Graph representations in genetic programming

Léo Françoso Dal Piccol Sotto¹ · Paul Kaufmann² · Timothy Atkinson³ · Roman Kalkreuth⁴ · Márcio Porto Basgalupp⁵

Published online: 30 September 2021
© The Author(s) 2021

Abstract

Graph representations promise several desirable properties for genetic programming (GP); multiple-output programs, natural representations of code reuse and, in many cases, an innate mechanism for neutral drift. Each graph GP technique provides a program representation, genetic operators and overarching evolutionary algorithm. This makes it difficult to identify the individual causes of empirical differences, both between these methods and in comparison to traditional GP. In this work, we empirically study the behaviour of Cartesian genetic programming (CGP), linear genetic programming (LGP), evolving graphs by graph programming and traditional GP. By fixing some aspects of the configurations, we study the performance of each graph GP method and GP in combination with three different EAs: generational, steady-state and $(1 + \lambda)$. In general, we find that the best choice of representation, genetic operator and evolutionary algorithm depends on the problem domain. Further, we find that graph GP methods can increase search performance on complex real-world regression problems and, particularly in combination with the $(1 + \lambda)$ EA, are significantly better on digital circuit synthesis tasks. We further show that the reuse of intermediate results by tuning LGP's number of registers and CGP's levels back parameter is of utmost importance and contributes significantly to better convergence of an optimization algorithm when solving complex problems that benefit from code reuse.

Keywords Linear genetic programming · Cartesian genetic programming · Evolving graphs by graph programming · Directed acyclic graph

T. Atkinson: Work partially done while at University of Manchester, UK.

✉ Léo Françoso Dal Piccol Sotto
leo.francoso.dal.piccol.sotto@scai-extern.fraunhofer.de

Extended author information available on the last page of the article

1 Introduction

Genetic programming (GP) is a well-known technique for evolving programs and has been successfully applied on several domains, like regression, control problems, and evolution of digital circuits [19, 25]. A program in GP is represented as a tree; however, throughout the years, researchers have proposed alternative representations, as forms of linear encoding and graphs [1, 6, 23, 25].

In Linear Genetic Programming (LGP), programs are represented as lists of instructions of a programming language, and the result of each instruction is assigned to a register from a predefined set of registers [6]. In Cartesian Genetic Programming (CGP), programs are grids of nodes, and each node can use the nodes of the previous layers as arguments [23]. In both of these methods, the genotype is linear, but the phenotype is interpreted as a Directed Acyclic Graph (DAG). Evolving Graphs by Graph Programming (EGGP), manipulates and evolves such graphs directly, without an intermediary encoding [1]. In this work, we use the general term graph GP to refer to any of these three methods.

Several works claim that graph representations provide inherent advantages over trees and demonstrate improved performance of graph GP methods over standard GP [4, 6, 10, 23, 27]. Given that graphs can naturally represent inactive code (code that is not connected to the main program outputs) which can be freely mutated without changing a given program's fitness, some publications show an improved performance of CGP due to neutral genetic drift [22, 30, 32, 36]. Furthermore, graphs present automatic code reuse, that is, the result of a sub-expression can serve as an argument for more than one node, and this can make solutions more compact [6, 10].

However, each graph GP method has its own set of genetic operators and evolutionary algorithm (EA). For instance, LGP uses a steady-state EA, whereas CGP and EGGP use a $(1 + \lambda)$ EA. On the other hand, GP uses a generational EA. As the genetic operators and EAs interact with the representation, it is not fair to claim that the graph representation is the sole cause of empirical differences in performance between these methods and standard GP. To better understand how graph GP methods work and how they can present an advantage over trees, we analyse the impact of three different factors in the performance of the methods: the representation, the genetic operators, and the evolutionary algorithm. To do so, we concentrate on two main research questions: 1) How does the EA used for each graph-based GP technique and standard GP impact the performance of these methods?, and 2) Do graphs present an advantage over trees when the same EA is used?

These questions have been approached by us for the first time in [28]. The current work seeks to extend the scope of the previous results by incorporating some changes in the experimental methodology, doing a more complete analysis of the results, and testing the algorithms on large real-world benchmarks, which leads to more insightful and robust conclusions. In addition to the first two research questions above, this work also tries to understand: 3) Is there a relationship between the frequency of reusing intermediate results in LGP and CGP and the

performance on parity circuits problems? The reuse of intermediate results can be parameterized in LGP by setting the number of registers and in CGP by configuring the levels-back parameter.

In this work, we study four symbolic regression problems, five real-world regression problems, five standard digital circuit synthesis problems and 5 parity problems. We find that standard tree-based GP with a generational EA generally works better for symbolic regression benchmarks, but that graph GP methods can outperform standard GP on real-world regression problems. Further, the $(1 + \lambda)$ EA obtains the best results on every problem for digital circuit synthesis benchmarks. Finally, graph GP methods, particularly in combination with the $(1 + \lambda)$ EA, are significantly more suited to digital circuit synthesis tasks than traditional tree-based GP. In particular, LGP and CGP provide a mechanism for controlling code reuse that is of benefit for solving complex parity problems.

The main methodological goal of this work is understanding the dynamics of some major components of an overall optimization process (optimization algorithm, encoding properties, search space categories) in different algorithms such as GP, LGP, CGP, and EGGP, and search for similar patterns. Similarities can help us in future to unify the different graph-based methods into a single and more general one as well as transfer results and analysis techniques from the research on one algorithm to the others. All algorithms in this work are suboptimally configured, as opposed to what has been done in related work (for example, [31]). We use the convergence performance of an algorithm not to conclude that one algorithm is better than the others, but to conclude that one component of an optimization process, such as for instance the optimization algorithm, fits better to a given configuration (e.g. goal function, encoding, evolutionary operators). Optimal convergence performance of an algorithm is not in the scope of this investigation. We use the convergence as indicator for the dynamics of one of the ingredients of an optimization process and not to establish an absolute ranking between GP, LGP, CGP, and EGGP. For instance, CGP performs much better for all Boolean and synthetic regression benchmarks than shown in this paper [15, 16]. However, it's not our goal to compare peak performances of GP algorithms, but to compare their inner mechanisms under a controlled study.

The rest of this work is arranged as follows. Section 2 presents a summary of GP, LGP, CGP, and EGGP, as well as a discussion on the differences between these techniques. Section 3 explains the experimental design used, and Sect. 4 shows the results obtained and a discussion. The paper is concluded in Sect. 5.

2 Background

In this section, we describe and contrast the methods that are considered in this work: GP, LGP, CGP, and EGGP.

Fig. 1 LGP program representing an 1-bit adder. Instruction 4 is an example of an inactive instruction

```

1: r[3] = r[0] XOR r[1]
2: r[4] = r[0] AND r[1]
3: r[5] = r[2] AND r[3]
4: r[6] = r[3] AND 1.0
5: r[0] = r[5] OR r[4]
6: r[1] = r[2] XOR r[3]

```

2.1 Genetic programming

Standard GP represents programs in a tree-based scheme [19], and uses two standard genetic operators: crossover and mutation. In tree-based encoding schemes (GP), crossover is given by swapping randomly chosen subtrees from two parents. The conventional approach to mutation is to replace a randomly selected individual's subtree by a randomly generated one (also known as subtree mutation).

GP usually makes use of the generational EA: individuals from the current population are selected via tournament selection and go through crossover and mutation, according to pre-defined probabilities. The process is repeated until there is a new population formed by the offspring of the current individuals, and the best individual from the current generation is passed onto the next one (elitism).

2.2 Linear genetic programming

LGP represents programs as lists of instructions from a programming language. It uses a registers vector r , where each position is initialized with the values of the input variables, in a sequential and cyclic manner [6]. For instance, if a system has eight registers and four input variables, the four first registers are initialized with the four inputs, and so are the four last registers. An instruction is encoded in the form $(func, dest, args)$, where $func$ is a function (for instance, a logical function), $dest$ is the index of the destination register (where the outcome of the function is stored), $args$ are the arguments. An argument can be either a register index or a constant. In LGP, the second argument of a binary function generally has a probability of 50% of being a constant from a predefined range.

Figure 1 shows an LGP program for an 1-bit binary adder. The adder takes three inputs: two bits in registers $r[0]$ and $r[1]$ and a carry-in in register $r[2]$. The instructions are interpreted and executed from the first to the last. Note that registers can be reused and overwritten. The final result is stored in registers $r[0]$ (carry-out) and $r[1]$ (sum).

LGP uses mainly macro and micro-mutations [5, 6]. A macro-mutation acts on the program level and can remove, insert, or replace an entire instruction. Micro-mutations act on the instruction level and change the function of an instruction, the destination register, or the argument register. Micro-mutations are equivalent to point mutations in CGP (see Sect. 2.3).

Fig. 2 Cartesian genetic program (top) and its encoding (bottom). All nodes contributing to primary outputs are called “active” and all other nodes “inactive”. For example, node 6 is an inactive node

Traditionally, a steady-state EA is used in LGP [6]. In this EA, two winners and two losers are chosen via tournament selection. Copies of the winners replace the losers in place, and a macro and a micro-mutation are applied to the winners according to user-defined probabilities. A generation is defined when P individuals have been processed, P being the population size.

2.3 Cartesian genetic programming

Cartesian genetic programming (CGP) is an evolutionary algorithm that uses DAGs to encode solution functions [23]. CGP’s genotype is a linear list of $(n_a + 1)$ -tuples of integers describing for every node of a graph the routing of node’s n_a inputs and node’s function. To ensure that the encoded graph is cycle-free, CGP uses an intermediate representation to restrict the routing. For this, CGP places the nodes on a $n_c \times n_r$ grid and requires that the connection between the nodes are always going in the same direction (eg. from left to right). The maximal number of columns a connection may span is called the levels back parameter l . The n_i primary inputs and n_o primary outputs of the graph are separate node sets treated as the left-most and right-most grid columns. The order of mapping the sequence of genotype tuples to graph nodes on the grid is from the top to bottom and from the left to the right, as shown in Fig. 2.

CGP uses point mutation exclusively. A function gene is mutated by replacing it with a randomly selected function index from the functions’ table. A connection gene is mutated by rewiring a node’s input to a randomly selected node in the previous l columns (ie. some node in l columns on the left of the currently mutated node). This ensures the cycle-free condition. The size of CGP’s genotype can be reduced, and in consequence, the convergence improved by setting the number of rows to $n_r = 1$. Related works almost exclusively use this “single-line” CGP model. Additionally, the $(1 + \lambda)$ EA with $\lambda=4$ is predominantly employed to optimize CGP. In the $(1 + \lambda)$ EA, there is only one individual and, each generation, λ off-springs are generated via the mutation operator. The selection scheme

Fig. 3 EGGP program representation, figure taken from [4]. Nodes i_1 and i_2 are input nodes. Node o_1 is an output node. An edge from x to y means that x uses y as an argument. Edges are explicitly ordered by their integer labels (see [4] and inactive material is annotated gray

is implemented such that off-spring individuals that are as fit as or better than the parent are preferred when selecting the new parent for the next generation.

CGP shares with LGP the property that not all graph nodes are contributing to primary outputs. A CGP phenotype is therefore only a small part of the intermediate grid graph, as showed in Fig. 2. CGP additionally is biased towards evolution of small solutions. This limits the tendency for bloat, as showed in [13].

2.4 Evolving graphs by graph programming

Evolving graphs by graph programming (EGGP) is an approach to graph GP where programs are represented directly as DAGs [1]. Each solution consists of input nodes, function nodes and output nodes, which correspond directly to the input nodes, inner nodes and output nodes of CGP respectively. In conventional EGGP, the only restrictions made on the topology of a program are that it may not contain cycles, although additional controls for program depth may be introduced if desired [4]. We give an example of the EGGP program representation in Fig. 3.

Genetic operators in EGGP are described through probabilistic graph programs [2], which are a programmatic extension to formal graph transformation (see [7]). While probabilistic graph programs may be used to describe domain specific mutation operators [3] and recombination operators [4], the original form of EGGP [1] that we study here provides two atomic mutation operators:

- *Edge mutation* An edge is chosen to mutate at random. The set of valid nodes that may be targeted without introducing a cycle is identified. The mutating edge is then redirected to target one of these nodes chosen at random.
- *Node mutation* A function node is chosen to mutate at random and its function is changed to some other function from the function set. If the arity of the function node has increased, new edges are inserted while preserving acyclicity. If the arity of the function node has decreased, edges are deleted at random. Finally, the ordering of the mutated node's edges is randomised.

Table 1 The main aspects of GP and the three graph GP techniques considered here

Technique	Representation	Operators	EA
GP	Tree	Crossover, subtree and point mutation	Generational
LGP	List of instructions (Genotype) / DAG (Phenotype)	Macro and micro-mutations	Steady-state
CGP	List of nodes (Genotype) / Grid, DAG (Phenotype)	Point mutation	$(1 + \lambda)$ EA
EGGP	DAG	Point mutation	$(1 + \lambda)$ EA

Given the mutation rate m_r and an individual with v_f function nodes and e edges, a number of node mutations $m_v \in \mathcal{B}(v_f, m_r)$ and edge mutations $m_e \in \mathcal{B}(e, m_r)$ are sampled. All $m_v + m_e$ mutations are then placed in a list which is then shuffled, applying mutations in a random order. The overall expected number of mutations is $m_r(v_f + e)$.

Although other EAs have been investigated [4], EGGP has in general assumed CGP's standard evolutionary algorithm [1, 3]; the $(1 + \lambda)$ EA with $\lambda = 4$ and neutral drift enabled.

2.5 Comparison between representations

Table 1 summarizes the main aspects of GP, LGP, CGP, and EGGP. The difference between LGP and CGP with respect to representation is that in LGP the previous instructions' results that can be used as arguments by the current instruction are defined in terms of the number of registers available, whereas in CGP the previous nodes' results that can be used as arguments by the current node are defined by the levels-back parameter. In EGGP, all nodes can be used as arguments by a given node, except when it results in a cycle. By setting $n_r = 1$, $n_c = N$, and $l = n_c$, for CGP, where n_r is the number of rows, n_c the number of columns, N the number of nodes, and l the levels-back parameters, then CGP can represent the same set of programs as EGGP. Also, if a separate vector for inputs and a unique register for each instruction is used for LGP, then it can also represent the same set of programs as EGGP.

Regarding the genetic operators, mutations are preferred for all graph GP variations [1, 5, 6, 21]. CGP and EGGP both use fixed-size programs, and mutations change only the function of each node or connections between nodes. In LGP, however, the macro-mutation can insert or delete instructions. Thus, programs begin with an initial length and are allowed to grow until a maximum size. Micro-mutations are equivalent to CGP point mutations, however, in EGGP, some mutations that are not allowed in CGP and LGP can occur. An example of such a mutation is shown in Fig. 4. Here, the red edge (that goes from node 2 to 1) is redirected to go from node 2 to 3 (blue edge). As CGP and LGP can only use previous nodes/instructions as arguments of the current node (feed-forward property), this mutation is impossible. However, as no cycle is created, it

Fig. 4 Example of mutation that is allowed in EGGP but not in CGP and LGP. i is an input and o an output. An edge from x to y means that x uses y as argument. In this mutation, the red edge is replaced by the blue one. From [1]

is possible in EGGP. In [1], it was demonstrated that this difference results in a performance gain for EGGP in comparison to CGP on digital circuit benchmarks.

There are publications that compare some graph GP variants with standard GP and with each other. In [6], LGP was able to outperform GP on symbolic regression, digital circuits synthesis, and classification tasks. In [10], LGP produced better programs than GP for classification benchmarks, both in terms of performance and understandability. LGP also outperformed GP on the Ant Trail problem in [27]. In [23], Miller and Thomson showed that CGP outperforms GP on the Ant Trail problem, and that neutral mutations play an important role in this result. CGP's $(1 + 4)$ selection scheme has been shown to outperform generational selection schemes for the evolution of Boolean circuits in [18]. Atkinson et al show in [1] that EGGP obtains better performance than CGP on digital circuits benchmarks, due to its mutation operator. [35] and [14] also made comparisons between LGP and CGP, and LGP and GP, respectively, but found that no method was clearly better than the other on all problems. Schmidt and Lipson [26] compare the tree encoding and a general graph encoding similar to LGP on a number of increasingly complex symbolic regression functions and conclude that the graph encoding produces similar results to trees with less bloat and better computational performance as it is not dependent on recursion.

Although these publications offer some comparison between the methods, as we can see in Table 1, many aspects differ from one algorithm to another. Our goal in this work is to study the role of the EA and the genetic operators that are used in combination with each representation, in order to investigate how graphs and trees can be better utilized. We also want to analyze the influence of the representation when the same EA is used, in order to identify if the structure alone is capable of outperforming trees, or if it is only able to do that when combined with a specific EA.

3 Experimental design

In this section, we present the methodology of our experiments, algorithm configurations, and the benchmark problems that we study.

3.1 Proposed methodology

The goal of our experiments is to use the base algorithms with uniform configurations in order to isolate the effect of the representation, operators, or EA. For that, we consider GP, LGP, CGP, and EGGP in their standard forms, that is, using the basic genetic operators and the standard evolutionary algorithms. We employ the parameter-free single active mutation (SAM) scheme for graph-based approaches [12]. The SAM scheme applies the original point mutation operator repeatedly until an active gene is mutated for the first time. A mutation of an active gene usually changes the phenotype and impacts on the functional quality of a candidate solution. We evaluate the fitness of an individual only on changes of active genes. A fitness of an individual is never re-evaluated in our algorithms if the phenotype-coding genes experience no changes. The rationale behind using the SAM scheme is the minimization of algorithmic parameters that have to be tuned and a fair comparison among the methods.

For the stopping condition for our algorithms we use the number of evaluated graph nodes instead of fitness evaluations. Counting the number of evaluated graph nodes is more accurate because it corresponds to the simulation time of an evolved program on a standard single-threaded processor. It is important to note that only active nodes in graph-based methods are evaluated. Additionally, because the number of fitness test cases is constant for a benchmark, the number of evaluated graph nodes is always a multiple of the number of test cases. For symbolic regression benchmarks, the fitness is computed for twenty points. For Boolean circuit benchmarks, the number of test cases is two to the power of the number of inputs. To simplify reporting, we show only the number of evaluated graph nodes divided by the number of test cases. This effectively corresponds to reporting the accumulated phenotype sizes. We divide our experimental design in the following manner:

1. In our first experiment, we test each of the methods using each of the three EAs described: generational, steady-state, and $(1 + \lambda)$. The goal of this is to study which EA performs best for each algorithm and benchmark class.
2. Second, we compare the performance of the three graph GP methods (LGP, CGP, and EGGP) when the same EA is used, in order to assess the impact of the combination of structure and operators. The difference being considered between CGP and EGGP here is the mutation operator, which is more general in EGGP. LGP, on the other hand, presents more differences: there is no unique identifier for each instruction, registers and inputs can be overwritten, and it uses macro-mutations that add and remove entire instructions. For this reason, we consider also a version referred to as *LGP-micro*, where only micro-mutations are allowed. In LGP-micro, the difference to CGP lies only in the representation.
3. We then select the graph GP method that works best for each of the problems and compare it with standard GP when the same EA is used, in order to assess the impact of the representation.

Table 2 Parameters for GP, LGP, LGP-micro, CGP, and EGGP. When a parameter or value is valid only for some techniques or EA, it is indicated between brackets. The parameters are not tuned but configured according to common values used in related works (e.g. [4, 6, 31])

Parameter	Value
Population Size	1000 (generational and steady-state) / $1+4(1+\lambda)$
Termination condition	Finding perfect solution, or accumulated phenotype size $\geq 10,000,000$
Perfect Solution	MAE $\leq 1e-6$ for regression benchmarks 100% correct output bits for Boolean benchmarks
Tournament Size	10 (generational and steady-state)
Initial program length	20 (LGP)
Maximum program length	100 (LGP)
Fixed program length	100 (LGP-micro, CGP, and EGGP)
Initial tree depth	1-3 (GP)
Maximum tree depth	7 (GP)
Tree initialization	Ramped half-and-half (GP)
Percentage of constants	50% (Regression)
Constants	1 (Regression)
#Registers	#Inputs + 10 (LGP)
#Rows	1 (CGP)
#Columns	#Nodes (CGP)
Levels-back (l)	#Columns (CGP)
Mutation rate	GP, LGP: see Table 3 LGP-micro, CGP, EGGP: single-active mutation

3.2 Algorithm parameters

Table 2 shows the parameters used for each algorithm. We set the population sizes to well-established values. An algorithm terminates if it has found a solution with a MAE below some threshold (symbolic regression benchmarks), if it has evolved 100% correct output bits (Boolean benchmarks), or if it has evaluated candidate solutions with accumulated phenotype sizes (active nodes) equal or above some limit. The fitness of a candidate solution is never reevaluated if its active genes remain unchanged. The tournament size was based on the literature [6] and also confirmed empirically by preliminary runs with different tournament sizes. The initial and maximum program lengths, as well as the fixed length, were based on the literature [1, 5, 6, 21]. We also set the maximum tree depth so that the number of internal nodes is similar to the genotype length in LGP, CGP, and EGGP. The tree initialization method and percentage of constants is the standard for GP [25], and the number of registers allowed for LGP is suggested in [6] and also defined empirically. We avoid using constants for the digital circuits and use only one constant for the regression problems, for all methods. The number of rows, columns, and the levels-back parameter in CGP were defined so that it represents the same set of programs as EGGP. For GP, we have adapted the implementation

Table 3 Mutation rates used by each method and benchmark class after optimization

Technique	Regression			Circuits		
	Gen	Steady	$1 + \lambda$	Gen	Steady	$1 + \lambda$
GP	1.0%	2.0%	6.0%	–	–	–
LGP	30.0%	30.0%	30.0%	2.0%	2.0%	2.0%

Gen refers to the generational EA and steady to steady-state

‘–’ means that a mutation rate was not used for the given configuration

from DEAP [11], in Python, while for LGP, CGP, and EGGP, we have used our implementations.¹

For LGP-micro, CGP, and EGGP, we employ the single-active mutation. However, for GP this is not possible as there is no inactive code and crossover is used, and LGP still has macro-mutations that can remove or add entire instructions. In preliminary runs, we have observed that using a mutation rate was beneficial for LGP and GP (only for the regression functions in GP), and have adopted it. In LGP, a mutation rate of $X\%$ means that $X\%$ of the instructions undergo a macro-mutation followed by a micro-mutation. In GP, it means that a subtree mutation is applied to $X\%$ of the nodes. Crossover is still used with a probability of 90%, as we found it to be important for GP in preliminary runs comparing GP with and without crossover. In the digital circuits, we use a probability of 10% for the subtree mutation in GP.

The mutation rate was optimized using *nguyen5* for regression and *adder2* for the digital benchmark classes (see Sect. 3.3 for benchmarks used). We performed 50 runs varying the rate between 1% and 30%, and chose the one that performed best for each method.² The mutation rates used are shown in Table 3.

All remaining parameters have been configured according to commonly used values in related works [1, 4, 6, 28, 31], and sometimes confirmed by preliminary runs with different parameter values. All algorithms would perform better if their configurations would be optimized for some benchmark [15, 16]. However, as performance comparison between differently configured algorithms optimized to specific problems is not a subject investigated in this paper, we set the parameters to general values, so that we can better isolate and investigate the role of the specific features (evolutionary algorithm, genetic operators, and representation).

3.3 Benchmarks

We use three different classes of benchmarks, which were chosen according to suggestions made in the literature for GP benchmarks [20, 24, 34]: symbolic regression, real-world regression data, and digital circuits. For symbolic regression, we have

¹ We have made our implementations of LGP, CGP, and EGGP, respectively, available at <https://github.com/leo-sotto/LGP>, <https://github.com/paul-kaufmann/cgp>, and <https://github.com/timothyatkinson/GraphComparison>.

² As explained in Sect. 3.3, no testing set is used for the digital circuit benchmarks, as the aim here is to find a circuit that fits to all combinations of input bits. For *nguyen5*, we used different points drawn from the same range as specified in [20] for the training set as a validation set to tune the mutation rates.

Table 4 Sizes of the training and test data sets in the regression experiments. For the *pagiel* and *nguyen* functions, definitions of training and test sets were taken from [20]. As no testing set is defined for these functions, for *pagiel* we have used the same interval from -5 to 5 but with a step of 0.1 instead of 0.4 for the test set, and for the *nguyen* functions we have sampled 20 *different* points from the same interval as for the training set

Benchmark	–Training set–	–Test set–
<i>pagiel</i>	25	100
<i>nguyen</i> {*}	20	20
<i>airfoil</i>	1052	451
<i>concrete</i>	721	309
<i>energyCooling</i>	537	231
<i>energyHeating</i>	537	231
<i>yacht</i>	215	93

used the functions *pagiel*, *nguyen3*, *nguyen5*, and *nguyen7* (definitions from [20], and for the digital circuits, $1 \times 1 \times c_{in}$ adder (*adder1*), $2 \times 2 \times c_{in}$ adder (*adder2*), $3 \times 3 \times c_{in}$ adder (*adder3*), 2×2 multiplier *mult2*, and 3×3 multiplier *mult3* (definitions from [33]). All adders implement the carry-in line. As the circuits used have more than one output, and solving problems with multiple outputs is not trivial for GP, in order to compare the results with GP we have used parity functions with only one output: 3-bit input even parity (*par3*), 4-bit input even parity (*par4*), 5-bit input even parity (*par5*), 6-bit input even parity (*par6*), and 7-bit input even parity (*par7*) (definitions from [33]).

The real-world regression datasets used can be found in the UCI machine learning repository [9]: the *airfoil* dataset with 5 inputs and 1,503 instances, the *concrete* dataset with 8 inputs and 1,030 instances, the *energyCooling* and *energyHeating* datasets both with 8 inputs and 768 instances, and the *yacht* dataset with 6 inputs and 308 instances. All datasets have only one numerical output. We have split the data into 70% for training and 30% for testing in a stratified manner. The number of training and test samples are summarized in Table 4.

The function set for symbolic regression benchmarks was: +, −, *, /, *sin*, *cos*, *e*, *ln* [20] (protected operators return 1.0). For the adder and parity circuits, it was AND, NAND, OR, NOR, and for the multiplier circuits AND, AND with one input inverted, XOR, OR [33]. We use median absolute error (MAE) as a fitness function for regression and the percentage of correct bits for the circuits. For the digital circuits, we additionally present Koza’s computational effort (CE) [19] values, with $z = 0.99$, which serves as an estimate of how many evaluations a method needs in order to find the solution for a given problem with 99% of success.

Table 5 MAE for all techniques using different EAs evaluated on the testing set for the symbolic regression benchmarks and real-world datasets

Method	EA	pagel	nguyen3	nguyen5	nguyen7	airfoil	concrete	cooling	heating	yacht
GP	Gen	0.117	7.63e-17	0.000	0.003	5.509	7.095	1.863	1.412	1.179
	Steady	0.127	0.008	0.000	0.004	5.896	7.464	1.862	1.534	1.188
LGP	1 + 4	0.109	0.017	0.000	0.008	9.584	7.519	2.338	2.026	1.472
	Gen	0.128	0.046	1.49e-08	0.016	15.551	12.249	2.789	2.684	3.283
LGP-micro	Steady	0.099	8.19e-08	1.49e-08	0.010	10.044	10.497	2.556	2.212	2.047
	1 + 4	0.093	0.032	1.49e-08	0.011	7.252	10.957	2.555	2.270	1.669
CGP	Gen	0.111	0.014	0.004	0.005	5.134	7.606	1.686	1.387	1.101
	Steady	0.115	0.021	0.009	0.006	5.220	8.043	1.859	1.429	1.275
EGGP	1 + 4	0.097	0.024	0.014	0.007	4.960	7.678	1.561	1.262	1.152
	Gen	0.087	0.039	0.008	0.010	3.416	6.662	1.705	1.335	0.939
EGGP	Steady	0.100	0.054	0.010	0.013	4.058	6.684	1.870	1.554	1.000
	1 + 4	0.084	0.041	0.004	0.009	4.099	6.938	2.021	1.696	0.878
EGGP	Gen	0.119	0.079	0.008	0.009	3.548	6.656	1.639	1.281	0.905
	Steady	0.110	0.072	0.008	0.016	4.086	7.042	1.733	1.322	1.007
EGGP	1 + 4	0.062	0.035	0.003	0.006	4.304	7.625	2.054	1.699	0.763

The best result for each problem is shown in boldface

MAE values are a median over 100 independent consecutive runs

Table 6 Percentage of correct bits and minimum computational effort (CE) divided by 1,000 for all techniques using different EAs evaluated on the digital circuit benchmarks. Percentage values are a median over 100 independent consecutive runs, and the CE is only computed if at least 10 runs were successful

Method	EA	adder1		adder2		adder3		mult2		mult3	
		Perc	CE	Perc	CE	Perc	CE	Perc	CE	Perc	CE
GP	Gen	–	–	–	–	–	–	–	–	–	–
	Steady	–	–	–	–	–	–	–	–	–	–
	1 + 4	–	–	–	–	–	–	–	–	–	–
LGP	Gen	1.00	536	0.96	6,678	0.87	–	1.00	195	0.94	–
	Steady	1.00	351	0.96	13,965	0.87	–	1.00	152	0.93	–
	1 + 4	1.00	44	1.00	504	0.94	13,364	1.00	69	0.93	–
LGP-micro	Gen	1.00	256	0.95	11,520	0.80	–	1.00	770	0.88	–
	Steady	1.00	216	0.92	21,204	0.81	–	0.98	1,261	0.88	–
	1 + 4	1.00	41	1.00	245	0.98	1,900	1.00	58	0.95	–
CGP	Gen	1.00	74	0.96	3,736	0.88	–	1.00	81	0.95	–
	Steady	1.00	108	0.96	3,758	0.87	–	1.00	116	0.95	–
	1 + 4	1.00	21	1.00	278	1.00	1,515	1.00	20	0.99	5,999
EGGP	Gen	1.00	101	0.94	2,643	0.89	–	1.00	131	0.94	–
	Steady	1.00	79	0.94	2,944	0.89	–	1.00	136	0.94	–
	1 + 4	1.00	28	1.00	186	1.00	917	1.00	25	0.99	5,793
Method	EA	par3		par4		par5		par6		par7	
		Perc	CE	Perc	CE	Perc	CE	Perc	CE	Perc	CE
GP	Gen	1.00	63	0.94	832	0.81	–	0.72	–	0.63	–
	Steady	1.00	78	0.94	1,824	0.81	–	0.72	–	0.63	–
	1 + 4	1.00	5	1.00	75	0.91	1,840	0.77	17,963	0.69	–
LGP	Gen	1.00	164	1.00	552	1.00	2,450	0.92	7,964	0.84	32,657
	Steady	1.00	173	1.00	698	1.00	1,725	0.94	7,650	0.84	55,539
	1 + 4	1.00	13	1.00	46	1.00	119	1.00	348	1.00	827
LGP-micro	Gen	1.00	26	1.00	160	1.00	672	0.94	2,828	0.87	13,696
	Steady	1.00	21	1.00	216	1.00	740	0.94	4,400	0.86	23,712
	1 + 4	1.00	5	1.00	21	1.00	33	1.00	82	1.00	260
CGP	Gen	1.00	28	1.00	454	0.94	2,587	0.84	–	0.75	–
	Steady	1.00	29	1.00	267	0.94	3,939	0.86	–	0.74	–
	1 + 4	1.00	8	1.00	47	1.00	206	1.00	609	0.98	2,791
	Gen	1.00	32	1.00	311	0.91	2,260	0.83	–	0.73	–
	Steady	1.00	27	1.00	654	0.91	2,152	0.83	–	0.74	–
	1 + 4	1.00	10	1.00	31	1.00	62	1.00	111	1.00	181

4 Results and discussion

Table 5 shows the results in terms of MAE for all methods using the generational, steady-state, and $(1 + \lambda)$ EAs on the regression benchmarks. Table 6 shows the percentage of correct bits and Computational Effort for all techniques on the digital circuits. As GP is only defined for 1-output problems, we did not run it for the adder and multiplier circuits.

4.1 Comparison between evolutionary algorithms

Based on Table 5, the following observations can be made for solving regression problems:

- The tendencies in the results for *pagie1*, *nguyen*, and real-world benchmarks are different and can be better analysed separately.
- The $(1 + \lambda)$ EA is consistently the best scheme among all optimization algorithms for the *pagie1* benchmark. While EGGP excel, the differences between remaining algorithms and evolutionary schemes are rather small.
- For the *nguyen3* and *nguyen5* benchmarks, generational GP and steady-state LGP are better than the remaining optimization algorithms.
- For the *nguyen7* benchmark, results among the optimization algorithms and evolutionary schemes are similar. GP and LGP-micro perform best, regardless of the evolutionary scheme and the remaining optimization algorithms follow closely.
- For the *nguyen* benchmarks, the generational EA works best for GP and LGP-micro, while the steady-state EA works best for LGP and the $(1 + \lambda)$ EA for CGP as well as EGGP.
- For the real-world datasets, the generational algorithm worked best for GP, CGP, and EGGP, the only exception being the dataset *yacht* for CGP and EGGP. For LGP and LGP-micro, however, the $(1 + \lambda)$ EA worked better, but the difference in comparison to the other EAs was small for LGP-micro.

For the Boolean benchmarks in Table 6, the following observations can be made:

- The $(1 + \lambda)$ EA is consistently and by far the best evolutionary scheme for all optimization algorithms and benchmarks.
- The generational and steady-state EAs present similar performances and do not scale well on the even-parity benchmarks.

We show in Table 7 the results of statistical comparisons between the generational, steady-state, and $(1 + \lambda)$ EAs for all methods. For each pair of EAs and benchmark category, we show the mean ranking for the EAs and the p -value resulting from a Friedman test, following the approach in [8] for comparison of multiple algorithms on multiple datasets.

Table 7 Rankings of EAs divided by problem class and method. The lower the rank, the better the EA. “Regression 1” refers to the benchmark functions and 2 refers to the real-world datasets, while “Circuits 1” refers to the adder and multiplier circuits and 2 refers to the parity circuits. For each category, we calculated the mean rank of each EA over the problems in that category (for “Regression 1”, for example, the value shown is a mean rank over functions *pagie1*, *nguyen3*, *nguyen5*, and *nguyen7*). Ranks in bold-face are the best ranks for each category. We also show the *p*-value resulting from a Friedman test

Method	EAs	Regression 1	Regression 2	Circuits 1	Circuits 2
GP	Generational	1.5	1.2	-	2.3
	Steady-state	2.5	1.8	-	2.5
	(1 + 4)	2.5	3.0	-	1.2
	Friedman <i>p</i> -value	0.47	0.01	-	0.08
LGP	Generational	2.75	3.0	2.1	2.3
	Steady-state	1.5	1.6	2.3	2.1
	(1 + 4)	1.75	1.4	1.6	1.6
	Friedman <i>p</i> -value	0.17	0.02	0.52	0.52
LGP-micro	Generational	1.25	1.6	2.3	2.1
	Steady-state	2.25	3.0	2.4	2.3
	(1 + 4)	2.5	1.4	1.3	1.6
	Friedman <i>p</i> -value	0.17	0.02	0.15	0.52
CGP	Generational	1.75	1.2	2.3	2.3
	Steady-state	3.0	2.2	2.3	2.3
	(1 + 4)	1.25	2.6	1.4	1.4
	Friedman <i>p</i> -value	0.03	0.07	0.25	0.25
EGGP	Generational	2.5	1.2	2.1	2.3
	Steady-state	2.5	2.2	2.5	2.3
	(1 + 4)	1.0	2.6	1.4	1.4
	Friedman <i>p</i> -value	0.04	0.07	0.21	0.25

The rankings confirm our observations: for the symbolic regression regression problems, the generational EA worked best for GP and LGP-micro, the steady-stated EA for LGP, and the $(1 + \lambda)$ EA for CGP and EGGP. For the real-world regression datasets, the generational EA worked best for GP, CGP, and EGGP, but the $(1 + \lambda)$ EA was the best for LGP and LGP-micro. For evolving digital circuits, the generational and steady-state EAs are similarly ranked, and the $(1 + \lambda)$ EA has the best rank for all combinations of algorithms and problem instances.

Most *p*-values are greater than 0.05, and this are not statistically significant. The exceptions are CGP and EGGP on the symbolic regression functions ($(1 + \lambda)$ with the best rank), and GP, LGP, and LGP-micro on the real-world regression datasets (generational with the best rank for GP and $(1 + \lambda)$ for LGP and LGP-micro). For regression, this outcome was expected, as results are sometimes mixed and vary between problem instances. For digital circuits, the three different EAs (generational, steady-state, and $(1 + \lambda)$) perform similarly in terms of percentage of correct bits for simpler circuits, but differ when we look at the CE. For example, LGP-micro achieves a performance of 1.0 for all EAs for functions *par3*, 4, and 5, but the CEs

Table 8 Selected statistical comparison between EAs for regression benchmarks

Method	EAs	pagel	nguyen3	nguyen5	nguyen7	airfoil	concrete	energyCooling	energyHeating	yacht
GP	Gen vs (1 + 4)	–	0.74	0.58	0.75	0.73	0.64	0.94	0.93	0.67
LGP-micro	Gen vs (1 + 4)	–	0.67	0.74	0.59	0.63	–	0.64	0.60	–
CGP	Gen vs (1 + 4)	–	–	0.69	–	0.75	–	0.75	0.78	–
EGGP	Gen vs (1 + 4)	0.70	0.64	0.81	0.67	0.64	0.72	0.78	0.77	0.65

For each pair of methods where the p -value of a Mann–Whitney U test is less than α/N , $\alpha = 0.05$, where N is the number of benchmarks for the given class (Bonferroni correction), we show the Vargha and Delaney A measure for assessing the effect size. Large effects are marked in boldface (> 0.70)

Table 9 Selected statistical comparison between EAs for digital circuits benchmarks

Method	EAs	add1	add2	add3	mult2	mult3	par3	par4	par5	par6	par7
GP	Gen vs(1 + 4)	–	–	–	–	–	–	0.80	0.85	0.79	0.86
LGP-micro	Gen vs (1 + 4)	0.57	0.96	0.99	0.66	0.99	–	–	0.64	0.85	0.96
CGP	Gen vs (1 + 4)	0.52	0.87	0.99	–	0.99	–	0.61	0.80	0.96	0.97
EGGP	Gen vs (1 + 4)	0.57	0.89	0.97	0.53	0.97	0.52	0.68	0.88	0.96	1.00

For each pair of methods where the p -value of a Mann Whitney U test is less than α/N , $\alpha = 0.05$, where N is the number of benchmarks for the given class (Bonferroni correction), we show the Vargha and Delaney A measure for assessing the effect size. Large effects are marked in boldface (> 0.70)

for the $(1 + \lambda)$ EA are much lower (Table 6). Even when the results differ, the difference is not always extremely large (for example, CGP and EGGP on *mult3* in Table 6), although there is a clear difference if we look at the CE.

We show in Tables 8 and 9 a statistical comparison of selected methods on each individual problem based on a Mann-Whitney U test and the Vargha and Delaney A measure, in order to assess possible statistical differences that were not captured by the Friedman test. We focus here on a comparison between the generational and the $(1 + \lambda)$ EAs, as the generational EA worked best for regression in some cases, while the $(1 + \lambda)$ EA worked best in other cases, and clearly produced the best results for all digital circuits problems.

From Table 8, we confirm that, on individual problems, the generational EA statistically outperforms the $(1 + \lambda)$ EA for GP and LGP-micro, with some large effect sizes. Whereas for CGP the differences are not significant on the symbolic regression functions, for EGGP the $(1 + \lambda)$ EA is statistically better than the generational EA on all problems, with mostly moderate effect sizes. For the real-world regression datasets, on the other hand, CGP and EGGP under the generational EA outperform the $(1 + \lambda)$ EA with large effect sizes. From Table 9, it is clear that the improvement of the $(1 + \lambda)$ EA over the generational EA is statistically significant for all methods on almost all problems, with many large effect sizes.

Based on these results, we can say that the results for the regression problem class are more mixed and dependent on the combination of the optimization algorithm and a problem instance. For the digital circuits, however, results fully support that the use of the $(1 + \lambda)$ EA causes a significant improvement in performance for this benchmark class, regardless of the representation being used, which suggests that solutions to these benchmark problems benefit from intensive exploitation. Similar conclusions have been observed by Kaufmann and Kalkreuth in their parameter studies [15, 16]. Increased exploitation by reducing $\lambda \rightarrow 1$ achieved best convergence rates over a wide range of Boolean benchmarks.

4.2 Comparison between graph-based GP methods

In this section, we focus on the comparison between LGP, LGP-micro, CGP, and EGGP when the same evolutionary algorithm is used. From Table 5, we make

Table 10 Rankings of graph GP methods divided by problem class and EA. The lower the rank, the better the EA. “Regression 1” refers to the benchmark functions and 2 refers to the real-world datasets, while “Circuits 1” refers to the adder and multiplier circuits and 2 refers to the parity circuits. For each category, we calculated the mean rank of each method over the problems in that category (for “Regression 1”, for example, the value shown is a mean rank over functions *pagie1*, *nguyen3*, *nguyen5*, and *nguyen7*). Ranks in boldface are the best ranks for each category. We also show the *p*-value resulting from a Friedman test

EA	Methods	Regression 1	Regression 2	Circuits 1	Circuits 2
Generational	LGP	3.0	4.0	2.2	2.1
	LGP-micro	1.5	2.8	3.4	1.7
	CGP	2.25	2.0	2.0	2.8
	EGGP	3.25	1.2	2.4	3.4
	Friedman <i>p</i> -value	0.21	0.005	0.32	0.16
1 + 4	LGP	2.5	4.0	3.1	2.4
	LGP-micro	2.75	2.2	2.7	2.4
	CGP	3.0	1.6	2.0	2.8
	EGGP	1.75	2.2	2.2	2.4
	Friedman <i>p</i> -value	0.55	0.02	0.52	0.98

the following observations for the comparison of the graph-based methods on the regression problems:

- When the generational EA is used, LGP-micro has the best performance for the symbolic regression functions, whereas CGP and EGGP present the best performance for the real-world datasets. For the symbolic regression functions, LGP, CGP, and EGGP present mixed results dependent on each problem. For the real-world datasets, LGP shows a dramatic decrease in performance when compared to LGP-micro and the other graph-based methods.
- With the steady-state EA, LGP produces the best results for the symbolic regression functions and CGP and EGGP for the real-world datasets. LGP-micro, CGP, and EGGP show again mixed results on the symbolic regression functions, while LGP again performs much worse in comparison to the other graph-based methods.
- For the $(1 + \lambda)$ EA, results are also mostly mixed, but EGGP has the lower MAEs on the symbolic regression functions. On the real-world datasets, LGP has still some remarkably higher MAEs.
- LGP was the only graph-based method that was able to achieve a near-optimal fitness on function *nguyen5*. As GP also has a good performance for this function, finding the optimal solution, this could suggest that this function benefited from a macro operator at the program level (crossover in GP and macro-mutation in LGP).

For the evolution of digital circuits, we focus on the performance of algorithms when the $(1 + \lambda)$ EA is used, as it by far outperformed the generational and steady-state EAs (Sect. 4.1). According to Table 6, the results are the following:

- For multi-output benchmarks (adder and multiplier circuits), CGP and EGGP scale similarly well, with LGP-micro lagging slightly behind. LGP has the worst performance.
- For the parity benchmark, LGP-micro performs the best. EGGP follows closely and CGP doesn't scale well with the increasing number of inputs. As an exception, EGGP presents a lower CE value for *par7*.

In Table 10, we show the rankings and Friedman p -values for a comparison between the graph GP methods with the same evolutionary algorithm. As the difference between the generational and steady-state EAs was not clear, we show here results only for the generational and $(1 + \lambda)$ EAs. For the symbolic regression functions, the rankings confirm that LGP-micro achieves the best result with the generational EA and EGGP with the $(1 + \lambda)$ EA. On the other hand, on the real-world regression datasets, the best result using the generational EA was obtained by EGGP, and by CGP when the $(1 + \lambda)$ is used. CGP and EGGP are the better ranking methods when the $(1 + \lambda)$ EA is used for the adder and multiplier circuits, but all ranks are similar for the even-parity functions. This time, no Friedman p -value is significant. Again, this is because all these methods perform well in terms of percentage of correct bits (Table 6), and the difference between them lies more in the computational effort.

In Tables 11 and 12, we again show a Mann Whitney and A measure analysis for all individual problems for selected methods. For regression, we show a comparison between LGP-micro, CGP, and EGGP using the generational and $(1 + \lambda)$ EA, as both EAs performed well depending on the graph-based method used. For the digital circuits, as the $(1 + \lambda)$ EA was the clear winner, we show the comparison only for it.

From Table 11, we see that the better performance of LGP-micro using the generational EA on the symbolic regression functions is statistically significant with some large effect sizes. When the $(1 + \lambda)$ EA is used, CGP and in particular EGGP statistically outperform LGP-micro. The difference between CGP and EGGP is sometimes significant but with low effect sizes only. On the real-world regression datasets, CGP and EGGP again outperform LGP-micro with some large effect sizes using the generational EA. When the $(1 + \lambda)$ EA is used, the differences are significant and with large effect sizes, although, as the results from Table 5 are mixed, this still provides no conclusive insight. For the digital circuits (Table 12), most differences are not detected as statistically significant, and even less present high effect sizes. As discussed previously, this is due to all methods performing similarly well in terms of the quality of the final solution found, although they differ in how many evaluations they need to find it (CE values in Table 6).

In summary, results are quite mixed and context dependent for symbolic regression, although LGP-micro with a generational EA performed the best for the

Table 11 Selected statistical comparison between graph-based methods for regression benchmarks

EA	Methods	pagie1	nguyen3	nguyen5	nguyen7	airfoil	concrete	cooling	heating	yacht
Gen	LGP-micro vs CGP	-	0.82	0.68	0.72	0.90	0.77	-	-	0.64
	LGP-micro vs EGGP	0.59	0.87	0.69	0.71	0.83	0.74	-	-	0.65
	CGP vs EGGP	0.60	0.62	-	-	-	-	-	-	-
(1 + 4)	LGP-micro vs CGP	-	0.65	0.76	-	0.89	0.73	0.88	0.82	0.72
	LGP-micro vs EGGP	0.61	0.64	0.82	-	0.86	-	0.89	0.83	0.85
	CGPvs EGGP	0.58	-	0.61	0.62	0.58	0.76	-	-	0.67

For each pair of methods where the p -value of a Mann Whitney U test is less than α/N , $\alpha = 0.05$, where N is the number of benchmarks for the given class (Bonferroni correction), we show the Vargha and Delaney A measure for assessing the effect size. Large effects are marked in boldface (> 0.70)

Table 12 Selected statistical comparison between graph-based methods for digital circuits benchmarks

EA	Methods	add1	add2	add3	mult2	mult3	par3	par4	par5	par6	par7
(1 + 4)	LGP-micro vs CGP	–	–	0.61	–	0.99	–	–	–	0.52	0.77
	LGP-micro vs EGGP	–	–	0.58	–	0.96	–	–	–	–	–
	CGP vs EGGP	–	–	–	–	0.68	–	–	–	0.52	0.78

For each pair of methods where the p -value of a Mann Whitney U test is less than α/N , $\alpha = 0.05$, where N is the number of benchmarks for the given class (Bonferroni correction), we show the Vargha and Delaney A measure for assessing the effect size. Large effects are marked in boldface (> 0.70)

symbolic regression functions and EGGP with a generational EA for the real-world datasets. For digital circuits, results are clearer, with EGGP being the best method but LGP-micro outperforming it on all but one even-parity function.

Based on these results, the use of LGP with a fixed-size genotype and mutations that change only functions inside instructions or connections (LGP-micro) is recommended, as is done in CGP and EGGP, and this becomes evident when looking at the results from LGP on the real-world regression datasets (Table 5). As the difference between LGP-micro and CGP lies in the representation, we claim that the representation in LGP, where the number of registers ($10 + \text{\#Inputs}$) is much lower than the genotype size and registers can be overwritten, can be a disadvantage. However, LGP-micro performed better for the even parity benchmarks, even though CGP and EGGP outperformed it on the adder and multiplier circuits. As all configurations were the same between the two experiments and the three algorithms, one hypothesis is that the even parity benchmarks benefit from more sharing of results - less sharing occurs in CGP and EGGP, as any node can use any of the previous nodes as arguments, whereas in LGP this is limited by the number of available registers, which is significantly lower than the total number of instructions. We examine this hypothesis in Sect. 4.4.

4.3 Comparison with tree-based GP

In order to assess the impact of the graph representation when the same evolutionary algorithm and similar configurations are used, we compare GP with the graph-based method and using the EA that worked best on each benchmark class: LGP-micro with the generational EA for the symbolic regression functions, EGGP with the generational EA for the real-world regression datasets, and LGP-micro with the $(1 + \lambda)$ EA for the even parity circuits. From Table 5, apart from *pagie1*, GP performs better than LGP-micro with the generational and steady-state EAs. When the $(1 + \lambda)$ EA is used, GP has better results on *nguyen3* and *nguyen5*. With the exception of the *concrete* dataset when the $(1 + \lambda)$ EA is used, EGGP outperforms LGP on all real-world regression datasets, with some large improvements in MAE. Looking at Table 6, LGP-micro outperforms GP on all parity functions, both in terms of percentage of correct bits as in terms of Computational Effort, which shows that the graph representation presents a great advantage in this benchmark class.

Table 13 Statistical comparison between GP and LGP-micro (regression benchmarks) and between GP and EGGP (real-world problems), when the same EA is used

Pair	pagiel	nguyen3	nguyen5	nguyen7	airfoil	concrete	cooling	heating	yacht
GP-gen x LGP-micro/EGGP-gen	0.59	0.80	0.84	0.65	0.82	0.59	0.62	0.60	0.67
GP-steady x LGP-micro/EGGP-steady	0.62	0.76	0.87	0.61	0.79	-	0.58	0.63	0.62
GP- λ x LGP-micro/EGGP- λ	-	0.70	0.94	-	0.88	-	0.90	0.88	0.93

For each pair of methods where the p -value of a Mann Whitney test is less than α/N , $\alpha = 0.05$, where N is the number of benchmarks for the given class (Bonferroni correction), we show the Vargha and Delaney A measure for assessing the effect size. Large effects are marked in boldface (> 0.70)

Table 14 Statistical comparison between GP and LGP-micro, when the same EA is used, on the digital circuits problems

Pair	par3	par4	par5	par6	par7
GP-gen xLGP-micro-gen	–	0.81	0.98	0.99	0.99
GP-steady x LGP-micro-steady	–	0.87	0.99	0.99	0.99
GP- λ x LGP-micro- λ	–	–	0.94	0.99	1.00

For each pair of methods where the p -value of a Mann Whitney test is less than α/N , $\alpha = 0.05$, where N is the number of benchmarks for the given class (Bonferroni correction), we show the Vargha and Delaney A measure for assessing the effect size. Large effects are marked in boldface (> 0.70)

Tables 13 and 14 show the effect sizes for a statistical comparison between GP and LGP-micro/EGGP. On the regression benchmarks, in general GP is statistically better than LGP-micro with some large effect sizes. LGP was better on *pagiel* using the steady-state EA and on *nguyen7* using the $(1 + \lambda)$ EA, although the effect sizes are not large. EGGP is statistically better than GP on the real-world regression datasets, and with large effect sizes under the $(1 + \lambda)$ EA. On the even-parity circuits, almost all differences between GP and LGP-micro were significant and with a very high effect size. When the $(1 + \lambda)$ EA is used, GP performs better than before, but is still outperformed by LGP micro from *par5* onward.

In conclusion, the graph representation was a disadvantage for the symbolic regression problems considered here. On the other hand, it outperformed trees on the real-world regression datasets, which are much more difficult problems based on the error values obtained. This suggests that, although the results for the regression problem class are quite mixed, the graph representation has the potential of improving results, especially for more complex problems.

Graphs were also able to outperform trees for digital circuits benchmarks regardless of the EA being used. Further, the magnitude of the increase in performance increases with the complexity of the function, and also when graphs are combined with the $(1 + \lambda)$ EA (*par6* and *par7* in Table 6). Thus, the graph representation has features that are of advantage for evolving digital circuits, and the $(1 + \lambda)$ EA is capable of better exploiting these features. As the $(1 + \lambda)$ EA performs more local search, one of these features may be neutral genetic drift, which occurs more frequently in graph representations due to mutations in inactive portions of the genotype. This is in accordance with publications examining the search space of the task of evolving circuits and showing that allowing neutral genetic drift is of help for these benchmarks in CGP [22, 30, 36]. Thus, as shown by our results, even if we change GP to work with the $(1 + \lambda)$ EA, the inclusion of graphs is still able to outperform it on digital circuit benchmarks. Sotito and Rothlauf also show in [29] that increasing mutations on inactive instructions slightly improved search performance for some symbolic regression benchmarks. As in that publication the authors used the standard EA for LGP, which is the steady-state EA, the feature of neutral search should be probably potentialized in combination with the $(1 + \lambda)$ EA, especially for evolving digital circuits.

(a) CE over values for R in LGP-micro.

(b) CE over values for l in CGP.

Fig. 5 Computational effort (CE) for different values of R in LGP-micro and the levels-back parameter l in CGP for the even-parity problems (logarithmic scale). Both methods are using the $(1 + \lambda)$ EA, and the CE values were calculated over 100 runs for each configuration

4.4 Number of registers and levels-back parameter

In Sect. 4.2 we hypothesized that the better performance of LGP-micro on the parity functions lies in the small number of registers. A small number of registers forces evolution to reuse intermediate results more frequently. In turn, this helps optimization to develop more complex solutions quicker. To elaborate on this idea, we fix the evolutionary algorithm as being the $(1 + \lambda)$ EA, as it performed best on the parity functions, and carry out two experiments. In the first experiment we measure the performance of LGP-micro on the parity benchmarks using a rising number of registers from one to 100, by a step of 2. In a second experiment we test the “intermediate results reuse” factor for CGP. CGP implements the levels back parameter l which, similarly to the number of registers in LGP, can control the use of intermediate results. Measuring the performance of CGP

for $l = 1 \dots 100$ with a step of 2 helps us to see whether restricting the levels back parameter shows a specific behaviour, how this behaviour compares to restricting $|R|$ for LGP-micro, and how the results compare to the previous experiments with $l = \infty$.

Figure 5 shows the development of the CE for LGP-micro and CGP when letting R and l sweep from 1 to 100. All remaining algorithm parameters are set to the same values as in previous experiments. Following observations can be made:

- There is an optimal interval for R and l . LGP-micro shows best performance for $R \in [10, 15]$ and CGP for $l \in [15, 25]$. Because in previous experiments we have configured R for LGP-micro almost optimal based on the literature and selected for CGP the common, but vastly suboptimal $l = n_c = 100$, CGP underperformed. Given a better configuration of l , CGP should perform similarly to LGP-micro and EGGP in Table 6.
- The more complex a parity function gets, the more sensitive the setting of the R of LGP and l of CGP become. For LGP-micro the optimal interval for R gradually rises from $[10, 13]$, to $[10, 20]$ for *par7*, *par6*, *par5*, *par4*, and *par3*, in this order. CGP is more robust towards misconfigured l 's. For *par3* and *par4* there are no large differences in performances for $l > 20$. However, for larger parity functions the increase of CE rises significantly for $l > 20$.

These results confirm that more intermediate results reuse is beneficial for complex parity problems, and LGP and CGP provide a mechanism to control this reuse. The fact that LGP is less robust to higher values of R can be a consequence of registers being overwritten, as then we have two factors decreasing intermediate results reuse: more available instructions from the beginning of programs and overwritten results.

Similar impacts of the configuration parameters R of LGP-micro and l of CGP is an indication that these DAG-based approaches could probably deploy the very similar mechanisms and are in fact two different forms of the same principle. Similar insights have been observed in a more detailed work in [17].

5 Conclusions and future work

We have considered three graph GP methods (LGP, CGP, and EGGP), two forms of applying mutation to LGP (LGP and LGP-micro), and three evolutionary algorithms (generational, steady-state, and $(1+\lambda)$), as well as standard GP. After testing each combination of technique and evolutionary algorithm on regression and digital circuits benchmarks, we studied: (1) the impact in performance caused by the EA that is used; (2) the difference in performance between the graph GP methods; (3) the difference in performance between GP and the best performing graph GP method. Our main conclusions are:

1. The evolutionary scheme that performs better on the regression problem is dependent on the algorithm. For GP, it is always the generational EA. For LGP-micro, it is the generational EA on the symbolic regression functions and the

- $(1 + \lambda)$ on the real-world regression datasets. For CGP and EGGP it was the opposite. On the other hand, the $(1 + \lambda)$ EA greatly outperforms the other EAs on digital circuits for all algorithms, which shows that this problem class benefits from an intensified local search.
2. For graph-based methods, it is advisable to use a fixed genotype length combined with point mutations, as in LGP-micro, CGP, and EGGP. A representation that allows all nodes to be reused, instead of the limited registers set from LGP, also proved to work better, but presented worse performance on the even parity circuits, which shows that there are problems that benefit from limited reuse of instructions. The unrestricted mutation of connection genes in EGGP resulted often in better performances compared to CGP.
 3. There is no advantage of graph representations over trees on the symbolic regression problems, as GP using a generational EA worked generally better. However, graphs outperformed GP on the real-world regression datasets, which shows that graph-based methods can potentially improve performance on complex regression problems. Graphs also outperform GP on digital circuits, regardless of the EA being used, which leads us to conclude that this problem class benefits from features of the graph representation, such as neutral genetic drift. When used in combination with the $(1 + \lambda)$ EA, graph-based methods present the greatest advantage over trees, as this form of EA can further explore the graph representation features. Furthermore, graphs present a great advantage over trees for multiple output problems, regression included, as they can easily encode more than one output.
 4. LGP and CGP provide a way of controlling the reuse of intermediate results via the number of registers R and the levels back parameter l , respectively. By using lower values for these parameters, one can promote code reuse and improve performance in more complex parity functions, which explains the better performance of LGP-micro on this problem class. Although EGGP also presents a good performance without this feature, it is possible that adding this type of control could be of benefit.

We have made an initial effort to point out general differences for different groups of problems, so that we can have a direction to more specific analysis in the future, as an in-depth study in order to understand which properties of the $(1 + \lambda)$ EA and the graph representation, for example, are responsible for the improved performance on digital circuits. Some possibilities of future work include: (1) study if properties like storage of evolved information, preservation of diversity, and neutral search, are present when graphs are combined with the $(1 + \lambda)$ EA and if they are of help, as done in [29] for LGP and steady-state EA; (2) study how parametrization impacts the performance of graph GP methods, as done for the number of nodes available for reuse here and more generally for CGP in [17]; (3) expand the results obtained in this paper to other types of problems, such as control problems and more real-world problems; (4) study the phenotype biases of LGP, CGP, and EGGP, as well as the probability of a node being active, and if this can additionally explain the poor scaling of CGP in the parity functions, for example, or if a higher probability of

mutation to nodes that are least active could influence any phenotype length bias and impact search performance.

Acknowledgements This work was supported by Grant 016/07095-5, São Paulo Research Foundation (FAPESP). Part of this work was carried out during the tenure of an ERCIM ‘Alain Bensoussan’ Fellowship Programme.

Funding Open Access funding enabled and organized by Projekt DEAL.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. T. Atkinson, D. Plump, S. Stepney, Evolving graphs by graph programming, in *European Conference on Genetic Programming*. (Springer International Publishing, Cham, 2018), pp. 35–51
2. T. Atkinson, D. Plump, S. Stepney, Probabilistic graph programs for randomised and evolutionary algorithms. In: *Proc. International Conference on Graph Transformation, ICGT 2018, LNCS* (Springer, 2018, vol. 10887, pp. 63–78)
3. T. Atkinson, D. Plump, S. Stepney, Evolving graphs with semantic neutral drift. *Natural Computing* (2019). [arXiv:1810.10453](https://arxiv.org/abs/1810.10453)
4. T. Atkinson, D. Plump, S. Stepney, Horizontal gene transfer for recombining graphs. *Genetic Programming and Evolvable Machines* (2020)
5. M. Brameier, W. Banzhaf, *Effective Linear Genetic Programming*. Tech. Rep., Department of Computer Science (University of Dortmund, Dortmund, 2001)
6. M.F. Brameier, W. Banzhaf, *Linear Genetic Programming* (Springer, Berlin, 2007)
7. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, M. Löwe, Algebraic approaches to graph transformation—part I: basic concepts and double pushout approach, in *Handbook Of Graph Grammars And Computing By Graph Transformation: Volume I: Foundations* (World Scientific, 1997), pp. 163–245
8. J. Demšar, Statistical comparisons of classifiers over multiple data sets. *J.Mach. Learn. Res.* **7**(1), 1–30 (2006). <http://jmlr.org/papers/v7/demsar06a.html>
9. D. Dua, C. Graff, UCI machine learning repository (2017). <http://archive.ics.uci.edu/ml>
10. C. Fogelberg, M. Zhang, Linear genetic programming for multi-class object classification, in *AI 2005: Advances in Artificial Intelligence*. (Springer, Berlin Heidelberg, Berlin, Heidelberg, 2005), pp. 369–379
11. F.A. Fortin, F.M. De Rainville, M.A. Gardner, M. Parizeau, C. Gagné, DEAP: Evolutionary algorithms made easy. *J. Mach. Learn. Res.* **13**, 2171–2175 (2012)
12. B.W. Goldman, W.F. Punch, Reducing wasted evaluations in Cartesian genetic programming, in *Genetic Programming*. (Springer, Berlin Heidelberg, 2013), pp. 61–72
13. B.W. Goldman, W.F. Punch, Analysis of Cartesian genetic programming’s evolutionary mechanisms. *IEEE Trans. Evolut. Comput.* **19**(3), 359–373 (2015)

14. S. Harris, T. Bueter, D.R. Tauritz, in *A comparison of genetic programming variants for hyper-heuristics*, in *GECCO 2015 5th Workshop on Evolutionary Computation for the Automated Design of Algorithms*, vol. ECADA'15, (Madrid, Spain, 2015), pp. 1043–1050
15. P. Kaufmann, R. Kalkreuth, An empirical study on the parametrization of Cartesian genetic programming, in *Genetic and Evolutionary Computation (GECCO)*. (Compendium) (ACM, 2017)
16. P. Kaufmann, R. Kalkreuth, in *Parametrizing Cartesian genetic programming: an empirical study*, in *KI 2017: Advances in Artificial Intelligence: 40th Annual German Conference on AI*. (Springer International Publishing, 2017)
17. P. Kaufmann, R. Kalkreuth, On the parameterization of Cartesian genetic programming, in *IEEE Congress on Evolutionary Computation (CEC)* (IEEE, 2020)
18. P. Kaufmann, M. Platzner, in *Advanced techniques for the creation and propagation of modules in Cartesian genetic programming*, in *Conference on Genetic and Evolutionary Computation (GECCO)*, (ACM Press, 2008), pp. 1219–1226
19. J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (MIT Press, Cambridge, MA, USA, 1992)
20. J. McDermott, D.R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaskowski, K. Krawiec, R. Harper, K. De Jong, U.M. O'Reilly, Genetic programming needs better benchmarks, in *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation, GECCO '12* (2012), pp. 791–798
21. J. Miller, Cartesian genetic programming: its status and future. *Genetic Programming and Evolvable Machines* (2019). <https://doi.org/10.1007/s10710-019-09360-6>
22. J.F. Miller, S.L. Smith, Redundancy and computational efficiency in Cartesian genetic programming. *IEEE Trans. Evol. Comput.* **10**(2), 167–174 (2006). <https://doi.org/10.1109/TEVC.2006.871253>
23. J.F. Miller, P. Thomson, Cartesian genetic programming, in *Genetic Programming*, ed. by R. Poli, W. Banzhaf, W.B. Langdon, J. Miller, P. Nordin, T.C. Fogarty (Springer, Berlin Heidelberg, Berlin, Heidelberg, 2000), pp. 121–132
24. M. Nicolau, A. Agapitos, M.O'Neill, A. Brabazon, Guidelines for defining benchmark problems in genetic programming, in *Proceedings of 2015 IEEE Congress on Evolutionary Computation (CEC 2015)* (Sendai, Japan, 2015), pp. 1152–1159
25. R. Poli, W.B. Langdon, N.F. McPhee, A field guide to genetic programming. Published via <http://lulu.com> and freely available at <http://www.gp-field-guide.org.uk> (2008). (With contributions by J. R. Koza)
26. M. Schmidt, H. Lipson, in *Comparison of tree and graph encodings as function of problem complexity*, vol. GECCO '07, (Association for Computing Machinery, New York, NY, USA, 2007), pp. 1674–1679. <https://doi.org/10.1145/1276958.1277288>
27. L.F.D.P. Sotto, V.V. de Melo, M.P. Basgalupp, λ -LGP: an improved version of linear genetic programming evaluated in the ant trail problem. *Knowl. Inf. Syst.* **52**(2), 445–465 (2017)
28. L.F.D.P. Sotto, P. Kaufmann, T. Atkinson, R. Kalkreuth, M.P. Basgalupp, in *A study on graph representations for genetic programming*, in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference, GECCO '20*, (Association for Computing Machinery, New York, NY, USA, 2020), pp. 931–939. <https://doi.org/10.1145/3377930.3390234>
29. L.F.D.P. Sotto, F. Rothlauf, in *On the role of non-effective code in linear genetic programming*, in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '19* (ACM, New York, NY, USA, 2019), pp. 1075–1083
30. A.J. Turner, J.F. Miller, Neutral genetic drift: an investigation using Cartesian genetic programming. *Genet. Program Evolvable Mach.* **16**(4), 531–558 (2015). <https://doi.org/10.1007/s10710-015-9244-6>
31. A.J. Turner, J.F. Miller, Neutral genetic drift: an investigation using Cartesian genetic programming. *Genet. Program Evolvable Mach.* **16**(4), 531–558 (2015)
32. V.K. Vassilev, J.F. Miller, The advantages of landscape neutrality in digital circuit evolution, in *Evolvable Systems: From Biology to Hardware*, ed. by J. Miller, A. Thompson, P. Thomson, T.C. Fogarty (Springer, Berlin Heidelberg, Berlin, Heidelberg, 2000), pp. 252–263
33. J.A. Walker, The automatic acquisition, evolution and reuse of modules in Cartesian genetic programming. *IEEE Trans. Evol. Comput.* **12**, 397–417 (2007)
34. D.R. White, J. McDermott, M. Castelli, L. Manzoni, B.W. Goldman, G. Kronberger, W. Jaskowski, U.M. O'Reilly, S. Luke, Better GP benchmarks: community survey results and proposals. *Genet. Program. Evol. Mach.* **14**(1), 3–29 (2013)

35. G. Wilson, W. Banzhaf, A comparison of Cartesian genetic programming and linear genetic programming, in *Genetic Programming*. ed. by M. O'Neill, L. Vanneschi, S. Gustafson, A.I. Esparcia Alcázar, I. De Falco, A. Della Cioppa, E. Tarantino (Springer, Berlin Heidelberg, Berlin, Heidelberg, 2008), pp. 182–193
36. T. Yu, J.F. Miller, Neutrality and the evolvability of boolean function landscape. In: Proceedings of the 4th European Conference on Genetic Programming, EuroGP '01, pp. 204–217. Springer-Verlag, Berlin, Heidelberg (2001). <http://dl.acm.org/citation.cfm?id=646809.704083>

Authors and Affiliations

Léo Françoso Dal Piccol Sotto¹ · Paul Kaufmann² · Timothy Atkinson³ · Roman Kalkreuth⁴ · Márcio Porto Basgalupp⁵

Paul Kaufmann
paul.kaufmann@uni-mainz.de

Timothy Atkinson
timothy@nnaisense.com

Roman Kalkreuth
roman.kalkreuth@tu-dortmund.de

Márcio Porto Basgalupp
basgalupp@unifesp.br

¹ Fraunhofer SCAI, Sankt Augustin, Germany

² Johannes Gutenberg University, Mainz, Germany

³ NNAISENSE S.A., Lugano, Switzerland

⁴ TU Dortmund, Dortmund, Germany

⁵ Federal University of São Paulo, São José dos Campos, Brazil