ORIGINAL RESEARCH

Analysing the Influence of Reorder Strategies for Cartesian Genetic Programming

Henning Cui[1] · Andreas Margraf[2] · Jörg Hähner[1]

Abstract

Cartesian Genetic Programming (CGP) suffers from a specific limitation: *Positional bias*, a phenomenon in which mostly genes at the start of the genome contribute to a program output, while genes at the end rarely do. This can lead to an overall worse performance of CGP. One solution to overcome positional bias is to introduce *reordering methods*, which shuffle the current genotype without changing its corresponding phenotype. There are currently two different reorder operators that extend the classic CGP formula and improve its fitness value. In this work, we discuss possible shortcomings of these two existing operators. Afterwards, we introduce *three novel operators* which reorder the genotype of a graph defined by CGP. We show empirically on four Boolean and four symbolic regression benchmarks that the number of iterations until a solution is found and/or the fitness value improves by using CGP with a reorder method. However, there is no consistently best performing reorder operator. Furthermore, their behaviour is analysed by investigating their convergence plots and we show that all behave the same in terms of convergence type.

Keywords Cartesian genetic programming · CGP · Mutation operator · Reorder · Genetic programming · Evolutionary algorithm

Introduction

Cartesian Genetic Programming (CGP) is a form of *Genetic Programming* (GP) and a nature inspired *search heuristic*. The standard CGP version is represented by a directed, acyclic and feed-forward graph—instead of a tree structured representation, as is the case in GP. CGP was introduced by Miller [1] to evolve digital circuits, which is still an active research field as of today [2, 3]. Furthermore, its concept is used in a diverse field of problem domains like classification or regression [4]. Another more prominent application for CGP is on the subject of neural architecture search [5, 6].

While CGP has some advantages over GP—for example the absence of bloat [1]—it has its own specific problems. One issue is *positional bias* [7], which describes a limitation of CGP to fully explore its search space. A possible solution to this problem is called *Reorder*, an operator which shuffles CGP's genome ordering without changing its phenotype [7, 8]. However, Cui et al. [9] showed that the original reorder method does not fully mitigate positional bias but only lessens it. They changed the original operator so that nodes that should be reordered are spaced *equidistantly apart*. This fully eliminated positional bias and lead to an improvement in fitness. However, upon further inspection, enforcing an equidistant spacing brings the possibility that other biases and problems arise. Hence, in this work we will discuss possible issues as well as evaluate and introduce other novel reordering methods. We show empirically that CGP always profits from some reorder method, as it will lead to a better fitness value and/or decrease the number of iterations until a solution is found. Furthermore, by presenting and analysing convergence plots, we show that the behaviour of CGP will not change by introducing some form of reordering.

✉ Henning Cui
  henning.cui@uni-a.de

  Andreas Margraf
  andreas.margraf@igcv.fraunhofer.de

  Jörg Hähner
  joerg.haehner@uni-a.de

1 Organic Computing Group, Institute for Computer Science, University of Augsburg, Am Technologiezentrum 8, 86159 Augsburg, Bavaria, Germany

2 Fraunhofer IGCV, Am Technologiezentrum 2, 86159 Augsburg, Bavaria, Germany

In this work, we further expand on the *Equidistant Reorder* method, which spaces specific nodes equidistantly apart. We extend our previous work on this topic originally presented at ECTA/IJCCI 2023 [9]. Based on the previous work, we introduce new reordering methods and provide a broader empirical analysis by extending our set of benchmarks to include symbolic regression. Furthermore, the behaviour of all CGP algorithms used is inspected via convergence plots.

We provide a quick overview of the core principles of CGP in Sect. 2. Afterwards, Sect. 3 gives an overview of related work. We then reintroduce existing reorder operators and present our novel operators in Sect. 4. In the following Sect. 5, their performance and behaviour is evaluated. At last, our findings and discussions for future research directions are summarized in Sect. 6.

Cartesian genetic programming

This section reintroduces the supervised learning algorithm called Cartesian Genetic Programming. An additional focus point will be *positional bias* which impacts the search behaviour of CGP negatively.

Representation

CGP is represented by a directed, acyclic and feed-forward graph. It contains *nodes*, which are arranged in a $c \times r$ grid, with $c \in \mathbb{N}_+$ and $r \in \mathbb{N}_+$ defining the number of columns and rows in the grid respectively [1]. Using nodes, it is possible to feed-forward an arbitrary amount of program inputs via partially connected nodes to calculate any desired amount of outputs. However, with today's standard, a CGP model consists of only one row for most applications [4]. That means, $r = 1$.

The set of nodes in a graph defined by CGP can be divided into input-, output- and computational nodes. Input nodes directly receive the program input and relay these values to computational and/or output nodes. Differently, computational nodes are *represented* by three genes: one *function gene* and m connection genes, with $m \in \mathbb{N}_+$ being the maximum *arity* of one function in the whole function set. Such connection genes define a path between a previous and the current node, and a function genes addresses the encoded computational function of a node. If a function needs less than m inputs, all excess input genes are ignored. Contrary, output nodes define the programs final output and are *represented* by a single connection

gene in the genotype. They redirect the output of an input- or computational node for that reason.

Another important distinction for this work is the categorization of computation nodes into two groups: *active* and *inactive* nodes. The former are nodes which contribute to the program's final output because they are part of a path to one or multiple output nodes. The latter are not part of a path to the output nodes, hence they do not contribute to an output. However, by allowing such inactive nodes to persist throughout the training process, it may improve CGP's evolutionary search through neutral genetic drift [10, 11].

Given this description of CGP's representation, when we mention a graph with $N \in \mathbb{N}_+$ nodes, this graph will have only one row, N computational nodes, as well as additional input and output nodes corresponding to the given learning task. Furthermore, to improve readability and clarity, a graph defined by this standard representation will be called STANDARD.

In Fig. 1, an illustrative example of a graph with $c = 6$, $r = 1$ and $N = 3$ defined by a CGP genotype is depicted. Active nodes are drawn with a solid line, while inactive nodes are marked by dashed lines. The first two nodes (n_0 and n_1) are input nodes and correspond to two different program inputs. The following nodes (n_2–n_4) are computational nodes, and one output node (n_5) defines the program's output. In this example, both inputs are subtracted. Afterwards, this intermediate result is being added to itself and redirected as the program output. The node n_2 does not contains a path to an output node and is inactive.

Positional bias

Enforcing a feed-forward grid leads to one major negative impact of CGP's evolutionary search: *Positional bias*, which was found by Goldman and Punch [7]. Their findings show that the probability of a computational node being active is not uniformly distributed throughout the whole graph. Instead, computational nodes closer to the input nodes have a *higher chance* of being active compared to computational nodes near the output nodes. With this negative impact, positional bias makes it difficult to solve certain tasks and could increase the number of iterations until a solution is found [7, 12].

This bias can be visualized for a better understanding, as is shown in Fig. 2. The graph shows the distribution of active nodes, averaged over 75 independent runs on the *3-bit multiply* Boolean benchmark. On the x-axis, the position of a

Fig. 1 Example graph defined by a CGP genotype. The dashed node and connections are *inactive* due to not contributing to the output
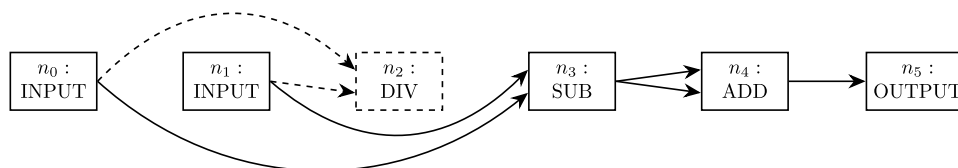
computational node in its graph is given. The y-axis indicates its probability of being active. Here, the effects of positional bias can be seen. About the first fifth of computational nodes have a (very) high probability of being active. However, the probability of being active for the remaining nodes are minimal. As a consequence, their mutations and usefulness cannot be evaluated regularly.

Related work

In our work, we focus on improving the *Reorder* operator from Goldman and Punch [7]. The motivation behind their work was mitigating the aforementioned positional bias [7, 8]. To the best of our knowledge, only Cui et al. [9] extended the concept of Reorder. In their work, the authors reordered active nodes by placing them equidistantly apart. This lead to an improved fitness value.

Additionally, various other articles layed out the foundation for this work. The most important ones for this work introduced novel operators for CGP and/or focused on various other limitations in CGP's ability to explore the search space.

Instead of reordering active nodes, Kalkreuth [13] swapped genes in the phenotype. For that reason, two new operators were introduced which were able to find solutions in less iterations.

Walker and Miller [14] also explicitly changed CGP's geno- and phenotype. With their extension, nodes or subgraphs in the genotype can be dynamically created, evolved, reused and removed. They were able to improve CGP's fitness value, with the addition of less training iterations needed.

It is also possible to extend the function pool of CGP, as was done by Harding et al. [15]. They included functions which directly change CGP's phenotype. This allowed them to solve problems with extremely large numbers of inputs.

The representation of a graph defined by CGP can be converted into an integer representation. Wilson et al. [16], however, introduced a floating point representation. This allowed them to use specialized operators to change the genotype. In this way, they were able to add or remove whole subgraphs.

Another method to mitigate positional bias is to allow arbitrary connections as long as no cycles form [7, 8]. Furthermore, regarding the mitigation of positional bias, their approach is also the current state of the art. However, Cui et al. [17] found that this may lead to other problems and, consequently, the search space is not explored optimally. To solve this problem, they introduced weights to guide the mutation of connections, which can reduce positional bias for STANDARD.

A different route is to decrease the size of the search space, as was done by Suganuma et al. [6]. They used a highly specific function set for neural architecture search for that goal. With another method, Torabi et al. [5] used a specific crossover operator to decrease the size of the search space which uses specific domain knowledge.

Reorder strategies

As previously mentioned in Sect. 2.2, enforcing a feed-forward grid leads to positional bias. To mitigate these negative effects, Goldman and Punch [7] introduced two new extensions to CGP. In this work, we build upon one of those two: *Reorder*, an extension which shuffles the genotype without changing the phenotype. For that reason, the original Reorder strategy [7] is reintroduced for better understanding. We also introduce several novel reorder strategies, which may completely remove positional bias or deliberately introduce other skewed distributions of active nodes.

Please note that all operators change CGP's genotype. However, by respecting the sequence of operations of active nodes, the phenotype—and thereby the program's output—stays the same.

Fig. 2 Distribution of active nodes over a graph defined by CGP. Example generated over 75 independent runs on the 3-bit multiply Boolean benchmark
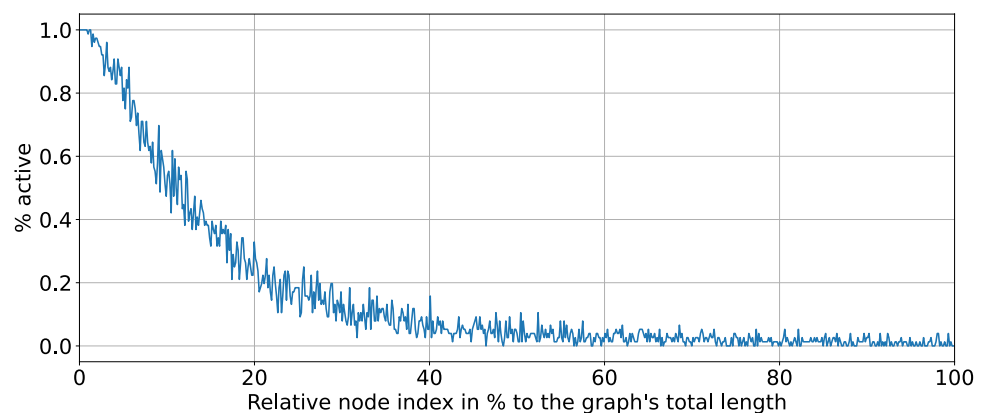
Original reorder

The original reorder operator is able to mitigate positional bias [7, 8]. However, we argue that it still suffers from a lessened positional bias.

Reorder strategy

The original Reorder operator begins by preparing a dependency set $D := \{(a, b) \mid a \text{ is input or computational node}, b \text{ is computational node}\}$. It contains information about each computational node and the nodes from which it gets its input from. Furthermore, a new genome $\mathcal{G}'$ must be initialized, containing all input and output nodes from the original genome. As input and output nodes should not be assigned a new position, they will not be modified in the shuffled genotype.

Now, in order to be able to proceed with the shuffling of nodes, the concept of *satisfying the dependencies of a node* must be introduced. Let m be the arity, b be a computational node, and a_i for $i = 1, \cdots, m$ be an arbitrary node from which b gets its input from, that means $(a_i, b) \in D$. This indicates a directed edge connecting a_i to b via the connection gene of b for $i = 1, \cdots, m$. Furthermore, let $A_b := \{a_1, \cdots, a_m\}$ be the set of all nodes from which b gets its input from. Given this information, b is *satisfied* when all nodes in A are mapped into the new genome $\mathcal{G}'$.

With the introduction of satisfied nodes, the set of *addable* nodes Q can be created. This set is initialized by adding all computational nodes whose dependencies are *already satisfied* in $\mathcal{G}'$, that means: $Q := \{b \mid b \text{ is computational node in } \mathcal{G}, A_b \subseteq G'\}$.

Given both sets D and Q, a genotype can be reordered by repeating the following steps:

1. Select and remove a random node $n \in Q$.
2. Map n sequentially into the next free location in $\mathcal{G}'$.
3. For each node pair $(n, b) \in D$ with node b depending on n, do the following: If all dependencies of b are satisfied in $\mathcal{G}'$, add b to Q.
4. If all computational nodes have been assigned a new position in $\mathcal{G}'$, stop. Otherwise, go to Step 1.

To better illustrate this operator, its workings are also depicted in Algorithm 1. Please note: When all computational nodes have been shuffled into $\mathcal{G}'$, Q must be an empty set, too. Furthermore, to improve readability in the next sections, the original reorder operator will be called ORIGINAL-REORDER.

Algorithm 1 Original Reorder

1: Initialize dependency set D
2: Initialize new genome $\mathcal{G}'$
3: Initialize addable set Q
4: **while** $Q \neq \emptyset$ **do**
5:     Select and remove a random node $n \in Q$
6:     Insert n into the next free location in $\mathcal{G}'$
7:     Update Q
8: **end while**

Reorder's limitation

In an earlier work, Cui et al. [9] showed empirically that ORIGINAL-REORDER does not fully eliminate positional bias but only lessens it. This is due to the constraint of node satisfiability.

A node can only be assigned into the new genome $\mathcal{G}'$ when all its dependencies are satisfied. At first, the *addable* set Q contains only nodes depending on input nodes—as these are the only nodes with satisfied dependencies. However, positional bias states that active nodes are likely to concentrate near input nodes. Again, these active nodes have little or no computational nodes prior to them. Hence, they have a higher probability of depending on input nodes compared to other nodes. As a consequence, computational nodes near input nodes have a higher probability of being added into the *addable* set early, as their dependencies are likely to satisfy early. From this, it follows that nodes near input nodes are also more likely to be inserted into $\mathcal{G}'$ early. This leads to a lessened but not completely mitigated positional bias.

The lessened bias can be exemplarily visualized, as is seen in Fig. 3. It shows the active node distribution of STANDARD compared to ORIGINAL-REORDER. Each distribution shows the averaged result of 75 independent and aggregated solutions for the *3-bit multiply* Boolean benchmark. Again, the x-axis defines the position of a node in its graph, with the y-axis indicating its probability of being active. The distribution for ORIGINAL-REORDER does not decrease as heavily as for STANDARD. Furthermore, the lowest probability for a node being active is around 40% instead of almost zero. However, there is still a visible decline of the nodes probability of becoming active. While positional bias is lessened for ORIGINAL-REORDER, it is not fully mitigated.

Equidistant reorder

To fully avoid positional bias, Cui et al.[9] introduced *Equidistant-Reorder*. Instead of depending on an addable set, active nodes are placed equidistantly apart in the

Fig. 3 Distribution of active nodes for STANDARD compared to ORIGINAL-REORDER
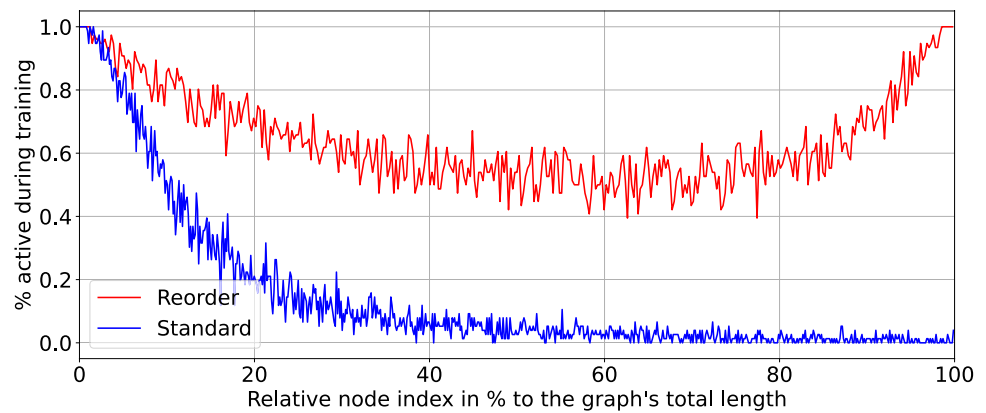
grid first. By enforcing an equidistant spacing, positional bias is completely eliminated. This tends to decrease the amount of computational nodes needed to train a CGP graph while also improving its fitness value.

To perform the Equidistant-Reorder operator on a genome, two different sets of nodes must be created. Firstly, all active computational nodes are defined in the set A. The second set $\tilde{A}$ contains all inactive computational nodes. They are defined as:

$$A := \{a_i \mid a_i \text{ is active computational node}, a_1 < a_2 < \cdots < a_n\}$$
$$\tilde{A} := \{\tilde{a}_j \mid \tilde{a}_j \text{ is inactive computational node}, \tilde{a}_1 < \tilde{a}_2 < \cdots < \tilde{a}_{N-n}\}$$

with $|A| := n$ and $N = \#$ computational nodes.

Let $s \in \mathbb{N}_+$ be the starting index and $e \in \mathbb{N}_+$ be the last possible index of computational nodes in the genome. Given s and e, the new positions for active nodes can be given by set L, which is also described in Algorithm 2. Here, L is defined as equidistantly spaced numbers over the interval $[s, e]$, with $L := \left\{ \lfloor s + i \cdot \frac{e-s}{n} \rfloor \mid i = 1, \cdots, n \right\}$. Given this formula, if $n = 1$, then $L = \{e\}$. If there exists only one active node, it will be placed at the end of the genome just before the output nodes. As a result, the active node is able to mutate its connection to an arbitrary computational node without any restrictions.

Opposing to L, $\tilde{L}$ is the set of genome locations for inactive nodes, which is defined as $\tilde{L} := \{s, s+1, \cdots, e\} \setminus L := \{\tilde{l}_1, \cdots, \tilde{l}_{N-n}\}$ with $\tilde{l}_1 < \tilde{l}_2 < \cdots < \tilde{l}_{N-n}$.

Afterwards, a new genome $\mathcal{G}'$ is initialized containing all input and output nodes from the original genome.

Again, input and output nodes will not be assigned new positions. All active computational nodes are placed into their new positions in $\mathcal{G}'$. This is done by assigning $a_i \in A$ the position $l_i \in L$, for $i = 1, \cdots, n$. The same approach is used for inactive nodes $\tilde{a}_j \in \tilde{A}$, as they are placed in $\tilde{l}_j \in \tilde{L}$ for $j = 1, \cdots, N - n$. As the order of computational nodes are not modified, the reordering does not change the semantic and no re-evaluation is required.

After inserting nodes, their connection genes must be corrected by changing their connection from the old genome's location to the new one. However, some inactive nodes may now hold a connection to a node in front of them. This can happen if they are connected to an active node, which gets a new position in front of it. As a result, the graph is not feed-forward anymore, violating the representation of CGP. In this case, a new connection to a previous node must be mutated to ensure the graph being feed-forward again.

The full pseudocode for a better understanding of this operator is depicted in Algorithm 3. Furthermore, to improve readability, the Equidistant Reorder operator is called EQUIDIST-REORDER in the following sections.

Algorithm 2 Lin-Space

Require: start point s, end point e, number of evenly spaced values n
1: $step \leftarrow \frac{e-s}{n}$
2: $S \leftarrow \emptyset$
3: **for each** $i = 1, \cdots, n$ **do**
4: $\quad S \leftarrow S \cup \lfloor s + i \cdot step \rfloor$
5: **end for**
6: **return** S

Algorithm 3 Equidistant Reorder

1: $A \leftarrow$ all active computational nodes
2: $\tilde{A} \leftarrow$ all inactive computational nodes
3: Initialize new genome $\mathcal{G}'$
4: $s \leftarrow |input\_nodes| + 1$
5: $e \leftarrow |input\_nodes| + |A| + |\tilde{A}|$
6: $L \leftarrow$ Lin-Space$(s, e, |A|)$
7: $\tilde{L} \leftarrow \{s, s+1, \cdots, e\} \setminus L$
8: **for each** $(a, l) \in (A, L)$ **do**
9:     $G'[l] \leftarrow G[a]$
10: **end for**
11: **for each** $\left(\tilde{a}, \tilde{l}\right) \in (\tilde{A}, \tilde{L})$ **do**
12:     $G'\left[\tilde{l}\right] \leftarrow G[\tilde{a}]$
13: **end for**
14: **for each** $node \in G'$ **do**
15:     **for each** $connection$ belonging to $node$ **do**
16:         $update(connection)$
17:         **if** $connection$ points to the front **then**
18:             $mutate(connection)$
19:         **end if**
20:     **end for**
21: **end for**

Uniformly distributed reorder

In the original work, EQUIDIST-REORDER is able to outperform ORIGINAL-REORDER and STANDARD in most cases [9]. However, by strictly enforcing an equidistant spacing, we believe that some useful structures might be destroyed. As the order of operation *for inactive nodes* are not conserved, they might have to mutate their connection to ensure a cycle-free graph. This, in turn, might dissolve meaningful structures.

Such effects of EQUIDIST-REORDER can be examined using a thought experiment, shown in Fig. 4. The optimal ordering for this program would be obtained by mutating one connection, from $B \rightarrow Result$ to $D \rightarrow Result$. However, by reordering nodes, such structures get destroyed. With node A in front of node C, it is not possible to obtain the optimal solution by changing only one connection. While the optimal solution is still possible to achieve, it is much harder after the reorder operator. The reason is that structures must be reinvented, which involves more changes in the genotype.

Motivated by this thought experiment, we introduce an operator similar to EQUIDIST-REORDER but without enforcing an equidistant spacing: *Uniformly Distributed Reorder* (UNIFORM-REORDER). Here, same algorithm described in Sect. 4.2 applies to UNIFORM-REORDER. The only difference lies in obtaining the new positions for active nodes, given by the set L. Instead of using Algorithm 2, we now sample from a continuous uniform distribution over the range $[s, e]$, with s being the starting index and e being the last index of computational nodes in the grid. This means, $L \sim \mathcal{U}_{[s,e]}$.

Negative positional bias based reorder

Another novel operator, which reorders and shuffles the genotype, will be called *Negative Positional Bias based Reorder* (NEGBIAS-REORDER). The idea of NEGBIAS-REORDER is to move *all* active nodes to the end of the genotype just before output nodes. Let n be the number of active nodes and e be the last possible index for computational nodes in a genome with $n, e \in \mathbb{N}_+$. Then, all active nodes are moved in their respective ordering to the positions $\{e - n + 1, e - n + 2, \cdots, e\}$. Concerning the inactive nodes, they are moved before all active nodes. Again, as is also the case for EQUIDIST-REORDER, all illegal connections must be
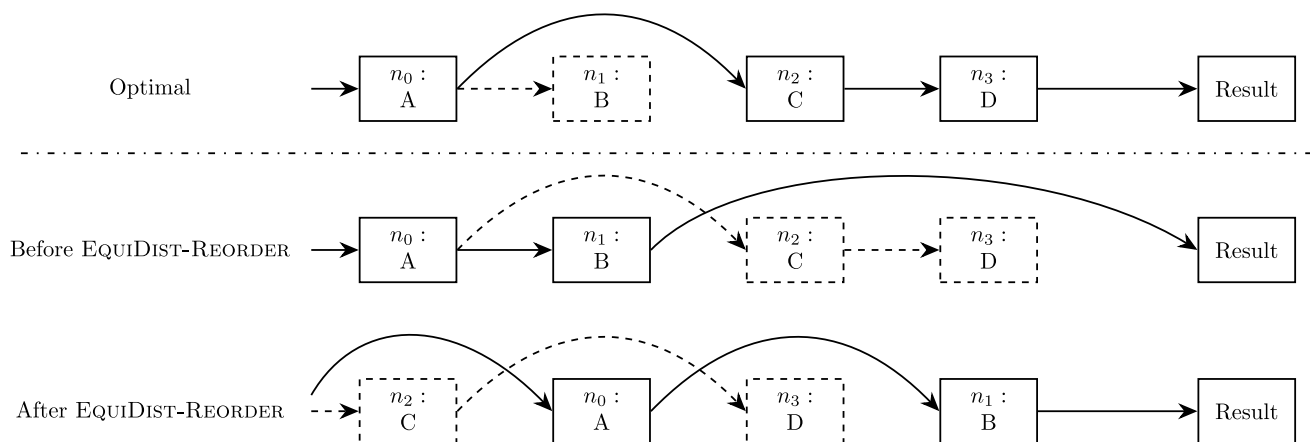

Fig. 4 Thought experiment illustrating limitations in EQUIDIST-REORDER. By reordering the nodes, the optimal solution is harder to obtain. Before reordering nodes, the node *Result* must mutate its connection

from node B to D. After reordering, more changes must be made to obtain the optimal solution

mutated anew. For NegBias-Reorder, Algorithm 3 can be used to describe its approach. The algorithm itself does not change with the sole exception of obtaining L, which is the set of new positions for active nodes. Instead of using Algorithm 2, we define $L := \{e - n + 1, e - n + 2, \cdots, e\}$.

By using this operator, we believe that it will have two different effects on the genotype. At first, active nodes now have a very high chance of mutating a connection to inactive nodes. Positive effects can be argued with CGP's *neutral genetic drift*, which describes the positive influence of *inactive nodes*. This should allow drastic changes in the phenotype during the following training iterations, which can play an important role in escaping local optima [11, 18, 19]. The other effect on the genotype is the mutation of new connections, which some inactive nodes must perform. For example, given the optimally found hyperparameters for Standard on the *3-bit multiply* Boolean benchmark averaged over 20 independent and randomly seeded runs, a mean of 863 out of 1,000 nodes were *inactive*. For *these inactive nodes*, a mean of 409 nodes had at least one connection to an active node. Because these inactive nodes are connected to an active node, this indicates that about half of all inactive nodes mutate due to NegBias-Reorder, which introduces further changes to the genotype. Nonetheless, deliberately mutating inactive nodes may improve CGP's performance for Standard, as was found by Turner and Miller [11] or Cui et al. [20]. Furthermore, according to Kaufmann and Kalkreuth [21], mutating a connection gene is the most meaningful change in the genotype of a CGP graph without extensions—as is the case for Standard. They show that changing connection genes can improve the fitness at most. To summarize, by introducing drastic changes in the genotype, it might help to efficiently explore the search space, and the escape of local optima.

We believe that Boolean benchmarks might benefit the most from NegBias-Reorder. According to Vasicek [22], the fitness landscape of such benchmarks are *deceptive*. In this context, this means that a multitude of different solutions lead to the same fitness value. Hence, by enforcing a drastic change in the phenotype due to NegBias-Reorder, more regions in the fitness landscape might be explored compared to Standard.

However, reordering active nodes toward output nodes after each iteration might hinder the evolutionary search process—as the genotype is drastically changing. This is why we also introduce a hyperparameter $p_{\text{reorder}} \in [0, 1]$, which describes the probability of NegBias-Reorder being executed. By varying the frequency of reorder operations, an equilibrium might be found between exploiting the current search space area and exploring new regions.

Left-skewed distribution reorder

Similarly to EquiDist-Reorder, enforcing a strict placement of nodes might greatly increase the difficulty to use existing genotypical structures. This is why we also include a probability based approach to NegBias-Reorder called *Left-Skewed Distribution Reorder* (LeftSkew-Reorder).

For this operator, active nodes are reordered according to a left-skewed beta distribution with $\alpha = 6$ and $\beta = 1$. Again, Algorithm 3 can be reused to better visualize this operator. The majority of the algorithm does not change with the exception of the set of new active node positions L. It is sampled by the aforementioned beta distribution $L \sim Beta(6, 1)$. To better visualize this specific distribution, its probability density function is also shown in Fig. 5. Again, similarly to NegBias-Reorder, the LeftSkew-Reorder operator also includes a hyperparameter $p_{\text{reorder}} \in [0, 1]$ to describe the probability of a reorder occurring.

Time complexity

The original authors of Original-Reorder show a time complexity of $\mathcal{O}(a \cdot N)$, where $a \in \mathbb{N}_+$ is the arity of the nodes and $N \in \mathbb{N}_+$ is the number of computational nodes of the considered CGP graph [8]. In their previous paper, Cui et al. [9] showed that EquiDist-Reorder also needs $\mathcal{O}(a \cdot N)$ time, which is the same as the original counterpart.

Fig. 5 Probability density function of the beta distribution with $\alpha = 6$ and $\beta = 1$
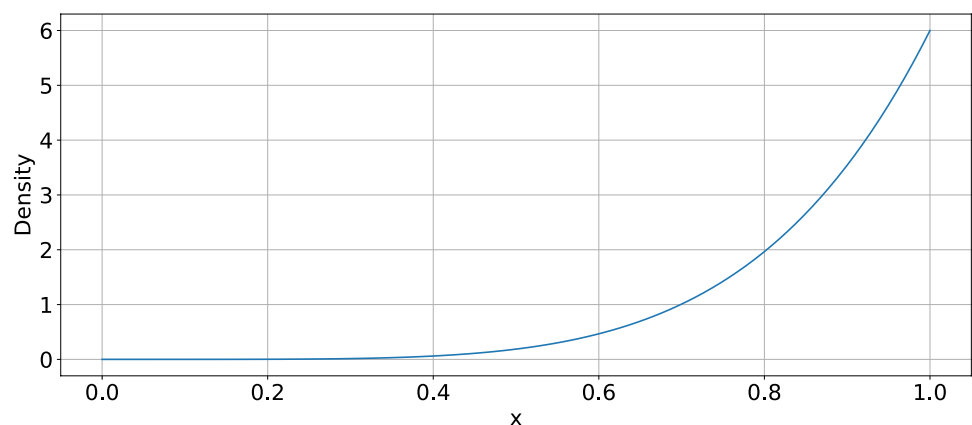
To obtain the time complexity for Uniform-Reorder, Neg-Bias-Reorder and LeftSkew-Reorder, their similarities and differences must be compared first. The reason is that they only differ in obtaining the set of new positions for active nodes.

Getting the set of all active and inactive computational nodes takes $\mathcal{O}(a \cdot N)$ time by following the connections from output nodes first backwards to input nodes. For $\tilde{L}$, a time of $\mathcal{O}(N)$ is needed to calculate the difference of two sets. The placing of active and inactive nodes into a new genome takes only $\mathcal{O}(N)$ time, as this only requires iterating through the sets of active and inactive nodes and their respective new positions once. For the procedure of updating the connections of every inactive computational node, each one must be visited to potentially update its c connection genes. This takes $\mathcal{O}(c \cdot N)$ time. Please note, $a := c$, as the highest arity a is also the number of connection genes c, leading to a runtime of $\mathcal{O}(a \cdot N)$.

To obtain the time complexity for L, two cases must be distinguished. For NegBias-Reorder, a set of numbers in a given range is generated. This can be done in $\mathcal{O}(N)$. Considering Uniform-Reorder and LeftSkew-Reorder, pseudo-random numbers must be generated from a predefined distribution. Here, initializing the distribution might be complex but is only done once before the start of the training process. Simply sampling N values from such distributions, however, can thus be achieved in amortized $\mathcal{O}(N)$.

Hence, Uniform-Reorder, NegBias-Reorder and Left-Skew-Reorder need $\mathcal{O}(a \cdot N)$ time, which is the same as Original-Reorder and EquiDist-Reorder.

Evaluation of different reorder strategies

To measure the effectiveness of our proposed methods, we conducted an empirical study. We describe our experimental design and attempt to answer the following questions:

Q1:   Which CGP variant needs, considering the learning task, less training iterations and/or has the better fitness value:

1. Standard

2. Original-Reorder

3. EquiDist-Reorder

4. Uniform-Reorder

5. NegBias-Reorder

6. LeftSkew-Reorder

Q2:   Do these variants behave differently regarding their convergence behaviour?

Experimental design

In this section, we describe CGP's configuration and the used benchmarks. Afterwards, a brief introduction into Bayesian comparison of models to evaluate and rank different CGP configurations is given. These models are used to compare multiple CGP configurations, which we use for our statistical analysis.

Evolutionary algorithms and configuration

As is standard for most CGP configurations, an elitist $(\mu + \lambda)$ evolutionary strategy (ES) with $\mu = 1$ and $\lambda = 4$ is used [4, 11]. Furthermore, CGP usually does not rely on crossover, as it generally does not improve CGP's performance [26]. This is why we also do not use one. Considering our mutation operator, we use *Single* [27]. It has the benefit that it does not need a mutation probability and achieves similar results compared to a standard point mutation. This operator works by mutating random nodes until an active node was mutated, which enforces a measurable change in the phenotype.

One major hyperparameter which must be optimized for all CGP versions and benchmarks is its number of computational nodes N. Therefore, we investigated $N \in \{50, 100, 150, \cdots, 2000\}$ and each N was tested 20 times with independent repetitions and different random seeds. If a probability for reorder p_{reorder} is needed, grid search was used. In addition to N, a $p_{\text{reorder}} \in \{0.0, 0.1, 0.2, \cdots, 1.0\}$ was examined. To find the best hyperparameter setting, we used the Plackett-Luce model described by Calvo et al. [25].

Bayesian data analysis

To ensure a fair comparison of extensions and a qualitatively sound evaluation, multiple configurations were investigated for each CGP version. Thus, each hyperparameter configuration has to be ranked to find the best solution. In our experiments, for all Boolean benchmarks we only examine the number of training iterations until a solution is found. The fitness value is considered regarding symbolic regression benchmarks. These numbers cannot be negative. Hence, other common distributions such as Student's t distributions can not be expected to model the data well [23]. This is why we performed a *Bayesian data analysis*—we utilized the Python library *cmpbayes* [24] for all statistical models—for the posterior distributions of our results. The model to compare the algorithms is based on the Plackett-Luce model described by Calvo et al. [25]. It allows the computation of a

set of ranked options by estimating the probabilities of each of the options to be the one with the highest rank.

Additionally, for each CGP version and their respective best hyperparameter set found, we report the 95 % *highest posterior density intervals* (HPDI) of the distribution of μ_{config}, where μ_{config} is a random variable corresponding to the respective *performance measurement*. At that, the distribution of μ_{config} is estimated by the gamma distribution–based model for comparing non-negative data from *cmpbayes* [24]. Please note, a 95 % HPDI interval $[l, u]$ can be read as $p(l \leq \mu_{\text{config}} \leq u) = 95\,\%$. This means, the probability of the algorithms results lying between the bounds l and u has a probability of 95 %.

Furthermore, prior sensitivity analyses were conducted prior to ensure the robustness of all models. As they always display similar results, robust and meaningful models are implicated. Finally, please note that *cmpbayes* uses Markov Chain Monte-Carlo sampling to obtain its distributions. Therefore, the usual checks to ensure convergence and well-behavedness (trace plots, posterior predictive checks, $\hat{R}$ values, effective sample size) were performed. For more information regarding the models, we refer to Kruschke and Pätzel [23, 24].

Benchmarks

To evaluate the methods, benchmarks from two different problem domains are used: Boolean and symbolic regression benchmarks.

Boolean Benchmarks

We used four Boolean benchmarks problems: 3-*bit Parity*, 16-4-*bit Encode*, 4-16-*bit Decode* and 3-*bit Multiply*. In the following, we will call these *Parity*, *Encode*, *Decode* and *Multiply*, respectively. Parity is regarded as too easy by the Genetic Programming community [28]. Nevertheless, it is commonly used as a benchmark in the literature [19, 21, 29]. Hence, we also included it in our evaluations for ease of comparison. Encode and Decode are problems with different input and output sizes (16 inputs and 4 outputs, and vice versa). At last, Multiply is a comparatively hard problem [30], and recommended by White et al. [28]. In addition, all four benchmarks were used to evaluate ORIGINAL-REORDER [7, 8] and EQUIDIST-REORDER [9], which is also why we utilized them for better comparison.

As we employed four Boolean benchmark problems, we also trained all CGP configurations with the standard function set for these problems. They contain the Boolean operators *AND*, *OR*, *NAND* and *NOR*. These benchmarks also lead to a standard fitness function, which is defined by the ratio of correctly mapped inputs. Let $f : \{0, 1\}^m \to \{0, 1\}^n$ be a correct Boolean mapping for $m \in \mathbb{N}_+$ inputs and $n \in \mathbb{N}_+$ outputs. Then, the fitness of

an individual $g : \{0, 1\}^m \to \{0, 1\}^n$, which relates to the learning task f, is defined as follows:

$$\frac{|\{x \in \{0, 1\}^m \mid f(x) = g(x)\}|}{|\{0, 1\}^m|} \tag{1}$$

The goal for Boolean benchmarks is to achieve a solution which is able to correctly map all inputs. Thus, each benchmark runs on an unlimited budget and we report the *number of training iterations until a solution is found (I2S)*. As four children are generated and evaluated in each iteration, the spent budget would be four times the I2S.

Symbolic Regression Benchmarks

In terms of symbolic regression benchmarks, we adhered to the recommendations from the GP community [28] and previous works [31]. Again, four different benchmarks were used: *Nguyen-7*, *Koza-3*, *Pagie-1* and *Keijzer-6*. Their functions are shown in Table 1.

The function set consists of eight mathematical functions: addition, subtraction, multiplication, protected division, sine, cosine, natural logarithm and the exponential function. As for the fitness function, the *mean absolute error* over the whole benchmark with n entries was used:

$$\frac{1}{n} \sum_{i=1}^{n} |y_i - x_i| \tag{2}$$

with $y_i \in \mathbb{R}$ being the prediction of a model and $x_i \in \mathbb{R}$ being the corresponding true value, for $i = 1, \cdots, n$.

In this setting, an algorithm is classified as converged when the fitness value becomes less than 0.01. Furthermore, each CGP variant is given $5 \cdot 10^5$ training iterations per run. As we employ a $(1 + 4)$-ES, each of the four children have to be evaluated during a single training iteration. This results in an upper limit for the budget of $2 \cdot 10^6$ fitness evaluations for symbolic regression benchmarks.

Performance on Boolean benchmarks

We report for each CGP variant its best configuration with respect to our hyperparameter search. To generate our

Table 1 Symbolic regression benchmarks used. $U[a, b, c]$ means that c uniform random samples are drawn from a to b, inclusive

Name	Variables	Equation	Training set	Test set
Nguyen-7	1	$\ln(x + 1) + \ln(x^2 + 1)$	$U[0, 2, 20]$	None
Koza-3	1	$x^6 - 2 \cdot x^4 + x^2$	$U[-1, 1, 20]$	None
Pagie-1	2	$\frac{1}{1 - x^{-4}} + \frac{1}{1 - y^{-4}}$	$E[-5, 5, 0.4]$	None
Keijzer-6	1	$\sum_i^x \frac{1}{i}$	$E[1, 50, 1]$	$E[1, 120, 1]$

results, each configuration was run 75 times with independent repetitions and different random seeds. Furthermore, we state the mean number of I2S, its HPDI, number of (active) nodes, and p_{reorder} if applicable. Additionally, we report the probability of one solution being the best out of the six, in accordance to the Bayesian model defined in Sect. 5.1.2. The summaries of our results on Boolean benchmarks can be seen in Table 2.

For all benchmarks, at least one reorder variant is always better than a standard CGP version. However, there is no reorder operator which behaves best for all benchmarks. ORIGINAL-REORDER—which is the current state of the art—is better than STANDARD in most scenarios but not ranked as the best performing one except Parity. For this benchmark, ORIGINAL-REORDER is preferred and the other shuffling methods do not lead to better results. However, these results should be taken with reservations as the different I2S are close and it is deemed as too easy.

Considering the other benchmarks, NEGBIAS-REORDER or LEFTSKEW-REORDER might be preferable in this setting. LEFTSKEW-REORDER is at the second place for Encode and the best option for Decode, while NEGBIAS-REORDER seems to be the best variant for Multiply. Their effectiveness might be explained due to their high amount of genetic drift and phenotypical changes, which they introduce each time. Combined with the deceptive fitness landscape mentioned in Sect. 4.4, this might be a reason why such reorder operators might be favourable. In addition, a high p_{reorder} value is preferred for almost all instances, which supports the claim of high exploration being favourable.

Another interesting insight is that EQUIDIST-REORDER and UNIFORM-REORDER do not perform similarly, and the same observation can be seen for NEGBIAS-REORDER and LEFTSKEW-REORDER. As their shuffling methods are very similar, we would expect a similar fitness as well. However, this is not the case for these four operators. Nevertheless, some trends can be seen. For both harder benchmarks—Decode

Table 2 Results on Boolean benchmarks for each CGP variant and its corresponding best configuration found

Variant	Mean(I2S)	HPDI	Active	Nodes	p_{reorder}	p(best)
Parity						
ORIGINAL-REORDER	343	[281, 419]	46.5	600	–	0.22
STANDARD	406	[329, 503]	30.5	200	–	0.18
UNIFORM-REORDER	423	[349, 515]	44.9	550	–	0.18
LEFTSKEW-REORDER	402	[324, 498]	42.3	500	1.0	0.15
EQUIDIST-REORDER	464	[377, 566]	37.2	400	–	0.14
NEGBIAS-REORDER	545	[427, 693]	45.9	700	0.5	0.13
Encode						
UNIFORM-REORDER	6,178	[5,380, 7,086]	78.8	350	–	0.20
LEFTSKEW-REORDER	6,466	[5,646, 7,385]	69.3	250	0.4	0.18
NEGBIAS-REORDER	6,413	[5,696, 7,198]	66.0	200	0.8	0.17
STANDARD	6,544	[5,627, 7,569]	82.4	400	–	0.16
ORIGINAL-REORDER	6,695	[5,722, 7,827]	94.9	500	–	0.16
EQUIDIST-REORDER	7,699	[6,591, 9,005]	71.3	250	–	0.13
Decode						
LEFTSKEW-REORDER	15,275	[13,859, 16,905]	166.8	550	0.8	0.21
STANDARD	15,432	[13,913, 17,100]	161.3	500	–	0.20
EQUIDIST-REORDER	16,177	[14,274, 18,319]	162.9	500	–	0.19
UNIFORM-REORDER	16,878	[15,221, 18,711]	173.8	600	–	0.16
ORIGINAL-REORDER	17,069	[15,139, 19,210]	152.4	450	–	0.14
NEGBIAS-REORDER	18,470	[16,844, 20,229]	129.8	300	0.6	0.11
Multiply						
NEGBIAS-REORDER	87,246	[72,937, 103,316]	125.2	750	0.9	0.21
EQUIDIST-REORDER	88,291	[77,922, 99,814]	91.1	350	–	0.19
LEFTSKEW-REORDER	94,940	[82,322, 109,039]	105.2	550	0.5	0.18
UNIFORM-REORDER	100,378	[84,129, 119,600]	90.4	350	–	0.18
ORIGINAL-REORDER	103,685	[89,632, 119,653]	101.2	500	–	0.16
STANDARD	145,299	[121,919, 172,348]	117.2	750	–	0.08

Here, *Active* is the mean number of active nodes; *Nodes* is the number of computational nodes used; and p(best) is the probability of the solution having the best test fitness value. Algorithms are sorted from best to worst in descending order according to p(best)

and Multiply—EQUIDIST-REORDER seems to be preferable over UNIFORM-REORDER, while the opposite can be said about Parity and Encode: Here, UNIFORM-REORDER leads to better results than EQUIDIST-REORDER.

One major caveat of NEGBIAS-REORDER and LEFTSKEW-REORDER, however, is their need for an additional hyperparameter $p_{reorder}$. This value differs for each CGP variant and benchmark, which is why no universal recommendation for $p_{reorder}$ can be given. As a result, this hyperparameter should be optimized for each problem statement, which leads to additional complexity and resources needed. This leads to a trade-off, as CGP's performance increases slightly but with a higher complexity for its hyperparameter search.

Performance on symbolic regression benchmarks

Considering the results on the symbolic regression benchmarks, they are shown in Table 3. Similar to Boolean benchmarks, each configuration was run 75 times with independent repetitions and different random seeds. Again, we list the CGP variant and their corresponding mean I2S, mean number of active and computational nodes, the probability of reorder (if applicable), success rate, train- and test fitness values, and the probability of the configuration being the best. For the success rate, we report the fraction of solutions that achieved a training fitness value of less than 0.01. The probability of one solution being the best is calculated via the Bayesian model defined in Sect. 5.1.2. The models and their corresponding probabilities were calculated on each configuration's test fitness values. In addition, the Keijzer-6 benchmark has very low fitness and HPDI values, indicating that it is very easy for CGP to solve it quickly. Thus, the results obtained from this benchmark should be met with scepticism.

Similarly to the Boolean benchmark results, there is no CGP variant that is always the best one. However, applying a reorder method is always preferable over STANDARD, as almost all configurations need less I2S while having a higher success rate. In addition, while ORIGINAL-REORDER is preferred over STANDARD in most cases, other reorder methods still outrank it.

Table 3 Results on symbolic regression benchmarks for each CGP variant and its corresponding best configuration found

Benchmark	Variant	Mean(I2S)	HPDI	Active	Nodes	$p_{reorder}$	SR	f(train)	f(test)	p(best)
Keijzer-6	EQUIDIST-REORDER	23	[16, 32]	11.6	50	–	1.0	0.001	0.001	0.28
	ORIGINAL-REORDER	25	[18, 35]	12.2	50	–	0.96	0.002	0.028	0.24
	LEFTSKEW-REORDER	28	[20, 38]	11.9	50	1.0	1.0	0.001	0.001	0.17
	NEGBIAS-REORDER	37	[26, 54]	14.8	100	0.4	1.0	0.002	0.001	0.15
	UNIFORM-REORDER	41	[32, 53]	24.4	300	–	0.96	0.004	0.002	0.10
	STANDARD	480	[312, 733]	13.3	150	–	0.92	0.005	0.025	0.06
Koza-3	LEFTSKEW-REORDER	20,339	[11,916, 34,454]	24.8	150	0.7	1.0	0.006	0.031	0.22
	EQUIDIST-REORDER	11,243	[6,888, 18,108]	20.8	100	–	1.0	0.006	0.045	0.19
	NEGBIAS-REORDER	9,370	[5,813, 15,051]	49.0	700	0.2	1.0	0.007	0.037	0.17
	ORIGINAL-REORDER	14,302	[8,676, 23,174]	30.1	250	–	1.0	0.007	0.023	0.16
	STANDARD	18,430	[11,651, 28,939]	28.1	650	–	0.98	0.009	0.018	0.14
	UNIFORM-REORDER	18,544	[11,183, 30,263]	41.6	600	–	0.98	0.008	0.028	0.12
Nguyen-7	NEGBIAS-REORDER	49,447	[32,829, 73,519]	33.4	250	0.4	0.98	0.009	0.023	0.19
	UNIFORM-REORDER	55,563	[36,527, 83,843]	40.6	550	–	0.94	0.009	0.829	0.18
	EQUIDIST-REORDER	54,855	[35,814, 82,851]	48.4	650	–	0.96	0.009	0.906	0.17
	LEFTSKEW-REORDER	59,022	[39,292, 87,894]	47.6	750	0.6	0.96	0.009	0.033	0.16
	STANDARD	126,498	[85,566, 182,333]	33.5	700	–	0.88	0.011	0.078	0.10
	ORIGINAL-REORDER	73,330	[47,996, 109,793]	34.3	300	–	0.92	0.01	0.034	0.02
Pagie-1	EQUIDIST-REORDER	352,345	[271,198, 442,921]	63.1	350	–	0.43	0.034	0.034	0.22
	ORIGINAL-REORDER	380,985	[299,196, 471,922]	63.4	350	–	0.29	0.036	0.036	0.19
	NEGBIAS-REORDER	385,752	[297,161, 485,266]	63.9	450	1.0	0.27	0.047	0.047	0.19
	LEFTSKEW-REORDER	405,887	[329,290, 493,721]	56.3	300	0.8	0.24	0.049	0.049	0.15
	STANDARD	491,071	[459,092, 500,000]	52.7	700	–	0.02	0.046	0.046	0.15
	UNIFORM-REORDER	430,564	[357,388, 500,000]	47.1	150	–	0.20	0.065	0.065	0.10

Here, *Active* is the mean number of active nodes; *Nodes* is the number of computational nodes used; *SR* is the success rate; and *p(best)* is the probability of the solution having the best test fitness value. Both *f(train)* and *f(test)* depict the respective mean fitness values. Algorithms are sorted from best to worst in descending order according to *p(best)*
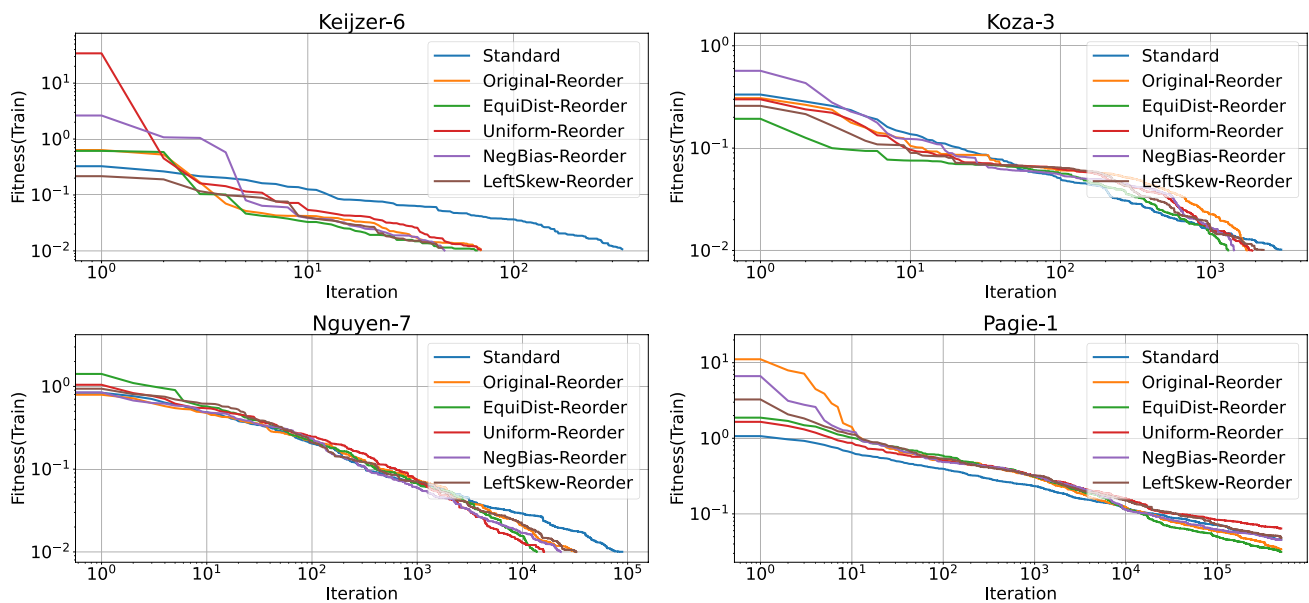
Fig. 6 Convergence plots for each regression benchmark. For better visualization, the x- and y-axis have a logarithmic scale
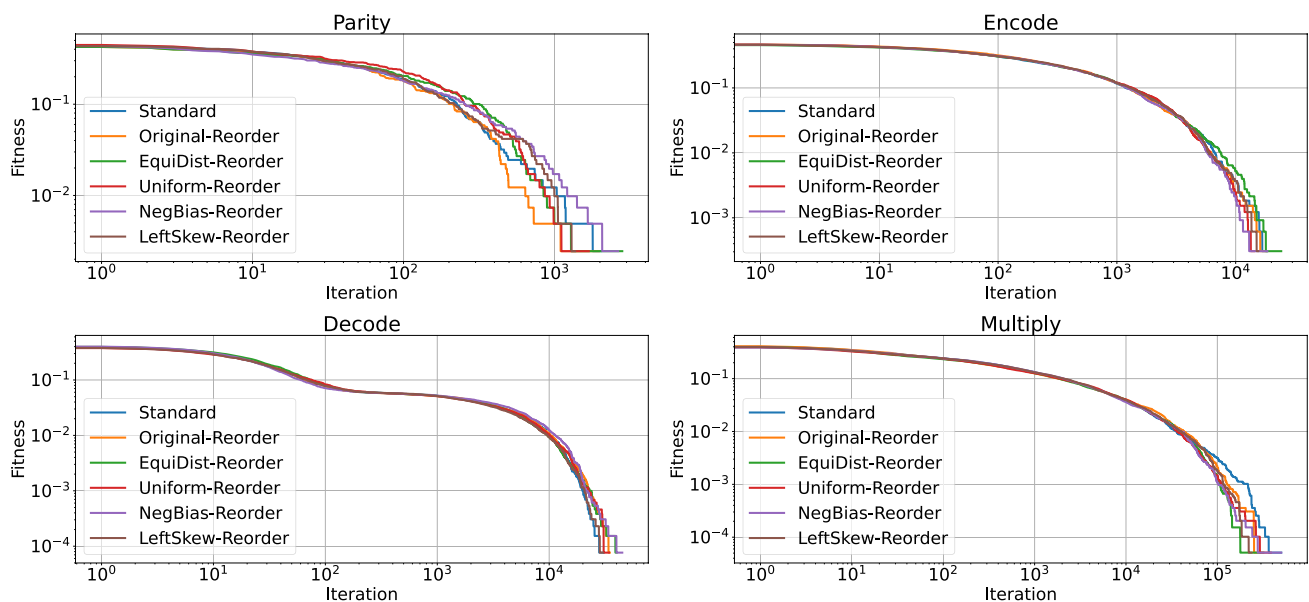

Fig. 7 Convergence plots for each Boolean benchmark. For better visualization, the x- and y-axis have a logarithmic scale

Considering the different shuffling methods, EQUIDIST-REORDER is the only one always under the top three CGP variants. In contrast to that, UNIFORM-REORDER—which has a similar reorder methods compared to EQUIDIST-REORDER—is performing worse in three cases. Similarly, NEGBIAS-REORDER and LEFTSKEW-REORDER show different fitness and HPDI values, a behaviour that can also be observed for Boolean benchmarks. Another similarity to Boolean benchmarks is the choice of p_{reorder}, as the optimally found values differ again for each benchmark and configuration. As a result, no general value for p_{reorder} can be proposed. This, again, implies a trade-off between better performance but an additional hyperparameter to optimize.
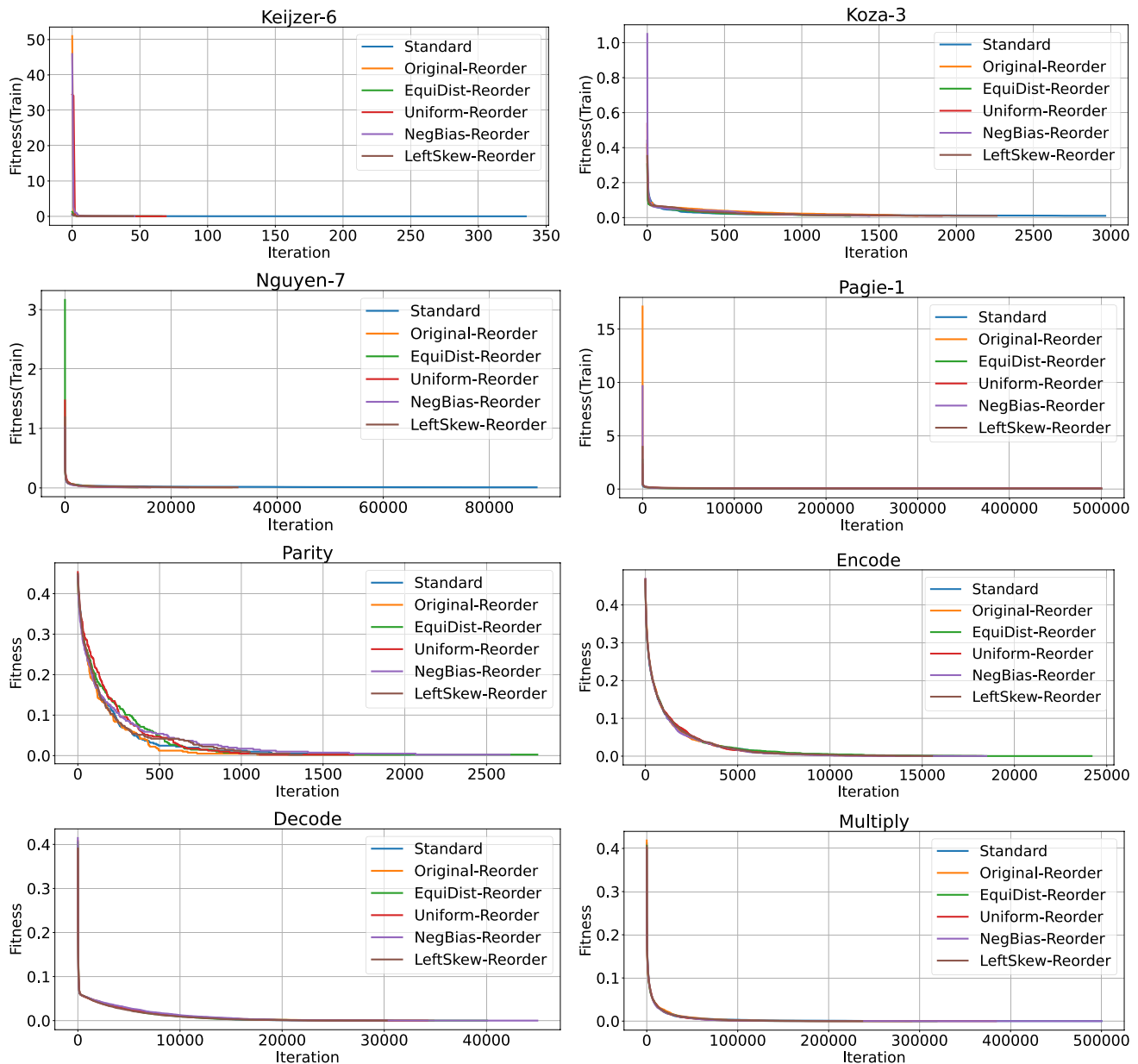
Fig. 8 Convergence plots for each symbolic regression and Boolean benchmark. Scales have not been altered

What is also interesting to see for the Koza-3 benchmark is the ranking of algorithms and their I2S. All top three variants have a success rate of one, which means that a solution was always found. However, the best solution needs, on average, about double the training iterations than EQUIDIST-REORDER and NEGBIAS-REORDER.

Convergence behaviour

To better understand the workings of our extensions, we depict convergence plots for both symbolic regression and Boolean benchmarks, as can be seen in Figs. 6 and 7 respectively. For these plots, we averaged the convergence of 75 runs. Furthermore, to easier see the differences, the x- and y-axis have a logarithmic scale. For unscaled plots, we refer to Fig. 8 in "Appendix". In addition, their respective standard deviation is included in Fig. 9.

When the plots for symbolic regression benchmarks are examined, they all show very similar behaviour. According to the categorization of Stegherr et al. [32], their convergence behaviour can all be grouped into the category *Fast to Slow*. Within the first few iterations, a relatively low
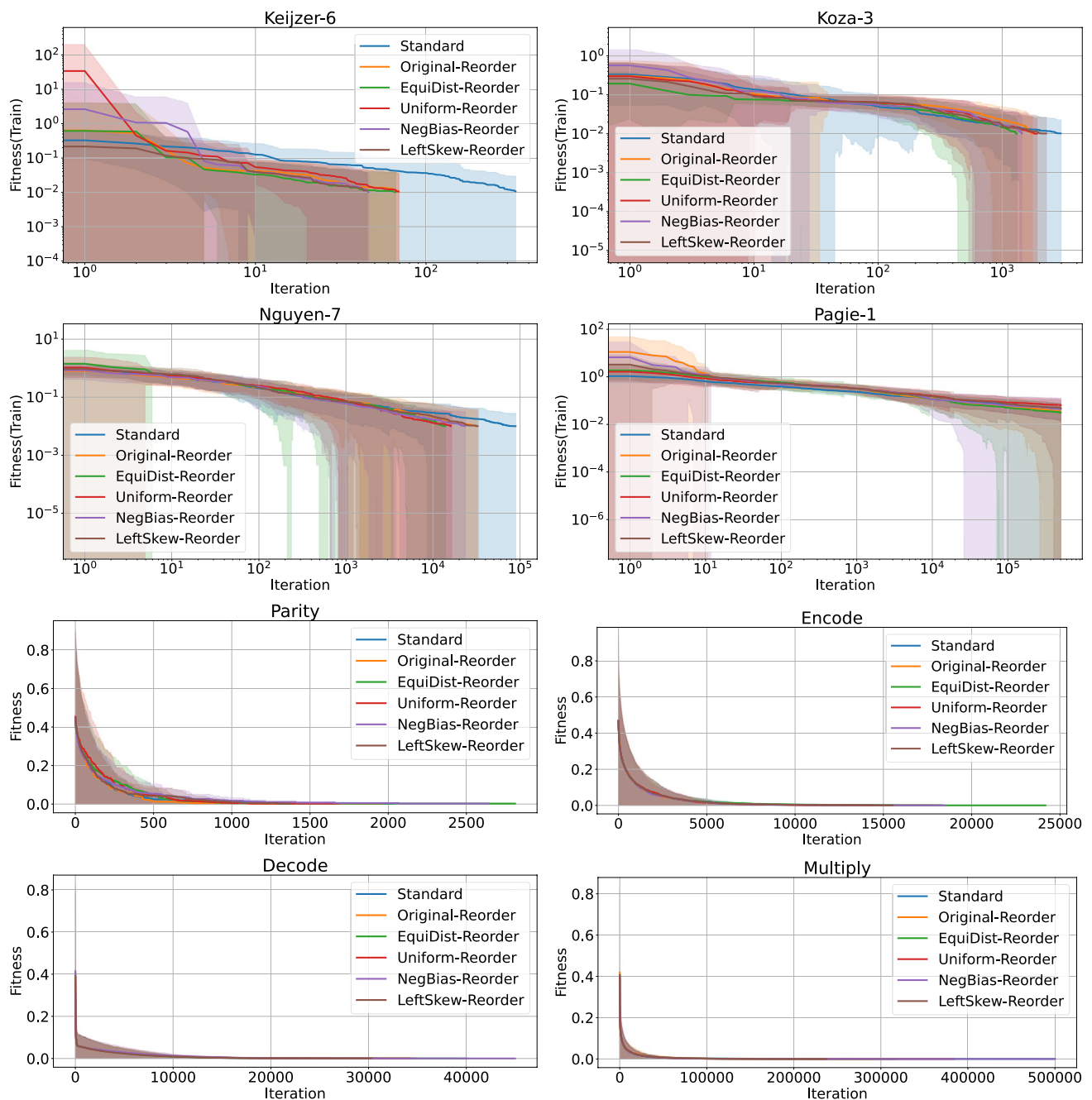
Fig. 9 Convergence plots for each symbolic regression and Boolean benchmark. The shaded area indicates their respective standard deviation. The x- and y-axis are scaled to make the standard deviation visible

fitness value is achieved. Afterwards, the rate of improvement decreases for all CGP variants, and a lot of training iterations are needed for small improvements. Interestingly, EQUIDIST-REORDER and UNIFORM-REORDER, as well as NEGBIAS-REORDER and LEFTSKEW-REORDER, respectively, did not show similar performances as can be seen in Sects. 5.2 and 5.3. Nevertheless, when only convergence

plots are examined, these reorder operators behave very similar. Thus, it can be concluded that the behaviour of CGP does not change when a shuffling method is included.

CGP's behaviour in accordance to Boolean benchmarks are all identical, as is depicted in Fig. 7. Again, their behaviour can be categorized into *Fast to Slow* [32]. As all CGP variants behave the same, no behavioural change can

be concluded again when a reorder method is included or not. However, please note that Boolean benchmarks show a *deceptive* fitness landscape [22]. As a multitude of different solutions lead to the same fitness value, examining the convergence plot might not lead to reasonable findings or might even be deceptive. This is why a conclusion based on Boolean benchmarks should be met with scepticism.

Conclusion

In this work, we *extended reorder methods* used by Cartesian Genetic Programming (CGP). We expanded upon the *original reorder* method (ORIGINAL-REORDER) from Goldman and Punch [7], as well as another reorder version called *Equidistant Reorder* (EQUIDIST-REORDER) from Cui et al. [9]. Three novel operators were introduced which shuffle CGP's genotype and assigns active nodes new positions in the genotype. Just as the two existing operators, the ordering of active nodes is preserved for all extensions, which means that the phenotype does not change. One of our new operators is called *Uniformly Distributed Reorder* (UNIFORM-REORDER) and builds upon EQUIDIST-REORDER. Instead of placing active nodes equidistantly apart—which is done by EQUIDIST-REORDER—their new positions are sampled from a continuous uniform distribution. The second novel extension is called *Negative Positional Bias based Reorder* (NEGBIAS-REORDER). It moves all active nodes to the end of the genotype, just before output nodes. The last operator, called *Left-Skewed Distribution Reorder* (LEFTSKEW-REORDER), is inspired by NEGBIAS-REORDER. Instead of using fixed placements, the positioning of active nodes is determined by a left-skewed beta distribution.

To measure the effectiveness of the new operators, we conducted an empirical study. A total of six algorithms were compared: the standard CGP formula (STANDARD), ORIGINAL-REORDER, EQUIDIST-REORDER, UNIFORM-REORDER, NEGBIAS-REORDER and LEFTSKEW-REORDER. We tested them on four Boolean- and four symbolic regression benchmarks, and optimized the hyperparameters for each setting. To find the best hyperparameters and to rank the effectiveness of algorithms, *Bayesian data analysis* for the posterior distributions of results was performed.

Our statistical analysis shows that, in all cases, CGP including a reorder operator is able to surpass STANDARD regarding its *number of training iterations until a solution is found (I2S)* and/or fitness value. However, there is no reorder algorithm that is always able to deliver the best results for all benchmarks and the choice of the right shuffling algorithm highly depends on the benchmark. Furthermore, while NEGBIAS-REORDER or LEFTSKEW-REORDER may be the best choice for some benchmarks, they depend

on an additional hyperparameter. Thus, the complexity to find the best hyperparameters increases which leads to a trade-off between better performance but higher computational overhead beforehand. Regarding symbolic regression benchmarks, some reorder methods might lead to a better test fitness value but also increase the I2S. This, again, might lead to a trade-off in some specific cases. Considering harder problems, EQUIDIST-REORDER always outperforms UNIFORM-REORDER. This indicates that enforcing an equidistant spacing should not limit CGP.

Algorithmically, EQUIDIST-REORDER and UNIFORM-REORDER, as well as NEGBIAS-REORDER and LEFTSKEW-REORDER, are close. EQUIDIST-REORDER enforces an equidistant spacing, while UNIFORM-REORDER samples from a continuous uniform distribution. NEGBIAS-REORDER moves all active nodes near output nodes, while LEFTSKEW-REORDER samples from a beta distribution to approximate NEGBIAS-REORDER. However, their respective fitness values greatly differ sometimes.

Additionally, we investigated their respective convergence behaviour. Interestingly, *all* CGP versions show a *Fast to Slow* convergence plot [32]. However, the behaviour for Boolean benchmarks must be met with reservation because of their *deceptive* fitness landscape [22].

On another note: Reorder methods lead to an improved convergence time. This may be especially convenient for CGP's practical applications. Neural architecture search, for example, is a prominent use case of CGP [5, 6]. By decreasing the training time while increasing the performance values, computational time and resources can be saved which is especially desirable in such computational demanding applications.

For future works, their behaviour can be further analysed. While the convergence plots of EQUIDIST-REORDER and UNIFORM-REORDER—as well as NEGBIAS-REORDER and LEFTSKEW-REORDER—depict the same behaviour, their fitness value might differ greatly. As they are algorithmically close, the reason for their gap in performance is still an open question. Answering this question might even reveal other hidden biases or properties of CGP.

Other types of reordering methods can also be explored. For example, one could exploit the dependencies of nodes. Let t be a node that is going to be reordered; node t_{prev} be the nearest node from which t gets an input from. Furthermore, let t_{after} be the nearest node which uses the output of t. Then, t could be reordered to an arbitrary / random location between t_{prev} and t_{after}. The reordering method should start from the rightmost nodes and work towards the leftmost nodes. In this way, another unique algorithm can be developed which fully retains the dependencies of both active and inactive nodes.

Appendix: Convergence plots without altered axis

For further information and visual inspection, convergence plots without changes to the y- and x-axis are provided in Fig. 8. As is already mentioned in Sect. 5.4, their convergence behaviour can be classified as *Fast to Slow* [32]. This categorization is easier to see when the axis are not scaled.

Furthermore, Fig. 9 include their respective standard deviation. We included them as separate plots to increase visibility. Please note that the x- and y-axis are logarithmically scaled again. Otherwise, the standard deviation would not be visible. For both benchmarks, the standard deviation is relatively low. In the context of symbolic regression benchmarks, the CGP variants have a higher standard deviation at the beginning and at the end of its training. Boolean benchmarks only show a higher standard deviation during the beginning of the training, and little to no deviation during the rest.

Data Availability Not applicable.

Code availability The source code including benchmarks used to produce the experimental data is available at: https://github.com/CuiHen/Reorder_Strategies_for_CGP.

Declarations

Conflict of interest Not applicable.

Ethics approval Not applicable.

Consent to participate Not applicable.

Consent for publication Not applicable.

References

1. Miller JF. An empirical study of the efficiency of learning Boolean functions using a cartesian genetic programming approach, GECCO'99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 1999. pp. 1135–42.

2. Froehlich S, Drechsler R. Unlocking approximation for in-memory computing with cartesian genetic programming and computer algebra for arithmetic circuits. IT Inf Technol. 2022;64:99–107.

3. Manazir A, Raza K. PCGP: a parallel implementation of cartesian genetic programming for combinatorial circuit design and time-series prediction; 2022. pp. 1–4.

4. Miller JF. Cartesian genetic programming. Berlin: Springer; 2011.

5. Torabi A, Sharifi A, Teshnehlab M. Using cartesian genetic programming approach with new crossover technique to design convolutional neural networks; 2022.

6. Suganuma M, Kobayashi M, Shirakawa S, Nagao T. Evolution of deep convolutional neural networks using cartesian genetic programming. Evol Comput. 2020;28:141–63.

7. Goldman BW, Punch WF. Length bias and search limitations in cartesian genetic programming, GECCO '13. New York, NY, USA: Association for Computing Machinery; 2013. pp. 933–40.

8. Goldman BW, Punch WF. Analysis of cartesian genetic programming? Evolutionary mechanisms. IEEE Trans Evol Comput. 2015;19:359–73.

9. Cui H, Margraf A, Hähner J. Equidistant reorder operator for cartesian genetic programming. INSTICC (SciTePress); 2023. pp. 64–74.

10. Miller J, Smith S. Redundancy and computational efficiency in cartesian genetic programming. IEEE Trans Evol Comput. 2006;10:167–74.

11. Turner AJ, Miller JF. Neutral genetic drift: an investigation using cartesian genetic programming. Genet Prog Evol Mach. 2015;16:531–58.

12. Payne AJ, Stepney S. Representation and structural biases in CGP; 2009. pp. 1064–71.

13. Kalkreuth R. Phenotypic duplication and inversion in cartesian genetic programming applied to Boolean function learning, GECCO '22. New York, NY, USA: Association for Computing Machinery; 2022. pp. 566–569.

14. Walker JA, Miller JF. Evolution and acquisition of modules in cartesian genetic programming. Berlin: Springer; 2004. p. 187–97.

15. Harding SL, Miller JF, Banzhaf W. Self-Modifying cartesian genetic programming. Berlin: Springer; 2011. p. 101–24.

16. Wilson DG, Miller JF, Cussat-Blanc S, Luga H. Positional cartesian genetic programming; 2018. arXiv:1810.04119.

17. Cui H, Pätzel D, Margraf A, Hähner J. Weighted mutation of connections to mitigate search space limitations in cartesian genetic programming, FOGA '23. New York, NY, USA: Association for Computing Machinery; 2023. pp. 50–60.

18. Vassilev VK, Miller JF. The advantages of landscape neutrality in digital circuit evolution. Berlin: Springer; 2000. p. 252–63.

19. Yu T, Miller J. Neutrality and the evolvability of Boolean function landscape. Berlin: Springer; 2001. p. 204–17.

20. Cui H, Margraf A, Hähner J, Mernik M, Eftimov T, Črepinšek M. Refining mutation variants in cartesian genetic programming. In:

Mernik M, Eftimov T, Črepinšek M, editors. Bioinspired optimization methods and their applications. Cham: Springer; 2022. p. 185–200.

21. Kaufmann P, Kalkreuth R. On the parameterization of cartesian genetic programming; 2020. pp. 1–8.

22. Vasicek Z. Bridging the gap between evolvable hardware and industry using cartesian genetic programming. Cham: Springer; 2018. p. 39–55.

23. Kruschke JK. Bayesian estimation supersedes the t test. J Exp Psychol Gen. 2013;142:573.

24. Pätzel D. cmpbayes; 2023. https://github.com/dpaetzel/cmpbayes, commit = 4de0abc37ee28b35267db173d32bb96ca9e69236.

25. Calvo B, Ceberio J, Lozano JA. Bayesian inference for algorithm ranking analysis, GECCO '18. New York, NY, USA: Association for Computing Machinery; 2018. pp. 324–5.

26. Husa J, Kalkreuth R, Castelli M, Sekanina L, Zhang M, Cagnoni S, García-Sánchez P. A comparative study on crossover in cartesian genetic programming. In: Castelli M, Sekanina L, Zhang M, Cagnoni S, García-Sánchez P, editors. Genetic programming. Cham: Springer; 2018. p. 203–19.

27. Goldman BW, Punch WF. Reducing wasted evaluations in cartesian genetic programming. Berlin: Springer; 2013. p. 61–72.

28. White D, et al. Better GP benchmarks: community survey results and proposals. Genet Prog Evol Mach. 2013;14:3–29.

29. Kaufmann P, Kalkreuth R. An empirical study on the parametrization of cartesian genetic programming, GECCO '17. New York, NY, USA: Association for Computing Machinery; 2017. p. 231–32.

30. Walker JA, Miller JF. The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. IEEE Trans Evol Comput. 2008;12:397–417.

31. Kalkreuth R. A comprehensive study on subgraph crossover in cartesian genetic programming. INSTICC (SciTePress); 2020. p. 59–70.

32. Stegherr H, Heider M, Hähner J. Assisting convergence behaviour characterisation with unsupervised clustering, INSTICC (SciTePress); 2023. p. 108–18.