Scalable Genetic Programming by Gene-Pool Optimal Mixing and Input-Space Entropy-Based Building-Block Learning

Marco Virgolin
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
Marco.Virgolin@cwi.nl

Tanja Alderliesten
Academic Medical Center
Amsterdam, The Netherlands
T.Alderliesten@amc.uva.nl

Cees Witteveen
Delft University of Technology
Delft, The Netherlands
C.Witteveen@tudelft.nl

Peter A.N. Bosman
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
Peter.Bosman@cwi.nl

ABSTRACT

The Gene-pool Optimal Mixing Evolutionary Algorithm (GOMEA) is a recently introduced model-based EA that has been shown to be capable of outperforming state-of-the-art alternative EAs in terms of scalability when solving discrete optimization problems. One of the key aspects of GOMEA's success is a variation operator that is designed to extensively exploit linkage models by effectively combining partial solutions. Here, we bring the strengths of GOMEA to Genetic Programming (GP), introducing GP-GOMEA. Under the hypothesis of having little problem-specific knowledge, and in an effort to design easy-to-use EAs, GP-GOMEA requires no parameter specification. On a set of well-known benchmark problems we find that GP-GOMEA outperforms standard GP while being on par with more recently introduced, state-of-the-art EAs. We furthermore introduce Input-space Entropy-based Building-block Learning (IEBL), a novel approach to identifying and encapsulating relevant building blocks (subroutines) into new terminals and functions. On problems with an inherent degree of modularity, IEBL can contribute to compact solution representations, providing a large potential for knock-on effects in performance. On the difficult, but highly modular Even Parity problem, GP-GOMEA+IEBL obtains excellent scalability, solving the 14-bit instance in less than 1 hour.

CCS CONCEPTS

•Computing methodologies → Heuristic function construction; Genetic programming;

KEYWORDS

program synthesis, genetic programming, linkage learning, optimal mixing, building blocks

ACM Reference format:
Marco Virgolin, Tanja Alderliesten, Cees Witteveen, and Peter A.N. Bosman. 2017. Scalable Genetic Programming by Gene-Pool Optimal Mixing and Input-Space Entropy-Based Building-Block Learning. In *Proceedings of GECCO '17, Berlin, Germany, July 15-19, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3071178.3071287

GECCO '17, Berlin, Germany

DOI: http://dx.doi.org/10.1145/3071178.3071287

1 INTRODUCTION

When a problem's structure has some inherent degree of modularity, being able to efficiently and effectively exploit this modularity in an Evolutionary Algorithm (EA), e.g., by recombining partial solutions, can lead to better solutions much faster than when using only blind variation operators [18]. The term *schemata* is often used in Genetic Algorithms (GAs) to refer to such partial solutions, which can be moderately to completely independent from each other.

In Black-Box Optimization (BBO), it is unknown how schemata are encoded, hence it is not possible to design any specific recombination operator beforehand that prevents their disruption when mixing solutions. In an attempt to learn and exploit problem structure, model-based EAs use a model to capture such structure [3]. In the case of BBO, model instances are inferred from the *genotype* (i.e., the encoding) of promising solutions.

In Genetic Programming (GP), the term *Building Blocks* (BBs) typically refers to connected parts of the genotype (i.e., connected nodes in tree-based GP) that represent useful subroutines. Whereas solutions in GAs have a fixed size and the main focus is to avoid the disruption of schemata, solutions in GP are typically free to grow. Therefore, many studies have explored steps to re-use BBs by encapsulating them into compact representations. With one of the first attempts, the Automatically Defined Functions (ADFs) [11], it has been shown that the re-use of BBs can be extremely beneficial, making GP capable of tackling very difficult, yet highly modular problems such as Even Parity. Many different approaches have been proposed in the last 25 years (see Sec. 5.2 of [12] for an overview). However, none of them has shown clear superiority in systematically identifying salient BBs [4]. Some works even synthesize BBs from randomly chosen subtrees [1, 15]. Other proposals relax the BBO hypothesis substantially to synthesize BBs from successful runs on smaller problem instances [8, 9].

Our purpose is to introduce novel, general, and principled ways to identify and exploit problem structure in tree-based GP. As a first contribution, we bring key strenghts of the Gene-pool Optimal Mixing Evolutionary Algorithm (GOMEA) to tree-based GP, resulting in GP-GOMEA. GOMEA is a model-based EA which performs a memetic variation of solutions by extensively exploiting

linkage information, i.e., strong interdependencies between parts of the genotype [17]. Our second contribution is a novel method to identify and encapsulate BBs into new terminals and functions in Boolean problems, thereby enhancing the search space with atomic representations of partial solutions. We call this method Input-space Entropy-based Building-block Learning (IEBL). IEBL is inspired by information theory and construction heuristics for classification trees [2], and can potentially be applied to a number of GP algorithms. To the best of our knowledge, no similar approach to identify salient BBs in GP has ever been proposed. Finally, under the hypothesis of no knowledge on the problem and for the sake of usability, we set out to design this algorithm to require no parameter specification.

2 GP-GOMEA

The current closest GOMEA implementation on which this GP version is based on is described in [17]. The general GOMEA outline is depicted in Algorithm 1. At the top level, GOMEA has the characteristics of any EA with population initialization and the generational loop that continues until a termination criterion is met (e.g., population convergence, evaluations limit, time limit). A generation consists of the learning of a linkage model F (which may be provided beforehand if the problem structure is known a priori) and the applying of the variation operator GOM to each solution in the population, which extensively exploits F to improve a solution.

Algorithm 1 GOMEA general outline

```
1 procedure RUNGOMEA(n)
2     P ←initializePopulation(n)
3     while ¬shouldTerminate() do
4         F ← buildLinkageModel(P)
5         for P_i ∈ P do
6             O_i ← GOM(P_i, F, P)
7         P ← O = {O_1, ..., O_n}
```

2.1 Genotype

Although the original implementation of GOMEA works on fixed-length strings of binary variables, handling variables of higher cardinality is straightforward. This representation is the first step in using GOMEA for GP, as we use it to map discrete values to program functions and program inputs. Like in Standard GP (SGP), solutions in GP-GOMEA are trees of variable size composed of terminal and function nodes. Trees can be encoded as fixed-length strings using pre-order tree-traversal (Figure 1). All nodes but the ones at maximum depth always have r child nodes, with r the maximum arity (i.e., number of expected inputs) of the function nodes. We make it possible for GOMEA to work with variable-size trees even though they are encoded with fixed-length strings. Trees always have a maximum height. Syntactically, trees are always full, but semantically they are not. If a terminal appears in an internal node, the subtrees below it are disregarded. Moreover, for function nodes with arity lower than r, only the leftmost child nodes are evaluated. Hence, some nodes are *introns*, i.e., they will be ignored during the evaluation of the tree.

2.2 Linkage Models

As in the original GOMEA, GP-GOMEA uses the Family Of Subsets (FOS) as linkage model. The FOS is a set of sets which contain *loci*,

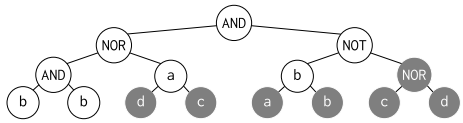

Figure 1: GP tree encoded by the fixed-length string of size 15 "&+&bbadc¬bab+cd". Gray nodes are introns.

i.e., indices representing positions in the genotype. Each one of these sets specifies which parts of the genotype should be replaced *en bloc* during variation. Note that a FOS containing all and only singletons of each locus, i.e., $F = \{\{0\}, \{1\}, \ldots, \{l-1\}\}$, with l the length of the genotype, models complete independence among loci. We call this FOS that allows only the variation of one locus at a time *Univariate* (U). In this paper, we analyze the contribution of three different FOSs: U, *Linkage Tree* (LT) and *Random Tree* (RT).

We consider LT as it has so far been found to lead to the best performance on a number of different BBO problems [17]. A key strength of this model is that it can capture at the same time multiple levels of dependency (linkage) among loci. An LT can be seen as a tree where the leaves are singletons (i.e., U) while its internal nodes are built by merging sets in an iterative fashion, up to reaching the root, i.e., the set that contains all loci. An LT may be fixed a priori, but especially in a BBO setting, it is learned from the population at each generation (line 4 of Algorithm 1). Specifically, a measure of linkage between couples of loci is measured by means of mutual information, which is *the* measure of mutual dependence between two variables in information theory. New sets are iteratively built by merging sets with the highest mutual information. Using only combinations of mutual information between pairs of variables, the hierarchical structure of dependencies expressed by an LT can be efficiently learned in $O(|\mathcal{P}|l^2)$, with $|\mathcal{P}|$ the population size [7].

Lastly, RT is built like LT, but using random information instead of measured linkage. This FOS enables the variation of multiple parts of the genotype like LT, but does not assume that specific parts of the genotype should be kept intact. RT is thus a model that enables blind variation that differs from the classic GP subtree crossover in that any configuration of nodes can be swapped.

2.3 Gene-Pool Optimal Mixing

The variation operator GOM, that also incorporates selection, always generates an offspring that is at least as fit as the parent. Different from standard crossover in GP where full subtrees are swapped, in our GP-GOMEA GOM mixes parts of the fixed-size genotype, i.e., it mixes tree nodes. Moreover, instead of generating two new solutions from two parents, it creates an offspring by iteratively mixing a parent solution with multiple other solutions.

The procedure is described in Algorithm 2. Given an input solution s, an identical offspring o is made, and each set F_i of the FOS F is used to try to improve o. Given F_i, a donor d is randomly picked from the population, and the symbols of o at the loci specified by F_i are replaced with the ones of d at the same loci. This means that the jth symbol of o can be replaced *only* with the jth symbol of d. This is not done when F_i contains all loci, since it would only fully replace o, or when all loci in F_i identify introns of o, because the semantic of the tree would not change. If the mixing results in a syntactical change of o, then this new solution is evaluated. The changes are kept only if the fitness of o does not worsen.

If o never changes during this first phase and no new best fitness has been found in the last $1 + \log_{10}(|\mathcal{P}|)$ generations, then the *forced improvement phase* is entered. In this phase Optimal Mixing is performed again, but this time o is mixed only with s^{elitist}, the best solution ever found. Also, this time changes to o are accepted only in case of strict fitness improvement. Moreover, upon an accepted change, the procedure is halted. If even this does not lead to any change of o, then o becomes an exact copy of s^{elitist}.

Because root nodes can only be exchanged with root nodes, in the classic ramped half-and-half generation of the initial population the first symbol is always initialized to represent a function.

Algorithm 2 Gene-Pool Optimal Mixing

```
1  function GOM(s, F, P)
2      o ← s; fitness[o] ← fitness[s]
3      b ← o; fitness[b] ← fitness[o]
4      I ←inactiveNodes(o)
5      R ←randomPermutation({0, 1, ..., |F − 1|})
6      c ← 0
7      for i ∈ {0, 1, ..., |F − 1|} do
8          F_i ← F[R[i]]
9          if |F_i| ≠ |o| & F_i ⊈ I then
10             d ←randomDonorSolution(P)
11             o_{F_i} ← d_{F_i}
12             if o ≠ d then
13                 evaluateFitness(o)
14                 if fitness[o] ≥ fitness[b] then
15                     b_{F_i} ← o_{F_i}; fitness[b] ← fitness[o]
16                     I ←inactiveNodes(o)
17                     c ← 1
18                 else
19                     o_{F_i} ← b_{F_i}; fitness[o] ← fitness[b]
20     if c = 0 & noImprovementsStretch() then
21         forcedImprovementOM(o, F)
22     return(o)
```

2.4 Partial Evaluations

To enhance the speed of evaluating solutions a simple mechanism can be used in tree-based GP to perform partial evaluations. We use this also for GP-GOMEA. This is done by maintaining the output of all tree nodes (i.e., string symbols) in memory. Note that introns do not have any output. During GOM, track is kept of which nodes are changed. Consequently, only subtrees where at least one (active) node changed, need to be re-evaluated, whereas the roots of unchanged subtrees can immediately return their cached output.

2.5 Interleaved Multistart Scheme

The task of sizing a problem-specific population and genotype (string length or, equivalently, tree height) is crucial in many EAs. Tuning such parameters is often tedious and time-consuming but also necessary to ensure efficiency and to guarantee the successful discovery of (near-)optimal solutions. Setting these parameters wrong can give a vastly wrong impression of an algorithm's capabilities. For this reason, we designed GP-GOMEA so that it does not require the user to specify any parameter. A similar scheme as the one proposed in [14] is adopted, where multiple runs of the algorithm with different parameter settings are interleaved. We call this scheme Interleaved Multistart Scheme (IMS).

Specifically, every g generations of a GP-GOMEA run, another run with double the population size performs 1 generation. This is repeated recursively. Similarly, the maximum tree height (and thus the encoding string length) increases by 1 every 2 runs.

The first run is initialized with a population size of 2 in ramped half-and-half, and with a maximum tree height such that full trees have a number of nodes at maximal depth equal or bigger than the number of inputs of the problem (i.e., for a problem with n inputs and functions of maximum arity r, it is $\lceil \log_r(n) \rceil$).

A copy of s^{elitist}, the best solution ever found by any run so far, is stored and used by all runs in the forced improvement phase of GOM (Algorithm 2, line 21). If a new best solution s^{elitist} is found the size of which (i.e., maximum tree height or, equivalently, string length) is smaller than the size of solutions evolved by a run R, then a copy of s^{elitist} is made for R that has the same size of the solutions of R and in which empty loci are filled with random introns. If the new best solution s^{elitist} has a larger size than the one of solutions evolved by R, then R is immediately terminated.

Other criteria for the termination of a run R are the following: (i) the population of R converged to all identical solutions; (ii) a run R' with larger population achieved a better average fitness than R or than a run R'' with bigger population size than R. Finally, the whole multi-run scheme can be terminated at a specific threshold by specifying a maximum number of evaluations or seconds.

3 IEBL

We here describe a novel method to identify and encapsulate useful, small trees into new terminals and functions, called Input-space Entropy-based Building-block Learning (IEBL). In this context, we use the term BBs to refer to such small trees. IEBL is aimed at improving the search process on Boolean problems which exhibit a degree of decomposability, i.e., for which meaningful BBs exist.

The identification of salient BBs is based on fitness cases (pairs of input values and desired output values) and is inspired by information theory and heuristics to build classification trees. To the best of our knowledge, no similar approach exists in GP.

Whereas the identification method attempts to find those BBs that represent partial solutions to the problem, the encapsulation of BBs changes the search space by providing the EA with compact representations of higher-level functionalities that can be used in the search process. Moreover, IEBL can be applied iteratively, using encapsulated BBs to generate higher-order BBs.

Here, we show how IEBL can be used in GP-GOMEA, but its salient concepts can straightforwardly be used in a number of other GP paradigms. The following sections explain the method in detail.

3.1 Identification of BBs

A dedicated population of small trees is used. This is due to some early experiments, confirming literature [4], where we observed that the frequency of known good subtructures in the population (i.e., the `XOR` and `XNOR` functions for the Even Parity problem) does not necessarily increase during optimization. To generate the dedicated population, we use a slight variation of the ramped half-and-half method. Specifically, roots are always functions and for each tree a subset of the set of all terminal nodes $\mathcal{T}$ is used, with cardinality between 2 and $|\mathcal{T}|$. This increases the redundancy of terminals contained in candidate BBs, increasing the probability of generating complex interactions between few terminals.

Let $\mathcal{I}$ be the set of input variables. Given a BB b, we say that b embodies an input variable i if there exists at least one non-intron

Table 1: BB identification in IEBL. Columns are the fitness cases, O^* is the desired output, O the observed output for the tree depicted in Figure 1. Entropy is measured on red cells.

a	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
b	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
c	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
d	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
O	1	**1**	**1**	1	0	**0**	**0**	0	0	**0**	**0**	0	**0**	0	0	**0**
O^*	1	0	0	1	0	1	1	0	0	1	1	0	1	0	0	1

terminal node that represents i in b. Let $\mathcal{J} \subset \mathcal{I}$ be the set of input variables not embodied by b. We only consider BBs for which $\mathcal{J} \neq \emptyset$, as they represent partial solutions using part of the inputs. Let $\mathcal{E}$ be the set of fitness cases for which the execution of b returns a wrong Boolean output. If $\mathcal{E} = \emptyset$, b is a solution to the problem and the EA is terminated. A quality-score is assigned to b by looking at the values taken by the input variables of $\mathcal{J}$ in $\mathcal{E}$. Specifically, the joint entropy of the values assumed by the inputs of $\mathcal{J}$ in $\mathcal{E}$ is measured. Lower entropy is considered to be better because this means that the fitness cases that are still wrong have more regularities and thus represent a less complex problem to be solved.

For example, consider the task of regressing a circuit that, given 4 bits, returns 1 when an even number of bits are set to 1 (4-bits Even Parity). The BB consisting of the tree in Figure 1 outputs 1 only when the input variables a and b are 0. Also, $\mathcal{J} = \{\mathsf{c}, \mathsf{d}\}$. $\mathcal{E}$ contains 8 cases. Table 1 shows the configurations of the input values of c and d over which the entropy is computed in red. Since "01" and "10" appear each in 3 out of 8 cases, while "00" and "11" appear each 1 out of 8 cases, the entropy is $E = -\sum p \log p = -2\frac{3}{8}\log(\frac{3}{8}) - 2\frac{1}{8}\log(\frac{1}{8})$.

Some BBs are discarded during this procedure, namely those (i) that have the same output of another BB, but higher or equal entropy; (ii) for which $\mathcal{J} = \emptyset$; (iii) whose output is *always-false* or *always-true*; (iv) for which $\mathcal{I} - \mathcal{J} = \{i\}$, since the only realizable functions of i are *always-false*, *always-true*, identity and negation.

3.2 Encapsulation of BBs — Terminal Nodes

After the identification method has computed the entropy of BBs, we encapsulate into new terminals the best (i.e., with lowest entropy) $|\mathcal{I}|$ BBs, thus doubling the number of terminals. Expanding the terminal set effectively changes the search space. We limit the number of new terminals to avoid an excessive complication of the search space. If more than $|\mathcal{I}|$ BBs are found with minimal entropy, then random $|\mathcal{I}|$ ones are kept and the others are discarded.

To enable running IEBL when earlier executions already identified and encapsulated new BBs, we keep track of which input variables are embodied. This allows to always define the set $\mathcal{J}$ needed to compute the entropy.

3.3 Encapsulation of BBs — Function Nodes

The best $|\mathcal{I}|$ BBs are also used for encapsulation into new function nodes. Let r be the arity of a BB, i.e., the number of different (non-intron) terminal nodes in it. The functional encapsulation of this BB is achived by generating a function node that accepts r children and, given one of the 2^r possible binary configurations of the inputs, returns the output of the BB for that configuration. We discard BBs leading to duplicate function nodes, i.e., those whose arity and output are identical to an already-encapsulated one or to a function from the starting set $\mathcal{F}$. Further, we discard BBs which realize *always-false*, *always-true* and functions of arity 1.

Similarly to what is done for terminal nodes, we impose a fixed limit of $|\mathcal{F}|$ new function nodes to expand $\mathcal{F}$. If more than $|\mathcal{F}|$ BBs have minimal entropy, then $|\mathcal{F}|$ at random are kept.

3.4 Implementation of IEBL in GP-GOMEA

To alleviate users from having to choose the dedicated population size and tree height for IEBL, we propose a scheme to include IEBL in GP-GOMEA that requires no parameter specification.

IEBL is applied at the start of each new GP-GOMEA run to expand the terminal and function sets. In particular, for the ith run, IEBL is iterated i times consecutively to discover higher-order BBs. This means that the nodes created by the jth iteration of IEBL are used (together with the starting functions and terminals) for the generation of the dedicated population of the $j+1$th iteration of IEBL. For the first GP-GOMEA run, the dedicated population size for IEBL is set to $|\mathcal{F}|(|\mathcal{F}| + t)^r$, which corresponds to the number of possibilities for the first two levels of an r-ary tree with any function as root, and with any function or terminal, among t different ones, as children of the root. For the function set of the Boolean benchmark problems it is $r = 2$, and we fixed $t = 4$ to ensure starting from a moderate dedicated population size (i.e., 256 trees). The dedicated population size is doubled for each new GP-GOMEA run, and the height of the trees constituting the candidate BBs is initially set to 2 and is incremented by 1 every 4 runs. In other words, IEBL is applied *once* before the first run of GP-GOMEA, using a dedicated population size of 256 with trees of height 2; IEBL is applied i times before the ith run, using a dedicated population size of $256 \times 2^{i-1}$ in each iteration, with trees of height $2 + \lfloor i/4 \rfloor$. If the ith IEBL finds a BB with an entropy of 0, then the number of iterations for all next IEBLs is frozen to i. Because IEBL provides nodes that inherently embody trees of a certain height, we lower by 1 the starting tree height of the IMS of GP-GOMEA. To avoid an unbounded growth of the genotype, we limit the learning of new node functions to BBs with arity 2 only. Finally, unless a BB entropy of 0 is reached, if no lower entropy is found after four consecutive uses of IEBL (i.e., four new runs of GP-GOMEA), then IEBL is disabled: all current GP-GOMEA runs are terminated (the elite solution is also forgotten), and subsequent runs will no longer use IEBL to learn and use BBs.

4 EXPERIMENTAL SETUP

The performance of GP-GOMEA[1] is tested in terms of scalability on well-known GP benchmark problems. This enables us to compare our algorithm with SGP, but also with state-of-the-art work.

All experiments were executed on a machine mounting 2 Intel® Xeon® CPU E5-2699 v4 @ 2.20GHz and 630 GB of RAM. Each experiment consists of 30 independent runs and only successful experiments are considered. Runs exceeding a time limit of 24 hours or a memory limit of 500 GB are aborted.

4.1 Benchmark Problems

We consider two sets of benchmark problems. The first set is composed of artificial problems often used to assess the performance of model-based EAs, because of their focus on obtaining specific substructures in the genotype. The second set considers well-known

[1]The code is freely available on the homepage of the last author.

regression problems of Boolean circuits, which differ from the problems in the first set in that the solution encoding is much more redundant, so that very different solutions have the same output. In all problems, maximization of the fitness is sought.

Artificial problems. Order is a GP version of the well-known OneMax problem in GA research and is known to be easy to solve for SGP [16]. Given a problem size n, the terminal set consists of $2n$ node variables X_i and their complement $\bar{X}_i$. The function set contains only one node which is a placeholder for a function of arity 2 with no semantic meaning. The output O of a tree is a list of its inputs derived from the inorder parsing of its nodes, such that there are no duplicates and only one of the two complementary inputs is present, depending on which is encountered first in the parsing. E.g., if $\{X_3, \bar{X}_0, X_1, X_1, X_0, \bar{X}_2\}$ are the inputs appearing in the inorder parsing, the tree outputs $O = \{\bar{X}_0, X_1, \bar{X}_2, X_3\}$. The fitness is $f^n_{\text{order}} = |O \setminus \{\bar{X}_0, \bar{X}_1, \ldots, \bar{X}_{n-1}\}|$. In other words, the optimal solution is a tree where X_i is present for each i, and if $\bar{X}_i$ is also present, then it appears after X_i in the inorder parse.

The problem Trap employs the same terminal and function sets of Order, but is considered a hard problem for SGP [16]. This is because of its deceptive fitness function defined for blocks of k variables, which is inspired by the well-known deceptive trap functions in GA research. The deceptive attractor corresponds to the number of $\bar{X}_i$ in the output, while the needle in the haystack is the optimum of Order. Specifically, for a block of k variables,

$$f^k_{\text{Trap}} = \begin{cases} 1 & \text{if } f^k_{\text{order}} = k \\ 0.75\left(1 - \frac{f^k_{\text{order}}}{k-1}\right) & \text{if } f^k_{\text{order}} < k \end{cases}$$

We denote with Trap-3 and Trap-4 the problem with trap length k of 3 and 4 respectively. The problem size n is the number of traps.

Like Order and Trap, the tunable benchmark problem introduced in [10], here denoted with KÜ (from the authors' surnames), uses binary trees, but with a predefined maximum height. The aim is to synthesize a tree in which function nodes are arranged according to positional constraints. The size of the problem n corresponds to the maximum tree height. The terminal set contains only one node, while the function set contains functions of arity 2, i.e., $\mathcal{F} = \{\mathcal{F}_0, \mathcal{F}_1, \ldots\}$. $\mathcal{F}$ can be changed together with n to tune problem difficulty. In this work, we always consider $\mathcal{F} = \{\mathcal{F}_0, \ldots, \mathcal{F}_{2n-1}\}$. The output of a tree is the output of its root. A function node outputs the weighted sum of its 2 children's outputs, determined according to the constraints: (i) the index of the parent function must be lower than that of its children; (ii) the index of child 1 must be smaller than the index of child 2. The terminal has no index and does not violate constraints. If child 1 violates the first constraint, then its output is penalized with a weight of η, and similarly for child 2. If the second constraint is violated, then the output of both children is penalized with η (we use $\eta = 0.25$). Since the maximum tree height *is* the problem size n, for this problem the maximum tree height in the IMS is fixed to n.

Boolean problems. These problems are defined with fitness cases, i.e., pairs of input and desired output values. Different from the artificial problems, inputs are now binary. In all these problems the aim is to synthesize a tree that realizes a Boolean circuit which satisfies all fitness cases, giving the correct output for any input configuration. The fitness of a solution is the number of correct fitness outputs. Boolean circuits we consider are: Comparator, Even Parity, Majority and Multiplexer. The terminal set contains a terminal for each input, and the function set contains the logic functions `AND`, `OR`, `NAND`, and `NOR`.

A circuit realizes the n-inputs Comparator if it outputs *true* only if the first $\lceil n/2 \rceil$ bits represent a number that is lower than the one encoded by the second $\lfloor n/2 \rfloor$ bits. The Even Parity problem of n inputs expects the output *true* when the number of input bits set to 1 is even and *false* otherwise. In Majority, the solution must return *true* only when at least $\lceil n/2 \rceil$ out of a total of n input bits are set to 1. Finally, a Multiplexer has $n = m + 2^m$ input bits: the first m bits encode the address, the second 2^m bits encode the data. A circuit satisfies the Multiplexer problem if it always outputs the value of the data bit encoded by the address.

4.2 Standard GP and State-of-the-Art

We compare the scalability of GP-GOMEA with SGP and two state-of-the-art algorithms. For Order and Trap, the model-based algorithm extended Compact Genetic Programming (eCGP) is considered [16]. For the Boolean benchmark problems, we consider the very recent algorithms based on Semantic Backpropagation (SB): the Iterated Local Tree Improvement (ILTI) [6] and GP using the Random Desired Operator (RDO) [13]. These two algorithms inherently embody the partial evaluations method with which GP-GOMEA is also equipped, since they require the memorization of each node output. Differently from GP-GOMEA and SGP, these former 2 algorithms always require the definition of fitness cases.

For eCGP, we consider the scalability reported in [16]. The performance is obtained with an empirically pre-computed good population size and on a predefined maximum tree height.

Both ILTI and (GP with) RDO rely on SB, a recent technique in which improvements at the level of single fitness cases are sought. SB is applied top-down (from the root to the leaves) and computes, for each node, a desired output O^*. For the root, O^* is the vector that satisfies all fitness cases of the problem. The desired output of a child of the root is computed by inverting the root function, using O^* and the current output of its other children as arguments. The design of inverted functions is not always straightforward, as sometimes a solution does not exist (e.g., looking for an input for AND with desired output 1 when the other input is 0) or can assume any value (e.g., looking for an input for OR with desired output 1 when the other input is 1). Among several differences, ILTI is an (1,1)-EA, while RDO is population based. Here ILTI is used in the best performing configuration, that is with a (maximum, when more are possible) library size of 450 full trees of height 2. Similarly, for RDO we adopt the best-performing configuration on Boolean problems (named RDO_p in [13]), which uses a dynamic library of semantically-unique trees taken from the subtrees in the population instead of a fixed-size library.

We propose two configurations for SGP: SGP_{param}, with hand-picked population size and initial tree height as typically done in literature, and SGP_{IMS}, enclosed in a scheme similar to the IMS of GP-GOMEA. For the former, we set the initial tree height to 6 and the maximum allowed one to 12. The population size for Order is set to 2^{n+2} (easy problem), for Trap-3 and KÜ to 2^{n+7} (difficult problem), and for Trap-4 to 2^{n+9} (very difficult problem), with n

the problem size. As to the Boolean problems, the population size for Comparator and Majority is set to 2^{n+5} (medium difficulty). For Even Parity, known to be difficult for SGP [11], the population size is 2^{n+7}. For the 3, 6, and 11-bits Multiplexer the population size is set to 256, 1024, and 4096, respectively. SGP$_{IMS}$ works with no parameter specification, within a IMS scheme that differs from the one of GP-GOMEA in the following aspects: (i) there is no common elitist solution among runs, nor a stopping criteria related to it; (ii) a run performs 1 generation every 8 generations of the smaller run; (iii) the initial tree height h is computed as the maximum tree height of GP-GOMEA (Section 2.5), and the maximum height is $h+4$. We experimentally found that increasing the intervals of the IMS is beneficial for SGP$_{IMS}$, since it performs much less evaluations than GP-GOMEA (with any of the 3 linkage models) per generation. Furthermore, we set a maximum tree height bigger than the initial tree height because the standard crossover and mutation swap and generate subtrees of arbitrary height. For both SGP configurations, we set tournament selection size to 4, probability of crossover to 0.9, probability of mutation to 0.1, and reproduction of the best solution. Finally, we equip SGP with the same caching method of node outputs used in GP-GOMEA, to perform partial evaluations.

5 RESULTS & DISCUSSIONS

Scalability graphs are provided in Figure 3. GP-GOMEA, SGP, ILTI and RDO use partial evaluations. When IEBL is applied, evaluations of BBs candidates are also counted, and the number of nodes is obtained by recursively unwrapping encapsulated BBs.

Artificial problems. Results show that GP-GOMEA with LT as the linkage model (GP-GOMEA$_{\text{LT}}$) is generally the best performing algorithm in all metrics: number of evaluations, time, and size of the final solution. On the easy Order problem, for which we run a limited number of instances, no marked difference between both SGP configurations and GP-GOMEA with any linkage model is observed. However, the more difficult the problem, the more GP-GOMEA$_{\text{LT}}$ shows superior performance. On Trap-3, SGP$_{param}$ markedly outperforms SGP$_{IMS}$, and performs slightly better than GP-GOMEA$_{\text{LT}}$. This result is possibly due to the immediate employment of a big population size in SGP$_{param}$ and a good setting of the initial tree height. On Trap-4 and KÜ, however, GP-GOMEA$_{\text{LT}}$ scales better than any other algorithm, showing an effective capability of learning and exploiting the problem structure. On the evaluations of Order and Trap-3, the scalability of eCGP of $O(n^{2.86})$ and $O(n^{3.18})$ respectively, as reported in [16], is shown. Although it may appear that GP-GOMEA$_{\text{LT}}$ performs slightly worse than eCGP, it is important to remember that the performance of eCGP is obtained on an empirically pre-computed good population size and on a predefined initial tree height, whereas GP-GOMEA runs according to the IMS.

Boolean problems. On the Boolean problems, the difference between the linkage models LT and RT used for GP-GOMEA is far less pronounced, suggesting that the LT is not capable of modeling key linkage information to help increase efficiency substantially. This may well be because of the high redundancy in the representation of solutions, and the consequential fact that locus-based dependence is not the most important source of problem structure.

GP-GOMEA$_{\text{LT}}$ and SGP$_{IMS}$ show similar scalability overall, with exception of Majority, where SGP$_{IMS}$ performs best. Nonetheless, in all other cases GP-GOMEA$_{\text{LT}}$ reaches the optimal solution faster and evolves much smaller solutions. This is also reflected when comparing scalability in terms of time: partial evaluations are much more beneficial for the operators of SGP than for GOM (e.g., Figure 2 shows a comparison on how time scalability is affected by partial evaluations in Majority). This is because GOM exchanges multiple nodes at the same time which are not necessarily connected, requiring to re-compute the output of the chain of parent nodes scattered in the solution. Different output caching methods may thus be much more beneficial for GOM (e.g., storing hash-output of the most recurrent subtrees). Moreover, GP-GOMEA typically needs much smaller populations than SGP$_{IMS}$ thanks to the extensive mixing trials performed by GOM, which is also much less prone to bloat. These aspects are very beneficial in terms of memory usage: on difficult problems like Trap-4 some runs of SGP$_{IMS}$ need even 100 times more memory than GP-GOMEA$_{\text{LT}}$.

Another crucial observation from our results comes from the comparison of SGP$_{IMS}$ with SGP$_{param}$. Whereas the former has inherent overhead due to multiple runs with increasing population and tree size, the second sometimes fails to find the optimal solution on complex problems. Although it is arguable that our parameter choice for SGP$_{param}$ is not optimal, some runs converge to a local minimum and are unable to escape it with the sole aid of mutation. Instead, the employment of multiple runs, each starting on random genetic material, dramatically increases the chances of finding the optimal solution. This also explains the success of GP-GOMEA.

The state-of-the-art algorithms ILTI and RDO generally run faster and require less evaluations than GP-GOMEA and SGP. It is worth noticing that these algorithms rely on a fitness function which is *de facto* different: in the semantic fitness, improvements at the level of single fitness cases are sought, whereas in its original specification the fitness is defined as the sum of all correct fitness cases. Therefore, SB needs a decomposition of the fitness to be defined, as well as the design of inverted functions, which compromises the applicability of this powerful technique to general problems. Moreover, whereas SGP$_{param}$ uses sub-optimal parameters, and GP-GOMEA and SGP$_{IMS}$ run with the IMS, ILTI and RDO use the best performing configurations reported. When comparing scalability, ILTI and RDO are superior on Even Parity and Majority, but not on Comparator and Multiplexer. In Comparator, the SB-based operators cause solutions to bloat even worse than the classic operators of SGP, ultimately harming runtime. RDO has particularly poor performance on Multiplexer, being unable to escape from a local minimum within 24 hours in some runs of the 11-bits instance.

Lastly, we observe the effect of IEBL combined with the on average well-performing GP-GOMEA$_{\text{LT}}$, and see that it is either detrimental or very helpful. By encapsulating BBs into new terminal and function nodes, the search space is increased consistently. If such nodes represent partial solutions and can be readily combined into bigger partial solutions, then the search improves. Otherwise, the search is harmed. In the Multiplexer problem, we found that IEBL is not capable of learning any reasonably useful BBs, suggesting that the entropy-based identification method cannot grasp the complex relationships between bits present in this problem. In Comparator, IEBL does catch some relevant relationships: e.g.,

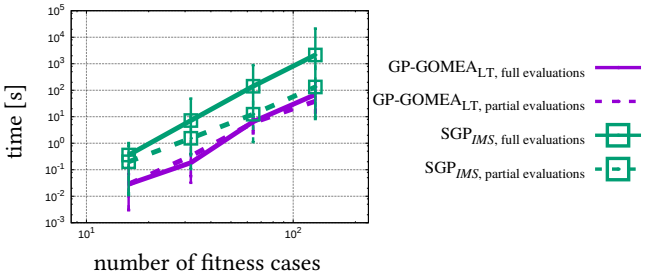

Figure 2: Example of the contribution of partial evaluations in terms of time for the Majority problem.

in the 6-bits instance, which outputs *true* when $b_0b_1b_2 < b_3b_4b_5$, the BB which returns 1 only if $b_i = 0$ and $b_{i+3} = 1$ is often identified. This BB represents the 2-bits Comparator between equally significant bits. As another example, the BB returning 1 when b_1b_2 are set to 1 is learned, which is part of the solution for the case $011 < 100$. However, the expansion of the search space is so big that even learning seemingly-useful material ends up being more detrimental than helpful. Similar considerations hold for the learning of new function nodes. In Majority there is no specific pattern to learn, since no specific relationships between bits are needed: the circuit outputs *true* as long as the majority of bits is set to one. Finally, in Even Parity, IEBL really leads to excellent performance, where in fact partial solutions can be combined to form bigger ones, e.g., two 2-bits Even Parity solutions can be combined with XNOR, which is the function performing a 2-bits Even Parity, to form the 4-bits one. Although the learning of BBs is noisy, resulting in a big performance difference among best and worst runs, the scalability is the best among all algorithms, with the number of evaluations increasing sublinearly with the number of fitness cases. A downside of IEBL is the size of solutions, which may be lowered by implementing mechanisms to prefer shorter BBs. In the plot of the evaluations of Even Parity, we also report the best performance we are aware of, obtained by the Binary Decision Diagrams (BDD) [5]. BDD scales even better than GP-GOMEA$_{LT}$+IEBL in evaluations. However, this EA is specifically designed for Boolean problems, with a dedicated genotype (diagrams assuming a fixed variable ordering of inputs) and particular parameter settings (population size of 5, mutation-only), while GP-GOMEA$_{LT}$+IEBL is a combination of a much more general EA with a Boolean-dedicated mechanism for learning and exploiting BBs. For future work, it would be interesting to attempt to automatically detect when the addition of IEBL is useful so that highly negative effects in the performance of GP-GOMEA are prevented.

6 CONCLUSIONS

In this paper we presented GP-GOMEA, a novel, scalable, model-based approach to GP. Our algorithm requires no parameter specification, which is important for making fair comparisons and for ease-of-use by practitioners. Even though GP-GOMEA is inherently not-tuned, it shows competitive scalability when compared with the latest state-of-the-art algorithms, based on semantic backpropagation. Moreover, while these algorithms need fitness cases and inverted functions to be defined, GP-GOMEA does not have these requirements, making it more generally applicable. Compared to SGP, GP-GOMEA exhibits superior performance on structured problems, while, in general, it evolves much smaller solutions and requires much less memory and time.

We further introduced a novel method to identify and encapsulate useful BBs into new terminals and functions, termed IEBL. The novelty of this method is that it tries to harvest information from the input-space on which fitness cases are defined. The combination of IEBL with GP-GOMEA has been shown to be detrimental in non-modular problems, but extremely efficient on a modular problem like Even Parity. In fact, IEBL helps tackling the complexity of Even Parity to a point where the scalability of GP-GOMEA becomes less than linear, which ultimately leads to solving the 14-bits instance in less than 1 hour.

7 ACKNOWLEDGMENTS

The authors acknowledge the Kinderen Kankervrij foundation for personnel financial support (project #187), and the Maurits and Anna de Kock foundation for financing a high performance computing system. We thank prof. Tomasz P. Pawlak and prof. Krzysztof Krawiec for assisting with reproducing experiments of RDO.

REFERENCES

[1] P. J. Angeline and Jordan B. Pollack. 1994. Coevolving High-Level Representations. In *Artificial Life III*, C. Langton (Ed.). Addison-Wesley, 55–71.
[2] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. 1984. *Classification and regression trees*. CRC press.
[3] Y. P. Chen, T.-L. Yu, K. Sastry, and D. E. Goldberg. 2007. A survey of linkage learning techniques in genetic and evolutionary algorithms. *IlliGAL report* 2007014 (2007).
[4] A. Dessì, A. Giani, and A. Starita. 1999. An Analysis of Automatic Subroutine Discovery in Genetic Programming. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 996–1001.
[5] R. M. Downing. 2005. Evolving binary decision diagrams using implicit neutrality. In *2005 IEEE Congress on Evolutionary Computation*, Vol. 3. IEEE, 2107–2113.
[6] R. Ffrancon and M. Schoenauer. 2015. Memetic semantic genetic programming. In *Proc. of the 2015 Annual Conf. on Genetic and Evol. Comp.* ACM, 1023–1030.
[7] I. Gronau and S. Moran. 2007. Optimal implementations of UPGMA and other common clustering algorithms. *Inform. Process. Lett.* 104, 6 (2007), 205–210.
[8] D. Jackson and A. P. Gibbons. 2007. *Layered Learning in Boolean GP Problems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 148–159.
[9] M. Keijzer, C. Ryan, and M. Cattolico. 2004. *Run Transferable Libraries — Learning Functional Bias in Problem Domains*. Springer, Berlin, 531–542.
[10] E. E. Korkmaz and G. Üçoluk. 2003. Design and usage of a new benchmark problem for genetic programming. In *International Symposium on Computer and Information Sciences*. Springer, 561–567.
[11] J. R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.
[12] L. O. V. B. Oliveira, F. E. B. Otero, G. L. Pappa, and J. Albinati. 2015. *Sequential Symbolic Regression with Genetic Programming*. Springer, 73–90.
[13] T. P. Pawlak, B. Wieloch, and K. Krawiec. 2015. Semantic backpropagation for designing search operators in genetic programming. *IEEE Transactions on Evolutionary Computation* 19, 3 (2015), 326–340.
[14] J. C. Pereira and F. G. Lobo. 2015. Java Implementation of a Parameter-less Evolutionary Portfolio. *CoRR* abs/1506.08867 (2015).
[15] S. C. Roberts, D. Howard, and J. R. Koza. 2001. Evolving Modules in Genetic Programming by Subtree Encapsulation. In *Proceedings of the 4th European Conference on Genetic Programming*. Springer-Verlag, London, UK, 160–175.
[16] K. Sastry and D. E. Goldberg. 2003. Probabilistic model building and competent genetic programming. In *Genetic Prog. Theory and Practice*. Springer, 205–220.
[17] D. Thierens and P. A. N. Bosman. 2013. Hierarchical Problem Solving with the Linkage Tree Genetic Algorithm. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*. ACM, New York, NY, USA, 877–884.
[18] R. A. Watson and T. Jansen. 2007. A Building-block Royal Road Where Crossover is Provably Essential. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*. ACM, New York, NY, USA, 1452–1459.

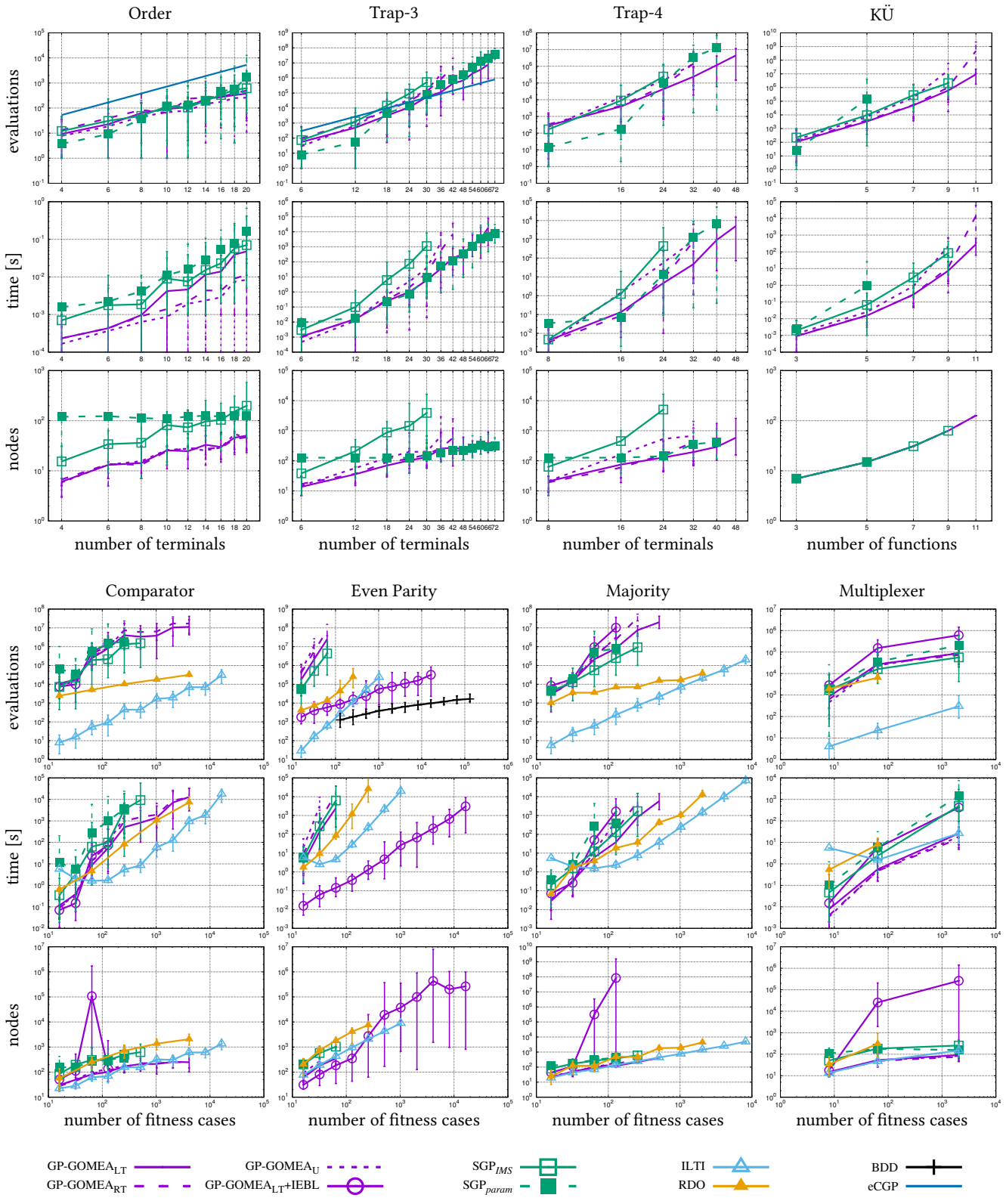

Figure 3: Average, maximum and minimum number of evaluations, time, and number of nodes in the final solution.