algorithms generally perform significantly better than other well known algorithms on run times and solution sizes. Furthermore, the third algorithms is deterministic, a property which makes it unique within the domain.

Acknowledgements

I wish to thank my internship supervisor Marc Schoenauer for interesting discussions and excellent supervision during my six month masters internship at INRIA, Saclay, France where this work was conducted.

Contents

List of Figures

List of Tables

Chapter 1

Thesis Overview

1.1 General Introduction

Genetic Programming (GP) is a well researched system which is used to synthesise S-expression tree functions using a method that is analogous to biological evolution [4]. Guided by evolutionary pressure, a population of partial solution functions are bread and mutated to form subsequent generations until a satisfactory solution function is found. GP has been used to tackle a wide range of different problems, such as the design of quantum algorithms [5] and the development of financial investment strategies [6].

This thesis presents three novel algorithms which also tackle function synthesis problems. However, where as GP evolves and manipulates a population of partial solution trees, each of these three algorithms perform iterated local (local to a tree's nodes) improvements on a single function tree until it becomes a satisfactory solution. This change in methodology resulted in significant improvements, as compared to several other successful algorithms, when testing on standard benchmarks within two different problem domains.

All three algorithms have been documented in separate papers which are provided chronologically in the Appendix. The *Iterative Local Tree Improvement* (ILTI) algorithm was documented in [7] (Appendix A). This paper is to appear as a full conference paper at GECCO 2015 [8]. It was nominated for the best paper award within the GP track. Secondly, the *Iterative Greedy Local Tree Improvement* (IGLTI) algorithm was documented in [9] (Appendix B). This paper is to appear in the Semantics Workshop at GECCO 2015 and has also been submitted to EA 2015 [10]. It features the first ever application of semantic backpropagation to finite algebra problems. Thirdly, the *Node-by-Node Growth Solver* (NNGS) algorithm was documented in [11] (Appendix C).

1.1.1 Motivation and Problem Domains

The ILTI and IGLTI algorithms were designed to solve problems from two different domains: Boolean and Finite Algebra. However, the NNGS algorithm was designed to exclusively solve Boolean domain problems.

An obvious example of a Boolean domain problem is the design of electrical circuits [12]. Some heat energy is inevitably lost during the operation of any electrical circuit. This loss can be minimised by ensuring that the circuit is as concise as possible. Therefore, the primary goal within this problem domain is to generate the smallest [1] circuits possible within a reasonable time constraint.

Typically, in a circuit design problem the input and output signals are well-defined and known to the solver. Therefore, the problem consists of finding a Boolean function which maps the input signals to the output signal. A set of Boolean operators is made available to the solver for synthesising the solution function. Usually, but not always, this set is restricted to the AND, OR, $NAND$, and NOR operators. Using these operators it is theoretically possible to synthesise any Boolean function.

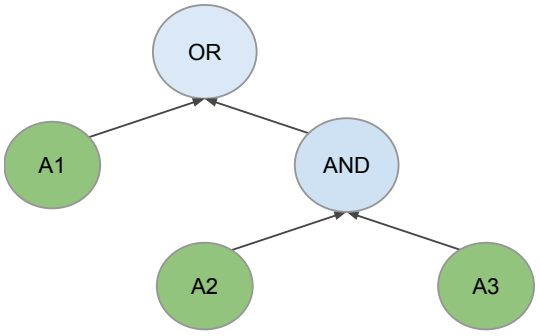

FIGURE 1.1: A simple S-expression tree with three green input argument nodes (labelled $A1$, $A2$, and $A3$), one blue internal AND operator node, and one blue OR operator root node. This example takes three input bits and one outputs bit.

Figure 1.1 shows a three input bits example S-expression tree Boolean function which could have been synthesised using GP or any of the tree algorithms presented in this thesis.

One reason for restricting the set of operators is to ease the task of performance comparison between multiple different algorithms. Another method for simplifying comparisons is to attempt to solve a set of well known benchmark problems [13]. One example benchmark is the *n-bits comparator* [1]. This benchmark takes as inputs n-bits and outputs a single bit. If the $n/2$ least significant input bits encode a number that is smaller than the number represented by the $n/2$ most significant bits the function returns *true* (1); otherwise the function returns *false* (0).

Usefully, this type of benchmark is tunably difficult, that is to say, the benchmark can be made harder to solve by simply increasing the benchmark bit size n. Other standard Boolean benchmarks are the *multiplexer, majority* and *even-parity* benchmarks. Further

[1] Note also that a solution's size is often closely related to the solution's generalizability. That is to say, a smaller solution is probably more general. If a solution generalises well, it would only require testing on a subset of test cases. This is clearly advantageous for problems with many test cases.

justification for the choice of benchmarks used in this thesis is given in [13] and full descriptions of all benchmarks used are given in the Appendices.

For a n-bits Boolean benchmark the synthesised solution function must satisfy 2^n different fitness cases. This is because there are 2^n unique permutations of the input argument values. In the case of three input arguments, two example permutations are $(A1 = 0, A2 = 0, A3 = 1)$ and $(A1 = 0, A2 = 1, A3 = 1)$. Each permutation has an associated single output bit. In fact, each input argument can be thought of as a 2^n element 1D array of Boolean values, where each row within the arrays correspond to one fitness case (one permutation of input argument values).

An operator acts on two input arrays element-by-element (one fitness case at a time). In this way, a node produces an output array by acting on it's child node input arrays. When the output array produced by the root node corresponds to the target output array given by the problem, the tree is a correct solution. Note, the inputs and output arrays are given by the problem definition. Furthermore, internal nodes (such as the AND node in Fig. 1.1) also produce intermediate output arrays. In fact, each node within a function tree has an associated 2^n element output array of Boolean values.

The second problem domain addressed by the ILTI and IGLTI algorithms deals with finite algebra. For our purposes, a finite algebra is a two input argument operator mapping with a single output. They can be considered generalisations of the operators found within the Boolean domain. A variable (e.g. input argument) in the Boolean domain can take the value 0 or 1. The input and output values of a finite algebra operator can be any integer value within the range $[0, m-1]$ where m is the size of the finite algebra.

Standard finite algebra benchmarks were first tackled with GP in [14] . They consist of finding ternary domain algebraic terms such as the *discriminator term* and the *Mal'cev term*. This is motivated by the fact that these terms could prove useful to mathematicians if they were small enough. Note that within this problem domain, as in the Boolean domain, each node generates an output array when a solution tree is executed.

1.1.2 Semantic Backpropagation

Recall that the goal of a solver algorithm is to synthesise a function, using a restricted set of operator(s), which produces a known target output array. In this thesis, it is shown how the output array associated with each node within a partially-correct tree can be used to iteratively synthesise a perfect solution. To this end, *Semantic Backpropagation* (SB) was used to iteratively improve a partially-correct tree in the ILTI and IGLTI

algorithms. SB was also used in the NNGS algorithm to guide the process of growing a single solution tree.

However, before understanding SB, the concept of node *semantics* must be understood. In simple terms, the output array associated with a node is the semantics of that node and its rooted subtree. Note that multiple different subtrees may produce the same output arrays, and therefore have the same semantics. GP systems have recently incorporated node semantics to improve search performance [15]. A good example is given by [1] which introduced the notion of Behavioural Programming. They used machine learning to analyse the internal semantics of S-expression trees so as to identify useful subtrees.

Following on from semantic GP, the notion of SB was developed in [2] amongst other works. The practical SB application methodology is always specific to the problem domain under investigation and the strategy used by the solver algorithm. For instance, ILTI and IGLTI both use SB to effectively assign fitness values (score) to nodes within a single partially-correct solution tree. The NNGS algorithm, on the other hand, uses SB to assign subsequent nodes whilst growing a single solution tree from root to leaves.

For the purposes of this outline it is sufficient to understand the core motivation behind SB. Firstly, recall a key fact: the target output array obtained from the synthesised solution function is given to the solver algorithm from the start. Secondly, note that SB relies on reversing operators. SB can be crudely understood as executing a tree in reverse (by reversing operators) whilst using the target output array as an input.

1.1.3 Algorithm Outline and Comparisons

The ILTI and IGLTI algorithms both iteratively improve a single erroneous solution tree by substituting subtrees with those found in static libraries. They are fundamentally similar in many ways but differ in the amount of analysis performed on the single improved tree before a change to that tree is made. The IGLTI algorithm performs a greater amount of analysis as compared to the ILTI algorithm. Its analysis is geared towards reducing the solution tree size at every possible opportunity whilst not sacrificing on error. As a result, the algorithm successfully finds smaller solutions but performs slower.

The NNGS algorithm, on the other hand, does not iteratively improve an existing (although erroneous) tree. Instead, it grows a solution tree from root node to leaf nodes, node-by-node. Whilst the algorithm grows the solution tree, the tree is incomplete (invalid) until the very last node is grown.

The NNGS algorithm is the least resource intensive algorithm introduced in this thesis. By growing a single solution tree it avoids (unlike the ILTI and IGLTI algorithms) forming and searching a library of static trees. It also avoids re-executing the single improved tree at each iteration. Indeed, the solution tree is not executable until the NNGS algorithm halts, at which point the solution tree is guaranteed to be correct. This means that the generated solution tree does not require execution once. Of course, this is a clear improvement on classical GP where a population of potentially thousands of trees must be evaluated (executed) at each generation.

The ILTI and IGLTI algorithms are both stochastic, a feature which is helpful for escaping local minima within the search space. The NNGS algorithm, on the other hand, is entirely deterministic. This feature makes the algorithm (as far as the author can tell) entirely unique within the problem domain with the exception of exhaustive search algorithms.

1.2 Results Overview

The ILTI algorithm Boolean benchmark results show a 100% success rate. This is a clear improvement on classical GP where 0% success rates are common on many benchmarks (e.g. the 6bits even-parity benchmark) [1]. Furthermore, even the best performing Behavioural Programming algorithm in [1], namely BP4A, routinely achieved less than 100% success rate (in one case as low as 40%). The ILTI algorithm also found smaller solutions than those synthesised by BP4A and the RDO operator as reported in [2].

In addition, ILTI finds solutions much faster than BP4A and other classical GP systems. For example, the 8bits even-parity benchmark was completed by BP4A in 3792 seconds on average with a 40% success rate. The same benchmark was completed in 622 seconds with 100% success rate by the ILTI algorithm. ILTI was even able to complete the harder 9bits even-parity benchmark with 100% success rate in 5850 seconds on average.

The IGLTI algorithm found smaller Boolean benchmark and finite algebra benchmarks solutions than the ILTI algorithm. However, the algorithm operated much slower. The NNGS algorithm, on the other hand, produced large solutions very quickly. However, the NNGS algorithm still produced solutions which were smaller than those found by the best performing RDO operator in [2]. Furthermore, the NNGS algorithms had 100% success rates on all Boolean benchmarks tested so far.

The NNGS algorithm completed all benchmarks in 0.8% of the runtime required by the BP4A algorithm on average. This is a very significant improvement. For instance, the 8bits even-parity benchmark that was completed by BP4A in 3792 seconds took

the NNGS algorithm 5.73 seconds. However, the solution size obtained by the NNGS algorithm for this benchmark was much greater than that found by BP4A.

1.3 Discussion and Conclusion

One of the main novelty common to all three algorithms presented in this thesis is their focus on developing a single solution tree. This strategy improved algorithm efficiency immensely as compared to classical GP.

The work done in designing and testing the IGLTI algorithm also included the first ever application of SB to finite algebra problems. The key innovation was to carefully chose the reverse mapping for the finite algebra operator. In fact, carefully defining the reverse mapping of the operators proved to be a critical factor in the operation of all three algorithms.

Future work which stems from this research will constitute carefully analysing why the NNGS algorithm works even though it is deterministic and also how it could be improved to generate smaller solutions.

Appendix A

Memetic Semantic Genetic Programming

A.1 Introduction

Memetic Algorithms [16] have become increasingly popular recently (see e.g., [17]): 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 [17]). 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 [18]).

Historical Genetic Programming (GP) [4] 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 [19] (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 [1]. 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) [2, 3, 20] 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 [2]. 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 [2, 3]. 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 [19]), 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 B.6).

The rest of the paper firstly recalls (Section B.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 [3]. Section B.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 B.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 A.6, while links with previous Semantic Backpropagation works are highlighted and discussed in Section B.6, together with related possible further research paths.

A.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

[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 [3, 20].

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.

A.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

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

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^*.

A.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 [1], 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).

If node A represents operator F, then, by definition

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

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

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

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

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

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 [3], where the backpropagation stops whenever either "don't care" or "impossible" are encountered. See the discussion in Section B.6.

Function tables for the Boolean operators AND^{-1}, OR^{-1}, NAND^{-1}, and NOR^{-1} are given in Fig. B.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.

A.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. B.1) can be reused, provided that it is extended to handle sets of

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

$\hat{c} = \text{AND}^{-1}(\hat{a}, b)$		
$\hat{a}$	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 A.1: Function tables for the AND^{-1}, OR^{-1}, NAND^{-1}, and NOR^{-1}.

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

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

A.2.4 Subtree Library

Given a node A in T that is candidate for replacement (see next Section B.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

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

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

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

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

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

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. B.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 $best(A)$.

A.3 Tree Improvement Procedures

A.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 [21].

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. B.7), 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

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

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

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**

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 damage the current tree (see Fig. A.3 and the discussion in Section B.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 B.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 $n \times m \times o$.

A.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 [17]). 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.

TABLE A.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 [1] (* indicates that not all runs found a perfect solution). The RDO column is taken from [2].

	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

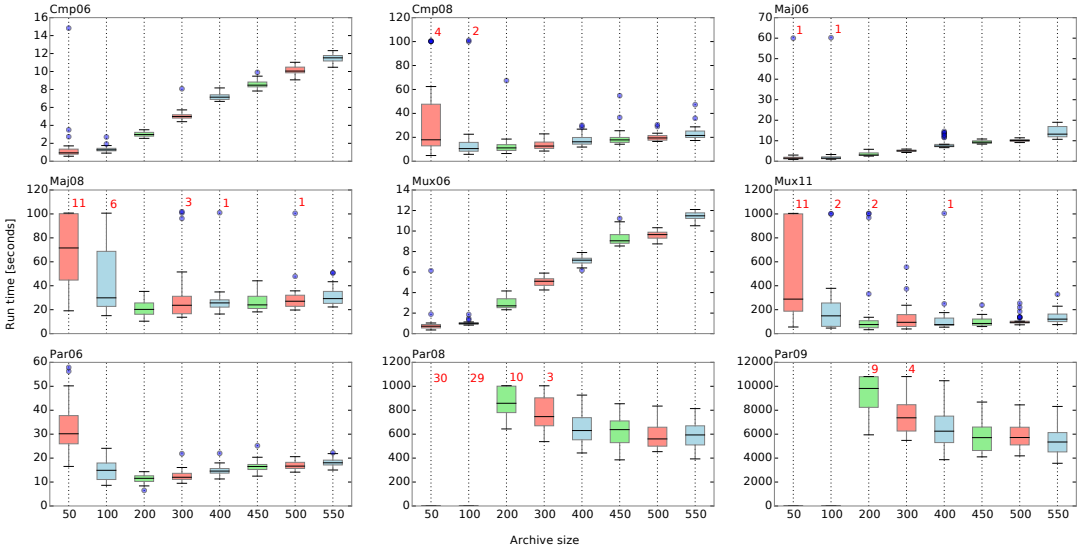

FIGURE A.2: Comparative results: time to solution for different library sizes. The benchmark problems appear in the order of Table A.1: Cmp06, Cmp08, Maj06; Maj08, Mux06, Mux11; Par06, Par08 and Par09.

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 B.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.

A.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 A.6 also). We have chosen this particular benchmarks because they are used in [1, 3] (among other types of benchmarks). For the sake of completeness, we reiterate their definitions as stated in [1]: 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 B.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 [1]). 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

A.5 Experimental results

Figure A.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 A.1, discussed next.

Based on the above, the comparative results of Table A.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* [1], and when possible with the results of RDO given in [2] or in [3].

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 A.1 for BP, and Par06 for RDO (success rate of 0.99 according to [2]).

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

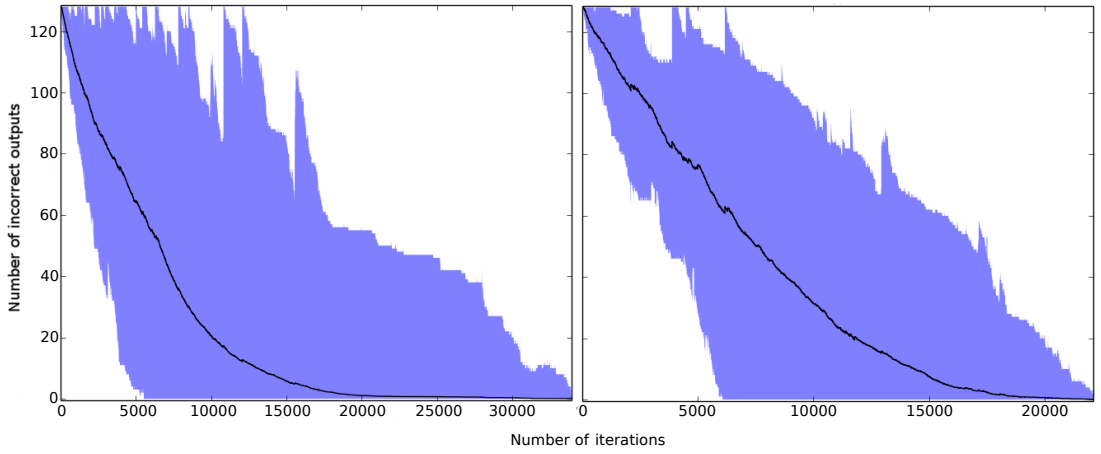

FIGURE A.3: Evolution of the global error during 30 runs of ILTI on Par08 problem: average and range. Left: Node selection by error only; Right: node selection by depth first.

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 runtimes 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 [2] 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 [2] report much larger tree sizes, from twice to 20 times larger than those of ILTI.

Finally, Fig. A.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 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.

A.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 [22]. Though this is indeed some sort of local search (the usual GP variation operators being used as elementary 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 [23]).

Some real memetic search within GP has been proposed in the context of decision trees for classification [24]: 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 [25]: 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 [26] and was first to solve the 20-multiplexer (Mux20); and 10 years later these results were consolidated on parity problems up to size 17 [27]. However, handling BDDs with GP implies to diverge from standard GP, in order to meet the constraints of BDDs during the search [27, 28], 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.

A.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 [2, 3], 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 [1]. 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. A.3 in Section B.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 [3] 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 "impossible" 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 B.2.3, 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 [1]), 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 Back-propagation, and we hope to have contributed to opening some useful research directions with the present work[4].

[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).

Appendix B

Greedy Semantic Local Search for Small Solutions

B.1 Introduction

Local search algorithms are generally the most straightforward optimization methods that can be designed on any search space that has some neighbourhood structure. Given a starting point (usually initialized using some randomized procedure), the search proceeds by selecting the next point, from the neighbourhood 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 neighbourhood of the current point). The resulting algorithms, so-called *Stochastic Local Search* algorithms (SLS) [29], are today the state-of-the-art methods in many domains of optimization.

The concept of a neighbourhood can be equivalently considered from the point of view of some *move* operators in the search space: the neighbourhood 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 $(1^+,\lambda)$-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 [4]. 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 [19] (see also [30] for the practical design of semantic geometric operators, and [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. In particular, *Semantic Backpropagation* (SB) [2, 3, 20] 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 [7], where the 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 [1], another successful approach to learn the usefulness of subtrees from their semantics using Machine Learning).

The present work builds on [7] 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

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

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 [14]), 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 [7], the paper is organized as follows: Section B.2 recalls the basic idea of Semantic Backpropagation, illustrated in the categorical case here. Section B.3 then describes in detail the new algorithm IGLTI. Section B.4 introduces the benchmark problems, again concentrating on the categorical ones, and Section B.5 presents the experimental results of IGLTI, comparing them with those of the literature as well as those obtained by ILTI [7]. Finally Section B.6 concludes the paper, discussing the results and sketching further directions of research.

B.2 Semantic Backpropagation

B.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

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

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.

B.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^*.

B.2.3 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 [1], 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

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

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

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

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

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

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 [7] 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. B.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. B.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. B.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 $c = 1$.

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

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

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

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

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. B.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.

B.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. B.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

$\hat{a}$	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 B.2: Pseudo-inverse operator function tables for the $A4$ categorical benchmark.

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

were

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

B.2.5 Subtree Library

Given a node A in T that is candidate for replacement (see next Section B.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

$$Err(B, A) = \sum_i \ell(b_i, \hat{a}_i) \tag{B.7}$$

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

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

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

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

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 B.1 gives library sizes for each categorical benchmarks.

Algorithm 2 Procedure GLTI(Tree T, library $\mathcal{L}$)

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

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

$bestErr \leftarrow +\infty$
$bestReduce \leftarrow +\infty$
$bestANodes \leftarrow \{\}$

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

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

        $BReduce \leftarrow \text{size}(B) - \text{size}(A)$

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

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

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

            else if $BReduce = A.minReduce$ **then**
                $A.libraryTrees.\text{append}(B)$

    if $A.minErr < bestErr$ **then**
        $bestErr \leftarrow A.minErr$
        $bestReduce \leftarrow A.minReduce$
        $bestANodes \leftarrow \{A\}$

    else if $A.minErr = bestErr$ **then**
        if $A.minReduce < bestReduce$ **then**
            $bestReduce \leftarrow A.minReduce$
            $bestANodes \leftarrow \{A\}$

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

$chosenA \leftarrow random(bestANodes)$
$chosenB \leftarrow random(chosenA.libraryTrees)$

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

TABLE B.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

B.3 Tree Improvement Procedures

B.3.1 Greedy Local Tree Improvement

Everything is now in place to describe the full LTI algorithm, its pseudo-code can be found in algorithm 2. 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 LTI 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 $n \times m \times o$.

B.3.2 Iterated GLTI

In the previous section, we have defined the LTI 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 LTI into a full Stochastic Search Algorithm.

As discussed in [7], or as done in [3], GLTI could be used within some GP algorithm to improve it with some local search, "à la" memetic. However, following the road paved in [7], 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. B.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.

B.3.3 Modified ILTI

The ILTI scheme (first introduced in [7]) 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.

B.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 [14] and recently studied in [1, 3]. For the sake of completeness, we reiterate their definitions as stated in [1].

"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 B.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.

[3]Notice that Behavioural Programming [1] performed rather poorly on problem D.A4 with a success rate of only 23%.

The second task defined for each of algebras consists in evolving a so-called *Mal'cev* 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].

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

B.5 Experimental results

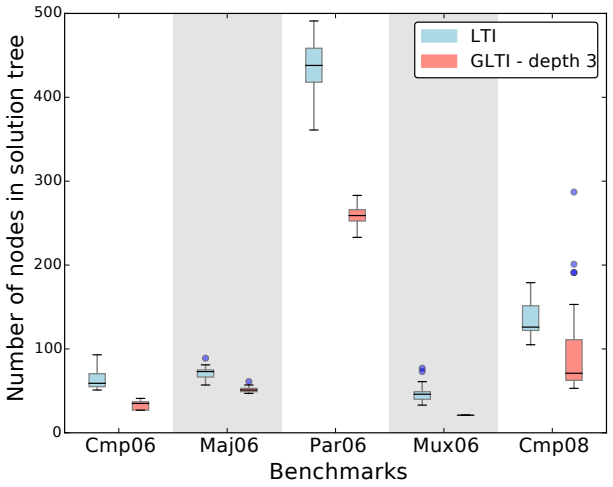

FIGURE B.3: Standard box-plots for the program solution tree sizes (number of nodes) for the ILTI algorithm and IGLTI depth 3 algorithm tested on the Boolean benchmarks. Each algorithm performed 20 runs for each benchmark. Perfect solutions were found from each run except for the Cmp06 benchmark where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

Figure B.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 B.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 B.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 B.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 B.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 [1] (* 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

Table B.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 *Mal'cev* 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.

TABLE B.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 [1] (* 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

B.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 [3] with dynamic libraries and a population of potential solutions.

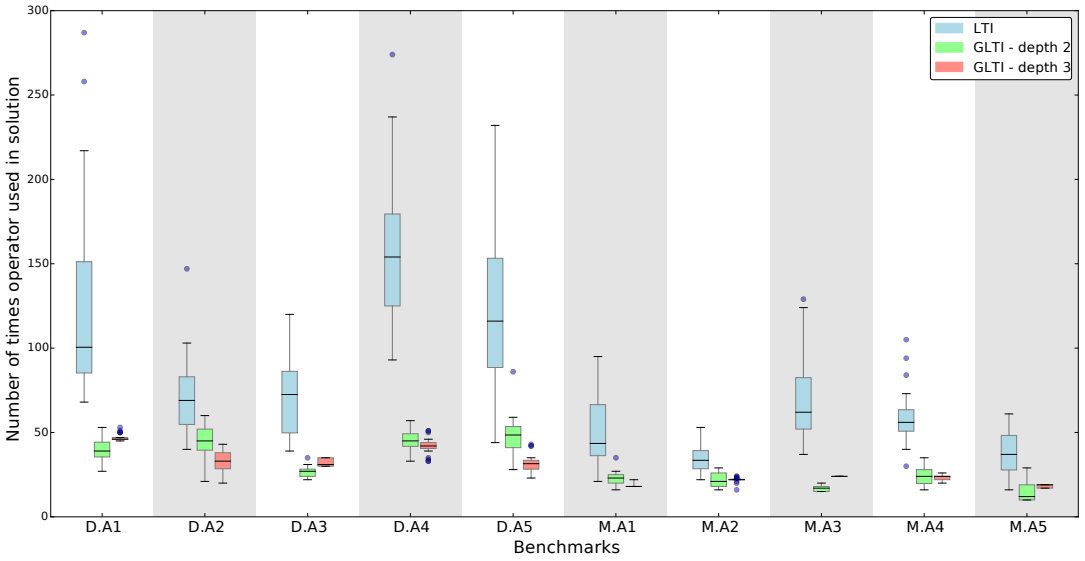

FIGURE B.4: Standard box-plots for the number of operators in program solutions for the ILTI algorithm and IGLTI algorithm (library tree maximum depths 2 and 3) tested on the categorical benchmarks: For each problem, from left to right, ILTI, IGLTI-depth 2, and IGLTI-depth 3. Each algorithm performed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

Appendix C

Retaining Experience and Growing Solutions

C.1 Motivation

Most genetic programming (GP) [4] systems don't adapt or improve from solving one problem to the next. Any experience which could have been gained by the system is usually completely forgotten when that same system is applied to a subsequent problem, the system effectively starts from scratch each time.

For instance, consider the successful application of a classical GP system to a standard n-bits Boolean function synthesis benchmark (such as the 6-bits comparator as described in [1]). The population which produced the solution tree is not useful for solving any other n-bits Boolean benchmark (such as the 6-bits multiplexer). Therefore, in general, an entirely new and different population must be generated and undergo evolution for each different problem. This occurs because the system components which adapt to solve the problem (a population of trees in the case of classical GP) become so specialised that they are not useful for any other problem.

This paper addresses this issue by introducing the *Node-by-Node Growth Solver (NNGS)* algorithm, which features a component called the *controller*, that can be improved from one problem to the next within a limited class of problems.

NNGS uses *Semantic Backpropagation (SB)* and the controller, to grow a single S-expression solution tree starting from the root node. Acting locally at each node, the controller makes explicit use of the target output data and input arguments data to determine the properties (i.e. operator type or argument, and semantics) of the subsequently generated child nodes.

The proof-of-concept controller discussed in this paper constitutes a set of deterministic hand written rules and has been tested, as part of the NNGS algorithm, on several popular Boolean function synthesis benchmarks. This work aims to pave the way towards the use of a neural network as an adaptable controller and/or, separately, towards the use of meta-GP for improving the controller component. In effect, the aim is to exploit the advantages of black-box machine learning techniques to generate small and examinable program solutions.

The rest of this paper will proceed as follows: Section C.2 outlines other related research. Section C.3 details semantic backpropagation. A high level overview of the NNGS system is given in Section C.4, and Section C.5 describes the proof-of-concept controller. Section C.6 details the experiments conducted. The experimental results and a discussion is given in Section C.7. Section C.8 concludes with a description of potential future work.

C.2 Related Work

The isolation of useful subprograms/sub-functions is a related research theme in GP. However, in most studies subprograms are not reused across different problems. In [31] for instance, the hierarchical automatic function definition technique was introduced so as to facilitate the development of useful sub-functions whilst solving a problem. Machine learning was employed in [1] to analyse the internal behaviour (semantics) of GP programs so as to automatically isolate potentially useful problem specific subprograms.

SB was used in [3] to define intermediate subtasks for GP. Two GP search operator were introduced which semantically searched a library of subtrees which could be used to solve the subtasks. Similar work was carried out in [7, 9], however in these cases subtree libraries were smaller and static, and only a single tree was iteratively modified as opposed to a population of trees.

C.3 Semantic Backpropagation (SB)

Semantic backpropagation (SB) within the context of GP is an active research topic [3, 7, 9, 20].

Consider the *output array* produced by the root node of a solution tree, where each element within the array corresponds to the output from one fitness case. This output array is the semantics of the root node. If the solution is perfectly correct, the output

array will correspond exactly to the target output array of the problem at hand. In a programmatic style, the output array of a general node *node_x* will be denoted as *node_x*.outputs and the output from fitness case i will be denoted by *node_x*.outputs[i].

Each node within a tree produces an output array, a feature which has been exploited in [1] to isolate useful subtrees. The simplest example of a tree (beyond a single node) is a triplet of nodes: a parent node *node_a*, the left child node *node_b*, and the right child node *node_c*.

As a two fitness case example, suppose that a triplet is composed of a parent node *node_a* representing the operator AND, a left child node *node_b* representing input argument $A1 = [0, 1]$, and a right child node *node_c* representing input argument $A2 = [1, 1]$. The output array of the parent node is given by:

$$\begin{aligned} \textit{node\_a}.\text{outputs} &= \textit{node\_b}.\text{outputs}\ AND\ \textit{node\_c}.\text{outputs} \\ &= [0, 1]\ AND\ [1, 1] \\ &= [0, 1]. \end{aligned} \tag{C.1}$$

On the other hand, given the output array from the parent node of a triplet *node_a*, it is possible to backpropagate the semantics so as to generate output arrays for the child nodes, if the reverse of the parent operator is defined carefully. This work will exclusively tackle function synthesis problems within the Boolean domain, and therefore, the standard [1, 31] set of Boolean operators will be used: AND, OR, $NAND$, and NOR.

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

$\hat{a}$	$\hat{b}$	$\hat{c}$
1	1	1
0	0	#
0	#	0
#	#	#

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

$\hat{a}$	$\hat{b}$	$\hat{c}$
1	1	#
1	#	1
0	0	0
#	#	#

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

$\hat{a}$	$\hat{b}$	$\hat{c}$
1	0	#
1	#	0
0	1	1
#	#	#

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

$\hat{a}$	$\hat{b}$	$\hat{c}$
1	0	0
0	1	#
0	#	1
#	#	#

FIGURE C.1: Function tables for the reverse operators: AND^{-1}, OR^{-1}, NAND^{-1}, and NOR^{-1}.

Figure C.1 gives function tables for the element-wise reverse operators: AND^{-1}, OR^{-1}, $NAND^{-1}$, and NOR^{-1} (their use with 1D arrays as input arguments follows as expected). As an example use of these operators the previous example will be worked in

reverse: given the output array of *node_a*, the arrays *node_b*.outputs and *node_c*.outputs are calculated as:

$$\begin{aligned} \textit{node\_b}.\text{outputs},\ \textit{node\_c}.\text{outputs} &= AND^{-1}(\textit{node\_a}.\text{outputs}) \\ &= AND^{-1}([0,1]) \\ &= [0,1],\ [\#,1] \\ &\text{or} \\ &= [\#,1],\ [0,1]. \end{aligned} \tag{C.2}$$

The hash symbol # in this case indicates that either 1 or 0 will do. Note that two different possible values for *node_b*.outputs and *node_c*.outputs exist because $AND^{-1}(0) = (0,\#)$ or $(\#,0)$. This feature occurs as a result of rows 4 and 5 of the $NAND^{-1}$ function table. Note that each of the other reverse operators have similar features, specifically for: $OR^{-1}(1)$, $NAND^{-1}(1)$, and $NOR^{-1}(0)$.

Note also, that for any array loci i in *node_a*.outputs where *node_a*.outputs$[i] = \#$, it is true that *node_b*.outputs$[i] = \#$ and *node_c*.outputs$[i] = \#$. For example, $NOR^{-1}([1,\#]) = ([0,\#],[0,\#])$.

Using the reverse operators in this way, output arrays can be assigned to the child nodes of any parent node. The child output arrays will depend on two decisions: Firstly, on the operator assigned to the parent node, as this is the operator that is reversed. And secondly, on the choices made (note the $AND^{-1}(0)$ example above), at each loci, as to which of the two child output arrays contains the # value. These decisions are made by the controller component.

Using these reverse operators for SB can only ever produce a pair of output arrays which are different from the problem target outputs in two ways. Firstly, the output arrays can be a flipped (using the NOT gate on each bit) or an un-flipped versions of the problem target outputs. Secondly, some elements of the output arrays will be # elements.

C.4 Node-by-Node Growth Solver (NNGS)

A visual representation of the NNGS algorithm can be seen in Fig. C.2, which shows a snapshot of a partially generated solution tree. This tree, in it's unfinished state, is composed of: AND and OR operators, an input argument labelled $A1$, and two unprocessed nodes. The basic role of the NNGS algorithm is to manage growing the solution tree by passing unprocessed nodes to the controller and substituting back the generated/returned node triplet.

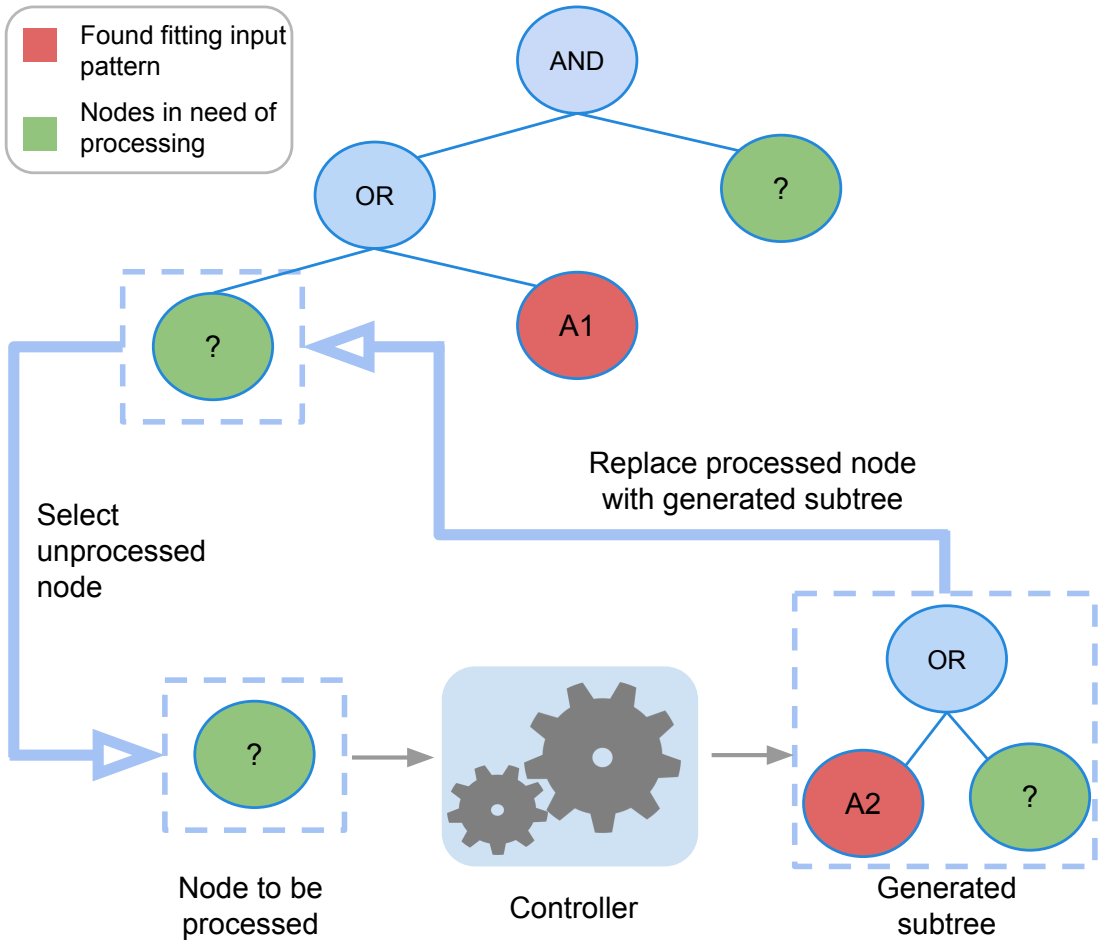

FIGURE C.2: A visual representation of the NNGS algorithm during the development of a solution tree.

Algorithm 3 gives a simple and more thorough explanation of NNGS. In line 2 the output values of the solution tree root node are set to the target output values of the problem at hand. The output values of a node are used, along with the reverse operators, by the controller (line 9) to generate the output values of the returned child nodes. The controller also sets the node type (they are either operators or input arguments) of the input parent node and generated child nodes.

Nodes which have been defined by the controller as input arguments (with labels: $A1$, $A2$, $A3$... etc.) can not have child nodes (they are by definition leaf nodes) and are therefore not processed further by the controller (line 6). When every branch of the tree ends in an input argument node, the algorithm halts.

Note that the controller may well generate a triplet where one or more of the child nodes require further processing. In this case the NNGS algorithm will pass these nodes back to the controller at a later stage before the algorithm ends. In effect, by using the controller component the NNGS algorithm simply writes out the solution tree.

Algorithm 3 The Node-by-Node Growth Solver
NNGS(target_outputs, controller)

```
1  solution_tree ← {}
2  root_node.outputs ← target_outputs
3  unprocessed_nodes ← {root_node}

4  while len(unprocessed_nodes) > 0 do
5      node_a ← unprocessed_nodes.pop()
                                              ▷ check for leaf node
6      if node_a.type = 'argument' then
7          solution_tree.insert(node_a)
8          continue                           ▷ move on to next node

9      node_a, node_b, node_c ←
           controller(node_a, target_outputs)
10     unprocessed_nodes.insert({node_b, node_c})
11     solution_tree.insert(node_a)

12 return solution_tree
```

C.5 Proof-Of-Concept Controller

Given an unprocessed node, the controller generates two child nodes and their output arrays using one of the four reverse operators. It also sets the operator type of the parent node to correspond with the chosen reverse operator that is used.

The ultimate goal of the controller is to assign an input argument to each generated child node. For example, suppose that the controller generates a child node with an output array $node\_b.\text{outputs} = [0, 1, 1, \#]$ and that an input argument is given by $A1 = [0, 1, 1, 0]$. In this case, $node\_b$ can be assigned (can represent) the input argument $A1$ because $[0, 1, 1, \#] = [0, 1, 1, 0]$. The algorithm halts once each leaf node has been assigned an input argument.

Before giving a detailed description of the proof-of-concept controller, there are a few important general points to stress: Firstly, the entire decision making process is deterministic. Secondly, the decision making process is greedy (the perceived best move is taken at each opportunity). Thirdly, the controller does not know the location of the input node within the solution tree. The controller has priori knowledge of the input argument arrays, the operators, and the reverse operators only. Furthermore, the controller, in its current state, does not memorise the results of it's past decision making. In this regard, when processing a node, the controller has knowledge of that node's output array only. In this way, the controller acts locally on each node. Multiple instances of the controller could act in parallel by processing all unprocessed nodes simultaneously.

C.5.1 Step-by-step

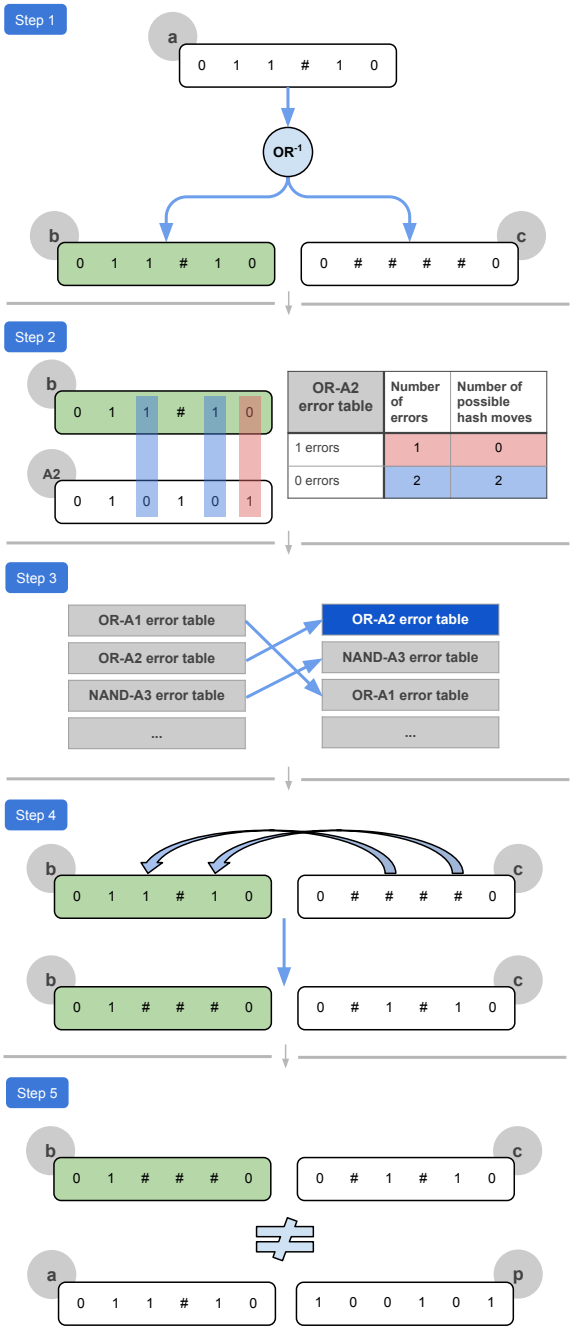

FIGURE C.3: Diagrammatic aid for the proof-of-concept controller.

This subsection will give a step-by-step run-through of the procedure undertaken by the proof-of-concept controller. Figure C.3 serves as a diagrammatic aid for each major step.

Step 1

Given an input (parent) node *node_a*, and for each reverse operator in Table C.1, the first step taken by the controller is to generate output arrays for each of the child nodes. In the example given in step 1 of Fig. C.3 only the OR^{-1} reverse operator is used. The OR^{-1} reverse operator generates # values in the child output arrays due to the following property $OR^{-1}(1) = (1, \#)$ or $(\#, 1)$. In this step, whenever the opportunity arises (regardless of the reverse operator employed), all generated # values within the child output arrays will be placed in the output array of the right child node *node_c*. For example in the case of $OR^{-1}(1)$: $(1, \#)$ will be used and not $(\#, 1)$.

Note that the reverse operators propagate all # elements from parent to child nodes. This feature is exemplified in step 1 of Fig. C.3 by the propagation of the # value at locus 4 of *node_a*.outputs to loci 4 of both *node_b*.outputs and *node_c*.outputs.

Step 2

By this step, the controller has generated four different (in general) *node_b*.outputs arrays, one for each reverse operator. The goal for this step is to compare each of those arrays to each of the possible input argument arrays ($A1$, $A2$... etc). As an example, in step 2 of Fig. C.3 the generated *node_b*.outputs array is compared to the $A2$ input argument array.

Two separate counts are made, one for the number of erroneous 0 argument values E_0 and one for the number of erroneous 1 argument values E_1 (highlighted in blue and red respectively in Fig. C.3). Two further counts are made of the number of erroneous *node_b* loci, for 0 and 1 input arguments values, which could have been # values (and therefore not erroneous) had the controller not specified in step 1 that all # values should be placed in the *node_c*.outputs array whenever possible. These last two statistics will be denoted by M_0 and M_1 for 0 and 1 input arguments values respectively. These four statistics form an *error table* for each reverse operator-input argument pair.

Step 3

In this step, the controller sorts the error tables by a number of statistics. Note that $M_0 - E_0$ and $M_1 - E_1$ are the number of remaining erroneous 0 argument values and erroneous 1 argument values respectively if all # values were moved from the *node_c*.outputs array to the *node_b*.outputs array whenever possible. To simplify matters we note that

$$\begin{aligned} &\text{if} \quad M_1 - E_1 \leq M_0 - E_0 \\ &\qquad \text{let } k = 1,\ j = 0 \\ &\text{otherwise} \\ &\qquad \text{let } k = 0,\ j = 1. \end{aligned} \tag{C.3}$$

Each error table is ranked by (in order, all increasing): $M_k - E_k$, E_k, $M_j - E_j$, E_j, and the number of # values in *node_c*.outputs. In a greedy fashion, the very best error table (lowest ranked) will be select for the next step (in Fig. C.3 the OR-$A2$ error table is selected). Note that the ranked list of error tables might need to be revisited later from step 5.

Step 4

The error table selected in step 3 effectively serves as an instruction which details how the *node_b*.outputs and
node_c.outputs arrays should be modified. The goal of the controller is to move the minimum number of # values from the *node_c*.outputs array to the *node_b*.outputs array such as to satisfy the error count for either 1's or 0's in one of the input arguments. In the example given in Fig. C.3, two # values in *node_c*.outputs are swapped with 1's in *node_b*.outputs.

Step 5

In this step, the algorithm checks that the generated
node_b.outputs and *node_c*.outputs arrays do not exactly equal either the parent node *node_a* or the grand parent node *node_p* (if it exists). If this check fails, the algorithm reverts back to step 3 and chooses the next best error table.

Step 6

The final step of the algorithm is to appropriately set the operator type of *node_a* given the final reverse operator that was used. In this step the algorithm also checks whether either (or both) of the child nodes can represent input arguments given their generated output arrays.

C.6 Experiments

The Boolean function synthesis benchmarks solved in this paper are standard benchmarks within GP research [1, 3, 7, 31]. They are namely: the comparator 6bits and 8bits (CmpXX), majority 6bits and 8bits (MajXX), multiplexer 6bits and 11bits (MuxXX), and even-parity 6bits, 8bit, 9bits, and 10bits (ParXX).

Their definitions are succinctly given in [1]:

"For an v-bit comparator Cmp v, a program is required to return true if the v/2 least significant input bits encode a number that is smaller than the number represented by the v/2 most significant bits. In case of the majority Maj v problems, true should be returned if more that half of the input variables are true. For the multiplexer Mul v, the state of the addressed input should be returned (6-bit multiplexer uses two inputs

TABLE C.1: Results for the NNGS algorithm when tested on the Boolean benchmarks, perfect solution were obtained for each run. BP4A columns are the results of the best performing algorithm from [1] (* indicates that not all runs found perfect solution). The RDO_p column is taken from the best performing (in terms of fitness) scheme in [3] (note that in this case, average success rates and average run times were not given).

	Run time [seconds]			Program size [nodes]			
	NNGS	BP4A	ILTI	**NNGS**	BP4A	ILTI	RDO
Cmp06	0.06	15	9	99	156	59	185
Cmp08	0.86	220	20	465	242	140	538
Maj06	0.19	36	10	271	280	71	123
Maj08	3.09	2019*	27	1391	563*	236	-
Mux06	0.21	10	9	333	117	47	215
Mux11	226.98	9780	100	12373	303	153	3063
Par06	0.35	233	17	515	356	435	1601
Par08	5.73	3792*	622	2593	581*	1972	-
Par09	25.11	-	5850	5709	-	4066	-
Par10	120.56	-	-	12447	-	-	-

to address the remaining four inputs, 11-bit multiplexer uses three inputs to address the remaining eight inputs). In the parity Par v problems, true should be returned only for an odd number of true inputs."

The even-parity benchmark is often reported as the most difficult benchmark [31].

C.7 Results and Discussion

The results are given in Table C.1 and show that the NNGS algorithm finds solutions quicker than all other algorithms on all benchmarks with the exception of the ILTI algorithm on the Mux11 benchmark. A significant improvement in run time was found for the Par08 benchmark.

The solution sizes produced by the NNGS algorithm are always larger than those found by the BP4A and ILTI algorithms with the exception of the Cmp06 results. The RDO scheme and ILTI algorithm both relay on traversing large tree libraries which make dealing with large bit problems very computationally intensive. As such, these methods do not scale well in comparison to the NNGS algorithm.

It is a clear that NNGS is weakest on the Mux11 benchmark. In this case a very large solution tree was found which consisted of 12,373 nodes. The multiplexer benchmark is significantly different from the other benchmarks by the fact that only four input arguments are significant to any single fitness case: the three addressing bits and the addressed bit. Perhaps this was the reason why the chosen methodology implemented in the controller resulted with poor results in this case.

C.8 Further Work

There are two possible branches of future research which stem from this work, the first centres around meta-GP. As a deterministic set of rules, the proof-of-concept controller is eminently suited to be encoded and evolved as part of a meta-GP system. The difficulty in this case will be in appropriately choosing the set of operators which would be made available to the population of controller programs.

The second avenue of research which stems from this work involves encoding the current proof-of-concept controller within the weights of a neural network (NN). This can be achieved through supervised learning in the first instance by producing training sets in the form of node triplets using the current controller. A training set would consist of randomly generated output arrays and the proof-of-concept controller generated child output arrays. In this way, the actual Boolean problem solutions do not need to be found before training.

As part of the task of find a better controller, the weights of the NN could be evolved using genetic algorithms (GA), similar to the method employed by [32]. The fitness of a NN weight set would correspond to the solution sizes obtained by the NNGS algorithm when employing the NN as a controller: the smaller the solutions, the better the weight set fitness. Using the proof-of-concept controller in this way would ensure that the GA population would have a reasonably working individual within the initial population.

A NN may also serve as a reasonable candidate controller for extending the NNGS algorithm to continuous symbolic regression problems. In this case, the input arguments of the problem would also form part of the NN's input pattern.

Bibliography

[1] Krzysztof Krawiec and Una-May O'Reilly. Behavioral programming: a broader and more detailed take on semantic gp. In *Proceedings of the 2014 conference on Genetic and evolutionary computation*, pages 935–942. ACM, 2014.

[2] Bartosz Wieloch and Krzysztof Krawiec. Running programs backwards: instruction inversion for effective search in semantic spaces. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1013–1020. ACM, 2013.

[3] T Pawlak, Bartosz Wieloch, and Krzysztof Krawiec. Semantic backpropagation for designing search operators in genetic programming. 2014.

[4] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[5] Howard Barnum, Herbert J Bernstein, and Lee Spector. Quantum circuits for or and and of ors. *Journal of Physics A: Mathematical and General*, 33(45):8047, 2000.

[6] Shu-Heng Chen. *Genetic algorithms and genetic programming in computational finance*. Springer Science & Business Media, 2012.

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

[8] Genetic and evolutionary computation conference 2015. http://www.sigevo.org/gecco-2015/. Accessed: 2015-06-24.

[9] Robyn Ffrancon and Marc Schoenauer. Greedy semantic local search for small solutions. In S. Silva and A. Esparcia, editors, *Companion Proceedings GECCO*. ACM, 2015. To appear.

[10] Artificial evolution 2015. https://ea2015.inria.fr/. Accessed: 2015-06-24.

[11] Robyn Ffrancon. Retaining experience and growing solutions. *arXiv preprint arXiv:1505.01474*, 2015.

[12] John R Koza, Forrest H Bennett, David Andre, Martin A Keane, and Frank Dunlap. Automated synthesis of analog electrical circuits by means of genetic programming. *Evolutionary Computation, IEEE Transactions on*, 1(2):109–128, 1997.

[13] James McDermott, David R White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Jaskowski, Krzysztof Krawiec, Robin Harper, Kenneth De Jong, et al. Genetic programming needs better benchmarks. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 791–798. ACM, 2012.

[14] Lee Spector, David M Clark, Ian Lindsay, Bradford Barr, and Jon Klein. Genetic programming for finite algebras. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1291–1298. ACM, 2008.

[15] Leonardo Vanneschi, Mauro Castelli, and Sara Silva. A survey of semantic methods in genetic programming. *Genetic Programming and Evolvable Machines*, 15(2):195–214, 2014.

[16] Pablo Moscato et al. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech concurrent computation program, C3P Report*, 826:1989, 1989.

[17] Ferrante Neri, Carlos Cotta, and Pablo Moscato. *Handbook of memetic algorithms*, volume 379. Springer, 2012.

[18] Yuichi Nagata. New eax crossover for large tsp instances. In *Parallel Problem Solving from Nature-PPSN IX*, pages 372–381. Springer, 2006.

[19] Nicholas Freitag McPhee, Brian Ohs, and Tyler Hutchison. Semantic building blocks in genetic programming. In *Genetic Programming*, pages 134–145. Springer, 2008.

[20] Krzysztof Krawiec and Tomasz Pawlak. Approximating geometric crossover by semantic backpropagation. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 941–948. ACM, 2013.

[21] David E Goldberg. Zen and the art of genetic algorithms. In *International Conference on {G} enetic {A} lgorithms\'89*, pages 80–85, 1989.

[22] Arpit Bhardwaj and Aruna Tiwari. A novel genetic programming based classifier design using a new constructive crossover operator with a local search technique. In *Intelligent Computing Theories*, pages 86–95. Springer, 2013.

[23] Khaled Rasheed and Haym Hirsh. Informed operators: Speeding up genetic-algorithm-based design optimization using reduced models. In *GECCO*, pages 628–635, 2000.

[24] Pu Wang, Ke Tang, Edward PK Tsang, and Xin Yao. A memetic genetic programming with decision tree-based local search for classification problems. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pages 917–924. IEEE, 2011.

[25] Brent E Eskridge and Dean F Hougen. Memetic crossover for genetic programming: Evolution through imitation. In *Genetic and Evolutionary Computation–GECCO 2004*, pages 459–470. Springer, 2004.

[26] Masayuki Yanagiya. Efficient genetic programming based on binary decision diagrams. In *Evolutionary Computation, 1995., IEEE International Conference on*, volume 1, page 234. IEEE, 1995.

[27] Richard M Downing. Evolving binary decision diagrams using implicit neutrality. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 3, pages 2107–2113. IEEE, 2005.

[28] Hidenori Sakanashi, Tetsuya Higuchi, Hitoshi Iba, and Yukinori Kakazu. Evolution of binary decision diagrams for digital circuit design using genetic programming. In *Evolvable Systems: From Biology to Hardware*, pages 470–481. Springer, 1997.

[29] Holger H Hoos and Thomas Stützle. *Stochastic local search: Foundations & applications*. Elsevier, 2004.

[30] Alberto Moraglio, Krzysztof Krawiec, and Colin G Johnson. Geometric semantic genetic programming. In *Parallel Problem Solving from Nature-PPSN XII*, pages 21–31. Springer, 2012.

[31] John R Koza. Hierarchical automatic function definition in genetic programming. In *FOGA*, pages 297–318, 1992.

[32] Jan Koutník, Giuseppe Cuccu, Jürgen Schmidhuber, and Faustino Gomez. Evolving large-scale neural networks for vision-based reinforcement learning. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1061–1068. ACM, 2013.