Weighted Mutation of Connections To Mitigate Search Space Limitations in Cartesian Genetic Programming

Henning Cui
henning.cui@uni-a.de
University of Augsburg
Augsburg, Germany

David Pätzel
University of Augsburg
Augsburg, Germany

Andreas Margraf
Fraunhofer Institute for Casting, Composite and Processing Technology IGCV
Augsburg, Germany

Jörg Hähner
University of Augsburg
Augsburg, Germany

ABSTRACT

This work presents and evaluates a novel modification to existing mutation operators for Cartesian Genetic Programming (CGP). We discuss and highlight a so far unresearched limitation of how CGP explores its search space which is caused by certain nodes being inactive for long periods of time. Our new mutation operator is intended to avoid this by associating each node with a dynamically changing weight. When mutating a connection between nodes, those weights are then used to bias the probability distribution in favour of inactive nodes. This way, inactive nodes have a higher probability of becoming active again. We include our mutation operator into two variants of CGP and benchmark both versions on four Boolean learning tasks. We analyse the average numbers of iterations a node is inactive and show that our modification has the intended effect on node activity. The influence of our modification on the number of iterations until a solution is reached is ambiguous if the same number of nodes is used as in the baseline without our modification. However, our results show that our new mutation operator leads to fewer nodes being required for the same performance; this saves CPU time in each iteration.

CCS CONCEPTS

• **Computing methodologies → Genetic programming**.

KEYWORDS

Cartesian Genetic Programming, CGP, Genetic Programming, Evolutionary Algorithm, Mutation

ACM Reference Format:
Henning Cui, David Pätzel, Andreas Margraf, and Jörg Hähner. 2023. Weighted Mutation of Connections To Mitigate Search Space Limitations in Cartesian Genetic Programming. In *Proceedings of the 17th ACM/SIGEVO Conference on Foundations of Genetic Algorithms (FOGA '23), August 30-September 1, 2023, Potsdam, Germany*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3594805.3607130

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in
FOGA '23, August 30-September 1, 2023, Potsdam, Germany
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0202-0
https://doi.org/10.1145/3594805.3607130

1 INTRODUCTION

Cartesian Genetic Programming (CGP) is a form of *Genetic Programming* (GP) that is not based on trees but instead on *directed acyclic graphs* whose nodes are arranged in two-dimensional grids. One of the first applications of this method was the development of electronic circuits [17] but CGP has also been used in other domains such as for the evaluation of sensor data [1] or for image processing [15].

Other than traditional GP, which typically performs *selection*, *mutation* and *crossover*, CGP rarely utilizes crossover as that typically deteriorates CGP's performance [9, 16][1]. Therefore, efficient mutation and selection operators are all the more important.

However, the mutation operators commonly used in CGP lead to some nodes spending many training iterations being unused. We conjecture that this leads to the search space being explored inefficiently, which, in turn, increases the iterations needed for training. To mitigate this problem, we introduce a novel mutation operator for CGP based on associating each graph edge with a weight which influences mutation probability.

We provide a quick overview of the core principles of CGP in Section 2. Afterwards, Section 3 gives an overview of related work. In Section 4, we discuss the aforementioned limitation of CGP's exploration more in-depth. Our novel mutation operator meant to remedy said limitation is introduced in Section 5 and its performance is analysed using several Boolean benchmark problems in Section 6. Section 7 summarizes our findings and discusses future research directions.

2 CARTESIAN GENETIC PROGRAMMING

In this section, we briefly reintroduce the core principles of the supervised learning algorithm called *Cartesian Genetic Programming* (CGP). In addition, the extension developed by Goldman and Punch [5] is presented.

2.1 Representation

A CGP model is commonly represented as a directed acyclic graph. This graph consists of *nodes* which are arranged in an $c \times r$ grid,

[1] Note that there are some exceptions that *do* suggest to use crossover such as [11, 30].

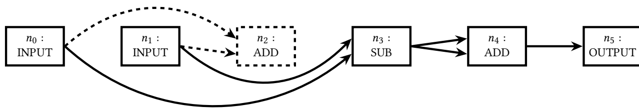

Figure 1: Graph defined by a CGP genotype. The dashed node and connections are *inactive* due to not contributing to the output.

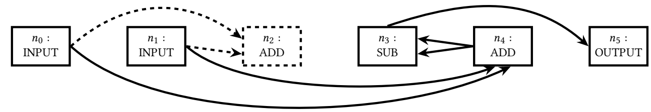

Figure 2: Graph defined by a CGP genotype with the DAG extension. Nodes can create connections to arbitrary nodes if it does not lead to a cycle.

with c and r defining the number of columns and rows in the grid, respectively [2]. Given an input, the CGP model produces an output by feeding it forward through partially connected nodes until an output node is reached.

The set of nodes in a CGP model can be divided into input, output and computational nodes. Input nodes are the ones that directly receive the program input. Computational nodes are *represented* by three genes in the genotype: a *function gene* and two *connection genes*. The function gene addresses the encoded computational function of the node. A connection gene defines a path between a previous and the current node. Via this path, data is transferred which can be either a program input or the computational output of another arbitrary previous node. At last, the output node defines where the program output is taken from. They are represented by a single connection gene.

Furthermore, computational nodes are categorized into two groups: *active* and *inactive* nodes. Nodes of the first group contribute to a program output, while inactive nodes do not because there is no path leading to an output node. In this manner, the set of active nodes define a model's phenotype while the genotype defines both active and inactive nodes.

An illustrative example of a graph defined by a CGP genotype can be seen in Figure 1. It shows a graph with $c = 6$ and $r = 1$, which is supposed to solve a task consisting of two inputs and one output. Here, the first two nodes n_0 and n_1 are input nodes and provide two different program inputs. The following nodes, n_2 to n_4, are computational nodes and the output is provided by n_5. In this example, the program inputs are subtracted. Afterwards, this intermediate result is added to itself and taken as the program output. Thus, every node drawn with a solid line is an active node; these nodes contribute to the program output. On the other hand, n_2 is an inactive node (marked by dashed lines) since it does not contribute to the output.

In the following sections, this standard CGP variant will be called Normal to improve readability.

2.2 Evolutionary algorithm used by CGP

Many CGP algorithms use the standard $(\mu + \lambda)$ evolution strategy with $\mu = 1$ and $\lambda = 4$. Here, elitism combined with neutral search is performed to improve its performance and speed in which a solution is found [25, 26, 31]. This means that, when an offspring and the current parent have the same fitness value, the offspring is always chosen as the next parent. By preferring the child, *neutral drift* is allowed to occur. Hence, the search algorithm is exploring various possible solutions given by different genotypes [19].

Concerning mutation operators, two types are often found. The original mutation operator simply iterates over all genes, mutating each with a probability defined by the user [18]. This, however introduces fitness plateaus since it is possible that neither of the active nodes is mutated within one iteration which in turn leads to the outputs not changing. In order to work around that, Goldman and Punch [6] proposed an alternative that they term *Single* mutation. It works by repeatedly mutating random genes until a gene associated with an active node has been changed. This makes changes in the phenotype (and thus the associated fitness value) more likely and eliminates the mutation rate hyperparameter. Furthermore, by mutating inactive genes earlier, neutral drift is able to occur. This is the reason why Goldman and Punch claim that *Single* is preferred when the optimal mutation rate is not known.

2.3 Extension: DAG

While the way that CGP restricts its search space leads to some advantages over less constrained GP methods (e. g., CGP does not need to actively control bloat [18]), there are additional implications. In practice, search space is actually even more limited than the grid alone might suggest: Goldman and Punch [5] found that the probability of a node being active is not uniform. On the contrary, nodes closer to the input nodes are more likely to be active than nodes further away from the input nodes. This can be explained visually with the help of Figure 1: Upon mutating a connection between nodes, the standard mutation operator assigns to each possible starting node the same probability to be chosen while the endpoint of the connection is kept fixed. Nodes at the beginning (the left side of the grid) have more options for successor nodes (remember that directed edges are only allowed to go from left to right) than nodes closer to the end (the right side of the grid). In this example, n_1's possible successors are n_2, n_3, n_4 or n_5 whereas n_4's possible successor is just n_5. Since a node is only active if one of its outgoing edges is connected to another active node, this leads to nodes at the beginning to be more likely to be active compared to nodes at the end, as there are more, possibly active, nodes that a connection could be evolved to.

This leads to some drawbacks, as this limitation makes it difficult or impossible to solve certain problems [5, 20]. As a remedy, Goldman and Punch [5] proposed two new variants of CGP: One of those is Dag, which removes the constraint of only allowing connections that go from left to right in the grid. Instead, Dag is able to mutate connections to every node in the genome as long as the new connection does not create a cycle. To visualize this concept, Figure 2 shows an example graph defined by Dag.

[2]Please take note of today's standard with $r = 1$ [18].

The second variant, called *Reorder*, proposed by Goldman and Punch [5] reorders the genotype. It takes all active nodes and rearranges them uniformly along the grid. By conserving the order of nodes as well as their processing sequences, the phenotype—and in turn the programs output—will stay the same. However, as we do not build upon it, we will not discuss Reorder in further detail.

3 RELATED WORK

Previously, other works have been investigating different search space limitations in CGP or explored new mutation operators, which layed out the foundation for this work. Our work focuses on improving the mutation operator, too. It is motivated by the possibility of a yet undiscussed limitation in CGPs search space. However, we include weights to influence the mutation of connections which separates us from previous works.

One of those is done by Goldman and Punch in [5] and [7]. Furthermore, some of their work has already been mentioned in Section 2.3. They hypothesize that the search space limitation leads to a decrease in performance and introduce two extensions to CGP to mitigate this problem.

The effects of neutral genetic drift, that is, the importance and influence of inactive nodes and thus the effect of genetic redundancy, have been considered by several groups. Yu and Miller [32] are one of the first to investigate neutrality in the context of CGP and showed empirically that neutral genetic drift improves the exploration and fitness of CGP, or at least does not impair it. On the other hand, Yu and Miller [31] later try to measure the impact of neutrality by comparing different genotypes and report a positive relationship between neutrality and efficient evolution of solutions. Finally, Turner and Miller [25] evaluate neutral genetic drift empirically and show that allowing it improves CGP's achieved fitness values.

However, the impact of neutral genetic drift is challenged by Goldman and Punch [5]. By experimenting with different extensions to the CGP formula, they present various other limitations and biases in CGP. Nonetheless, this leads them to believe that the current research on neutrality is incomplete, as they doubt that a great amount of inactive nodes lead to better neutral genetic drift. Furthermore, Collins [3] thinks along similar lines. The author refute the works of Yu and Miller [32] by showing empirically that the findings are worse than random search and introduce a CGP version without any inactive nodes.

It often depends on the use case, how the underlying search space can be transformed for a more efficient exploration. For example, Suganuma et al. [23] explore a highly specific function set for neural architecture search with the goal of reducing the size of the search space. Torabi et al. [24] on the other hand evaluate utilizing a specific crossover operator to improve the search space. This differs from our work in that we do not include specific domain knowledge from a single use case but try to make universal statements.

Other research directions focus on improving the mutation operator. Kalkreuth [10] introduces a new mutation operator specifically aimed at CGP's phenotype by duplicating and permuting active nodes. Another work by Cui et al. [4] extends existing mutation operators by adding a second mutation rate parameter. One defines a rate for inactive nodes while the other mutation rate defines a probability for active nodes.

4 LIMITED SEARCH SPACE EXPLORATION

CGP's limitation in exploring its search space is discussed at great length [3, 5, 7]. Nonetheless, in the context of DAG, nodes may be inactive during many iterations in the training process or only inactive for a very small number of training iterations. This could be another problem which has—to our knowledge—not been discussed as of yet.

4.1 Frequent node inactivity during training

As is mentioned in Section 2.3, there is the problem of limited search space exploration at hand. The authors Goldman and Punch [7] found a *positional bias* in NORMAL and conducted an extensive investigation to examine this problem in the context of NORMAL and DAG. With NORMAL, nodes at the beginning of a genome have a higher chance of becoming active than nodes at the end of a genome. In their studies, they found that a high percentage of nodes in NORMAL are never active. This is not desirable, as it can negatively affect CGP because those nodes are seldom used or their usefulness observed. This problem does not occur in DAG anymore as most nodes are active during training at least once.

However, in the analysis of DAG done by Goldman and Punch [7], their results are averaged over 50 runs. When individual runs are evaluated, we found similar (but less severe) problems regarding the limited search space exploration. While most nodes are indeed active at least once, it takes many iterations until inactive nodes become active. For example, given the boolean 4-16-bit Decode benchmark problem [6], nodes stay inactive for an average of 1,100 out of 22,000 iterations in our implementation.

Furthermore, we could see the trend of some nodes being active during most of the training steps, while other ones are seldom active. This can be visualized in Figure 3, which shows the active node distribution of training the boolean 16-4-bit Encode benchmark problem. It is trained with 100 nodes and the *Single* mutation operator. Here, each column corresponds to one node in the genotype and one row declares a single training step. If a node is active during one training step, it is visualized with a dot. Nodes which are more active during training have a darker colour in their respective column. The other way around, brighter colours indicate nodes which are less active during training. There are some dark columns, indicating that those nodes are almost always active during the training process. On the other hand, there are also some light columns with gaps. They reveal some periods during the training process where the node is not active. Similar trends can also be seen across multiple runs and on different datasets.

4.2 Periods of inactivity in the context of neutral drift

The vertical gaps in Figure 3 may not be desirable in particular. We hypothesize that they lead to—in the context of DAG—CGP's search space not being explored properly. As a consequence, more mutation steps could be necessary to solve a given problem. Some previous works claim that CGP benefits from *neutral drift* [25, 31] which is the effect of genotypic changes that do not change the

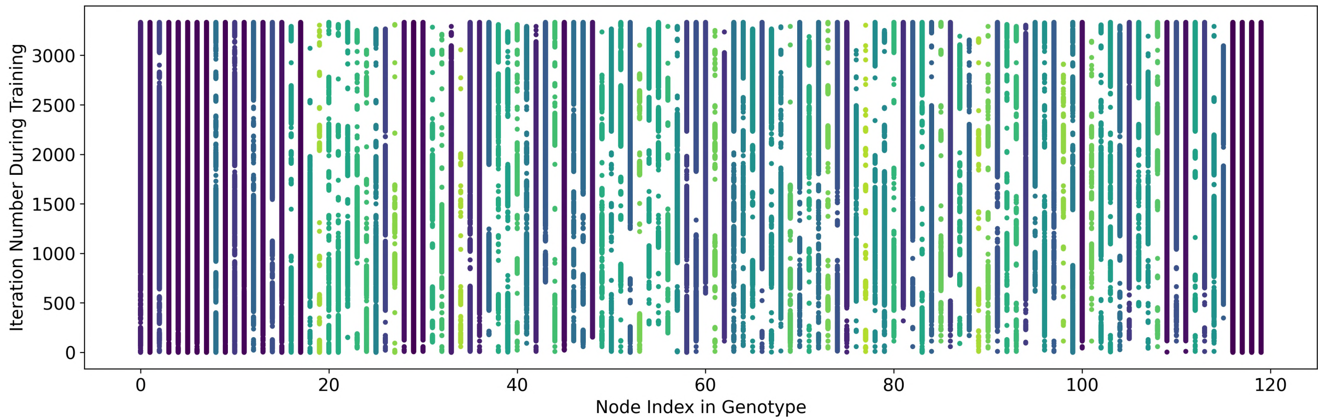

Figure 3: The active node distribution of one training processes of DAG on the Boolean 16-4-*bit Encode* benchmark problem. Each column corresponds to a node in the CGP graph with the x-axis defining its position. The y-axis corresponds to a specific iteration during its training. Each dot represents a specific node being active at a specific iteration. For each row, the darker the colour, the more iterations the node in this row has been active. Please note, the CGP graph contains 100 computational nodes, 16 input and 4 output nodes.

phenotype propagating to the next generations. In the context of CGP this means that genes of inactive nodes are mutated (without the nodes becoming active) and the containing individual is part of the next generation. However, we conjecture that if nodes are inactive too long periods of time, the benefits of neutral drift may decrease due to fitness pressure being too small.

5 MITIGATING THE LIMITED EXPLORATION

In order to shorten the periods of inactivity analysed in the previous section and promote the activation of inactive nodes, we now introduce a modification to the mutation operator used in CGP.

Our extended mutation operator is based on each node being associated with a weight which changes dynamically during training.

We chose to test our idea on both the NORMAL and DAG variants of CGP. As DAG is able to form a connection to arbitrary nodes, it could easily form connections to nodes with the highest probability. Thus, we hypothesize that it may profit from a guided mutation by weights.

As for NORMAL, we believe that our extension is able to improve this variant, too. By guiding the mutation of connections, we believe that we can decrease the severity of a positional bias to achieve a more uniform distribution of active genes in the genotype.

We refer to our CGP variant with weighted mutation of connections as WEIGHTED-CGP in the following text.

5.1 Weighted mutation of connections

Each computational or output node has one or several incoming connections. For computational nodes, there is one incoming connection for each input of the mathematical operation associated with the node whereas for output nodes there is exactly one incoming connection. Mutating one such connection between two nodes means that its *starting node* is changed while its end node is kept fixed (this ensures that each node always has as many incoming connections as its associated transformation requires inputs). Let S_v be the *set of allowed starting nodes* for a connection v. The original CGP mutation operator then simply chooses a new starting node for an existing connection *uniformly* from S_v. Note that S_v differs depending on the CGP variant: The standard CGP variant (called NORMAL in this paper) sets S_v as the set of nodes located in grid columns left of the columns that v's end node is in whereas DAG sets S_v as the set of all nodes except v's current end node.

For our weighted CGP mutation operator, we now associate each node n with a weight $w_n \in \mathbb{N}$. The weighted mutation operator then chooses a new starting node from the set of possible starting nodes weight-proportionally. This means, the probability of node $n \in S_v$ to be selected as the new starting node for an existing connection v is

$$\frac{w_n}{\sum_{\tilde{n} \in S_v} w_{\tilde{n}}} \tag{1}$$

Note that this is applicable without modifying NORMAL or DAG, since our extension only changes the set of allowed starting nodes S_v.

For each active node a, we set $w_a := w_{\min}$ where $w_{\min} \in \mathbb{N}_+$ is a hyperparameter. Furthermore, $w_{\min} > 0$ since otherwise active nodes are associated with a probability of 0 of being chosen for a connection again, which would, in return, restrict the exploration of the search space undesirably again.

For each inactive node b, the corresponding weight w_b is updated each iteration by

$$w_b \longleftarrow \min\left(w_b + w_{\text{step}}, w_{\max}\right) \tag{2}$$

where $w_{\text{step}} \in \mathbb{N}_+$ is a hyperparameter serving as a step size and $w_{\max} \in \mathbb{N}_+$ is another hyperparameter which restricts the maximum possible weight.

We deliberately chose to introduce a maximum weight $w_{\max}$ to maintain an element of balance in the context of exploration and exploitation. Given two inactive nodes b_1 and b_2 with b_1 being

inactive for less iterations than b_2. Without an upper bound on the weights, both nodes' weights could keep growing indefinitely. However, the w_{b_1} will always be smaller than w_{b_2} and mutating connections will in turn always favour b_2. Hence, we assume that unlimited weights would favour exploitation over exploration too strongly.

As a result of performing step-wise updates of w_n, the probability of a node n being chosen to be part of a new connection immediately after becoming inactive is only marginally higher than $w_{\min}$. This gives other nodes, which have been inactive for a longer period of time, a higher chance to be picked and having their potential in the phenotype explored. Furthermore, we hypothesize that it is not even desirable for nodes to immediately become active again, which would, for example be the case, if weights were set to $w_{\max}$ immediately upon becoming inactive. While this would lead to a higher percentage of nodes being active, this would inhibit neutral drift (inactive nodes changing over time) which may be beneficial (see Section 3). However, as has been already mentioned, Goldman and Punch [5] as well as Collins [3] doubt the importance of genetic drift in the context of CGP, so this preference may be debatable.

This concept of weighted connections is tested on both the Normal and Dag variants of CGP; within this paper, we name these new variants Weighted-Normal and Weighted-Dag, respectively.

5.2 Time complexity

The modifications to the mutation operator incur additional time complexity which we now shortly analyse. In the following, let $N \in \mathbb{N}_+$ be the number of input and computational nodes of the considered CGP graph. In each iteration, the weights of N nodes have to be updated which leads to an additional runtime of $O(N)$.

To mutate a single connection v, a new node must be drawn from S_v while considering the weights of each node in S_v. The distribution over possible nodes for a certain connection v is univariate discrete and has $|S_v| \leq N$ possible values. Drawing from this distribution is possible in $O(|S_v|)$ time. This is achieved by computing its cumulative distribution in $O(|S_v|)$, sampling a random number from $\mathcal{U}(0, 1)$ (possible in amortized $O(1)$) and then performing a binary search over the previously computed cumulative distribution in $O(\log |S_v|)$ [8].

6 EVALUATION OF WEIGHTED-CGP

In order to find out whether the modifications we proposed for the CGP mutation operator have merit, we conducted an empirical study[3]. This section first describes our experimental design and how we collected data. After that, we attempt to answer the following questions:

- **Q1** Which model is, on the learning tasks considered, faster in terms of *solution time*, the baselines or Weighted-CGP?
- **Q2** Is it possible to employ less nodes with Weighted-CGP while keeping similar number of training iterations?
- **Q3** In Section 4, we report that some nodes spend many training iterations inactive. Does the distribution of inactive nodes over the grid change with Weighted-CGP?

[3]Implementation and benchmarks can be found at https://github.com/CuiHen/Weighted-mutation-in-CGP.

Table 1: Number of nodes used for each benchmark task; determined by performing a hyperparameter study.

	Parity	Encode	Decode	Multiply
Normal	500	500	1,000	1,000
Dag	100	100	100	100

For each of these questions, we also want to assess how confident we can be in our answer given the data we collected.

6.1 Experimental design

We make use of four Boolean benchmark 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. Despite Parity being regarded as too easy by the Genetic Programming community [29], it is commonly used as a benchmark [12, 13, 31] and we include it in our evaluation for ease of comparison. Encode and Decode are problems with different input and output sizes and found in several CGP-related works as well [5–7]. Finally, Multiply [27] is a comparatively hard problem which was proposed by White et al. [29].

In 3-bit Parity, a Boolean function $f : \{0,1\}^3 \rightarrow \{0,1\}$ with property $f(x) = 1$ if and only if the number of ones in the vector $x \in \{0,1\}^3$ is odd, is learned. Considering 16-4-bit Encode, the objective is to take one-hot encoded numbers from zero to 15 and convert them into a four bit encoded integer. On the other hand, the goal of 4-16-bit Decode is the exact opposite, where four bit encoded integers are taken and must be converted into their respective one-hot encoding. For 3-bit Multiply, the aim is to multiply two 3-bit numbers into a 6-bit number. [27, 28]

Considering the function set used, all four benchmarks are trained with the standard function set for these Boolean problems: *AND*, *OR*, *NAND* and *NOR*.

As for the fitness function, let $f : \{0,1\}^n \rightarrow \{0,1\}^m$ be a correct Boolean mapping function. The fitness of an individual $g : \{0,1\}^n \rightarrow \{0,1\}^m$ relating to the learning task f is defined as:

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

For the baselines (Normal and Dag, we use the commonly-used $(1+4)$-ES with *Single* mutation introduced in Section 2.2. For Weighted-Normal and Weighted-Dag, we employ the $(1+4)$-ES as well but use the mutation operator proposed in Section 5.1, that is, *Single* mutation *extended with weighted connections.*

We determined an appropriate number of nodes N for each problem configuration by performing an exhaustive grid search over a fixed set of values for N. At that, for Normal we investigated $N \in \{250, 500, 1000, 2000\}$ and $N \in \{50, 100, 250, 500, 1000\}$ for Dag. Note that we deliberately chose lower values for Dag, since Goldman and Punch [7] showed that Dag needs less nodes than Normal. Table 1 states for each benchmark problem the number of nodes chosen for the following sections (highest mean performance over 15 independent repetitions with different random seeds).

As introduced in Section 5.1, Weighted-CGP has three hyperparameters: $w_{\min}$, $w_{\max}$ and w_{step}. For $w_{\min}$, this parameter can be chosen rather arbitrarily and we chose $w_{\min} = 100$. Since the other two hyperparameters can be expected to have strong impact on performance while at the same time a range of sensible values for them is not obvious, we perform experiments for a set of values for each of them: We combine $w_{\max} \in \{250, 500, 750, 1000\}$ and $w_{\text{step}} \in \{10, 25, 50, 100\}$. Note that the magnitude of sensible values for both $w_{\max}$ and w_{step} very likely depends on the magnitude of the chosen value for $w_{\min}$ which is why we deliberately chose larger steps.

With the number of nodes fixed according to Table 1, we now performed 20 independent repetitions with different random seeds for each combination of values of $w_{\max}$ and w_{step}. Runs were terminated either when a solution with maximum fitness (a rate of correct outputs of 100 %) was found or as soon as the number of iterations exceeded 750,000. We recorded as our measure of performance either the number of training iterations until that maximum fitness solution was found or the fact that no maximum fitness solution was found in time.

For the posterior distributions of our results and their evaluations, we perform a Bayesian data analysis[4]. The model to compare the baseline with Weighted-CGP is inspired by Kruschke [14]. It differs from Kruschke's model in that it is based on two gamma distributions, one for the baseline and one for our extension, and by the fact that we also model the observations where the tasks have not been solved within 750,000 steps as censored data. Since we only consider the number of training iterations, the units we consider can never be negative. Thus, other common distributions such as Student's t distributions can not be expected to model the data well. We also conducted a prior sensitivity analysis to ensure the robustness of the models. They always display almost identical results, implicating robust models. Finally, please note that *cmpbayes* uses Markov Chain Monte Carlo sampling to obtain its distributions. We performed the usual checks to ensure convergence and well-behavedness (trace plots, posterior predictive checks, $\hat{R}$ values, effective sample size). For more information regarding the models, we refer to Kruschke [14] and Pätzel [22].

6.2 Performance on training iterations

Summaries of the results obtained for the two baselines Normal and Dag (i. e. without our modifications) are given in Table 2. We provide these mainly to improve comparability with earlier work on CGP.

The best results on each of the four benchmark problems for Weighted-Normal and Weighted-Dag are reported in Table 3. For each individual result, we refer to Table 5 in Appendix B.

We compute for each considered hyperparameter configuration $(w_{\max}, w_{\text{step}})$ the 95 % *highest posterior density intervals* (HPDI) of the distributions of μ_1/μ_2 with $\mu_1 := \mu_{w_{\text{step}}, w_{\max}}$ and $\mu_2 := \mu_{\text{baseline}}$ where μ_1 and μ_2 are random variables corresponding to the respective *mean numbers of iterations until solution.* At that, the distributions of μ_1 and μ_2 are estimated by the gamma distribution–based model for comparing non-negative data from *cmpbayes* [22]. A 95 % HPDI interval $[l, u]$ can be read as $p(l \leq \mu_1/\mu_2 \leq u) = 95\,\%$, that is, *the ratio between the modified algorithm with the given parametrization and the baseline lying between the two bounds has a probability of 95 %.* Correspondingly, if the ratio takes a value of *less than one*, then our extension needs less iterations, whereas if the ratio's value is *greater than one* then the baseline needs less iterations. More information on these models can be found in Appendix A.

[4]For this task, the Python library *cmpbayes* [22] is used.

Table 2: Mean number (20 independent repetitions with different random seeds) of iterations until a solution is found by the Normal and Dag baselines (i. e. without our extension) on the benchmark problems. All runs considered here finished within the maximum number of iterations.

	Parity	Encode	Decode	Multiply
Normal	698.5	15,672.3	22,558.2	247,726.6
Dag	744.3	10,439.2	51,908.2	69,293.6

When best results are mentioned, please consider that the best results are always select via a Bayesian model to compare multiple algorithms. The model is based on the Plackett-Luce model described by Calvo et al. [2]. This model allows to compute for a set of ranked options an estimate of the probabilities of each of the options to be the one with the highest rank. In this case, the set of ranked options are all $(w_{\max}, w_{\text{step}})$ configurations, ranked by our performance metric.

In addition, we define for each benchmark a *region of practical equivalence* (ROPE). For the results in Table 3, the ROPE is defined by 1 % of the mean number of iterations until Weighted-CGP finds a solution for the given benchmark.

6.2.1 Results for Weighted-Normal. For our results of Weighted-Normal on Parity, Encode and Decode, the 95 % HPDIs for the ratio of the numbers of iterations have a lower bound of less than one but an upper bound of greater than one. Hence, there is no guarantee of Weighted-Normal performing better than the baseline in all cases. On another note, the Bayesian models employed allow us not only to estimate HPDIs but also the posterior probability of one method outperforming the other. Given these probabilities, our extension is slightly favoured, as the means are negative and the probability of Weighted-Normal being better is between 60 % to 81 %. This means, on average, Weighted-Normal will need less iterations to find a solution. However, based on the number of iterations alone, we cannot make a clear recommendation as to whether to keep using the baseline or our proposed extension. While some improvements can be achieved with Weighted-Normal, two additional hyperparameters are needed to achieve these results.

The only configuration where we can say with 95 % confidence that there is a definite difference is Weighted-Normal on Multiply with $(250, 50)$. Here, the 95 % HPDI is in the interval of $[0.3, 0.7]$, which corresponds to Weighted-Normal being faster than Normal by around 54,000 to 240,000 iterations. For this parametrization of Weighted-Normal on Multiply, there is a high probability (about 100 %) that Weighted-Normal has, indeed, a lower number of iterations until solution than the corresponding baseline.

6.2.2 Results for Weighted-DAG. Considering Weighted-Dag, a benefit can be seen for all benchmarks, as the lower and upper

Table 3: Results of iterations until finished training for WEIGHTED-NORMAL and WEIGHTED-DAG. Among others, we report the difference of means between the best configuration and the baseline for both empirical and model data. In addition, we present the lower and upper bounds of the 95 % HPDI as described in 6.2 and their numerical bounds. Furthermore, the probability p_b (in %, rounded) of the baseline being better, probability p_e of both configurations being practically equivalent and the probability p_w of WEIGHTED-CGP being better, is reported. The probability is given as a triplet: (p_b, p_e, p_w).

Variant	Benchmark	Weights	Empirical mean diff.	Model mean diff.	HPDI	total HPDI	Prob.
WEIGHTED-NORMAL	Parity	(750, 50)	−256	−260	[0.3, 1.4]	[−746, 96]	(8, 11, 81)
	Encode	(750, 100)	−3,912	−3,860	[0.4, 1.4]	[−11,773, 2,602]	(17, 2, 81)
	Decode	(500, 25)	−855	−866	[0.7, 1.3]	[−6,157, 4,273]	(38, 2, 60)
	Multiply	(250, 50)	−144,211	−138,464	[0.3, 0.7]	[−247,099, −54,358]	(0, 0, 100)
WEIGHTED-DAG	Parity	(250, 25)	−266	−267	[0.3, 1.3]	[−6367, −201]	(7, 8, 85)
	Encode	(1000, 10)	−3,092	−3,081	[0.5, 1.0]	[−22,381, −2,370]	(2, 5, 93)
	Decode	(500, 10)	−11,643	−11,637	[0.6, 1.0]	[−21,570, −1,862]	(2, 1, 97)
	Multiply	(500, 25)	−12,508	−12,397	[0.6, 1.1]	[−30,109, 4,136]	(10, 2, 88)

bounds of the HPDIs tend towards WEIGHTED-DAG being better than DAG. This notion can be confirmed by their stated probabilities, as WEIGHTED-DAG is better than the baseline by 85 % or more.

6.3 Influence on the number of nodes

Section 6.2 was only concerned with the number of iterations while keeping the number of nodes fixed for each experiment. We did deliberately not choose to change the number of nodes to see WEIGHTED-CGPs performance on the same amount of nodes. However, the usage of weights may influence the number of nodes needed to efficiently evolve CGP graphs. A higher number of nodes leads to a higher amount of CPU time per iteration as well as higher memory requirements. Also, a lower number of nodes decreases the size of the search space which can aid the evolutionary search.

For NORMAL, each combination of the following parameters are exhaustively observed[5]:

- $w_{\text{max}} \in \{250, 500, 750, 1000\}$
- $w_{\text{step}} \in \{10, 25, 50, 100\}$
- $N \in \{50, 100, 200, 250, 500, 1000, 2000\}$

Based on that data, we then fitted and sampled for each benchmark a model to compare multiple algorithms based on [2] to get an estimate of which of the available configurations performs best with the highest probability. Note that we used the same kind of model to select well-performing $(w_{max}, w_{\text{step}})$ configurations in Section 6.2.

We then compare that probably best-performing configuration with the baseline by fitting and then sampling the gamma distribution–based model described in Section 6.1. The probably best-performing parameters as well as a summary of posterior distribution of the gamma distribution–based model is given in Table 4. We define for each benchmark a *region of practical equivalence* (ROPE) as 1 % of the average number of iterations until NORMAL finds a solution for the given benchmark.

Our results of number of nodes needed is reported in Table 4. These results can be compared to our baselines in Tables 1 and 2.

[5]We do not consider DAG in this context, as we already use a very low N for that CGP variant.

Examining the number of nodes for each task as well as the resulting average number of iterations achieved, it can be seen that

- the optimal hyperparameter configuration for WEIGHTED-NORMAL contains a smaller number of nodes than the baseline, and
- that optimal configuration performs with reasonably high probabilities of 84 % (Parity and Encode), 95 % (Decode) and 98 % (Multiply) *better or equal* to the baseline.

We thus conclude that introducing the weighted mutation operator decreases the required number of nodes on all benchmark datasets.

6.4 Positional bias and node inactivity

In Section 4, we hypothesize that a limitation of DAG may be the prolonged inactivivity of nodes. Hence, we next investigate the number of iterations that nodes stay inactive during training and analyse DAG compared to WEIGHTED-DAG. Furthermore, we investigate the positional bias mentioned by Goldman and Punch [5]. To do so, we compare the positional bias of NORMAL to the positional bias of WEIGHTED-NORMAL graphically.

Note that we always compare the baselines against the best hyperparameter configuration of WEIGHTED-CGP which are given in Tables 3 and 4 for WEIGHTED-DAG and WEIGHTED-NORMAL, respectively. Please note that, for both NORMAL and DAG, we only visualize our findings for Multiply as Parity, Encode and Decode show almost-identical plots and behaviours.

6.4.1 Positional Bias. Regarding the positional bias of NORMAL, WEIGHTED-NORMAL is able to neutralize the bias slightly. This trend can be seen in Figure 4. It shows the percentage of nodes being active during training. As the baseline and WEIGHTED-NORMAL have different number of nodes, reporting the actual node index for both approaches would lead to less legible graphs. Hence, we report the relative node index in percentage to the graph's total length.

According to Figure 4, the baseline (orange line) shows the aforementioned positional bias. Nodes to the left of the grid are more active during training while subsequent nodes are barely active. Contrary to this, WEIGHTED-NORMAL (blue line) shows a similar

Table 4: Results of nodes needed with WEIGHTED-NORMAL. We report the probability p_b (in %) of the baseline being better, probability p_e of both configurations being practically equivalent and the probability p_w of WEIGHTED-NORMAL with less nodes being better. The probability is given as a triplet: (p_b, p_e, p_w).

Benchmark	#Nodes NORMAL vs. WEIGHTED-NORMAL	Weights	HPDI	total HPDI	Probabilities
Parity	500 vs. 100	(750, 25)	[0.49, 1.42]	[−474, 184]	(16, 16, 67)
Encode	500 vs. 250	(250, 100)	[0.65, 1.23]	[−3470, 1509]	(16, 21, 64)
Decode	1000 vs. 200	(500, 25)	[0.75, 1.12]	[−7640, 1015]	(5, 15, 80)
Multiply	100 vs. 200	(750, 50)	[0.50, 0.99]	[−92232, −8950]	(2, 0, 98)

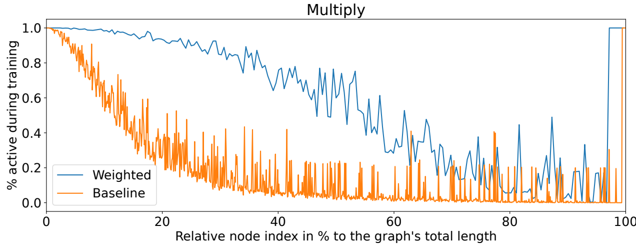

Figure 4: Node activity of NORMAL compared to Weighted-Normal during the training of Multiply. Weighted-Normal has less positional bias compared to the baseline, as later nodes are active more often.

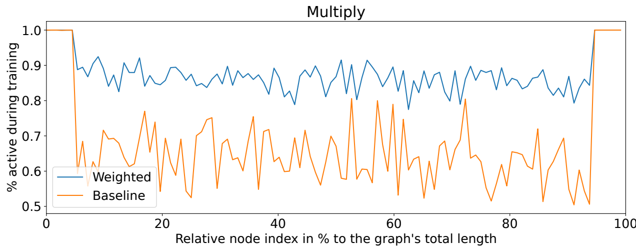

Figure 5: Node activity of DAG vs. Weighted-DAG during the training of Multiply. The nodes of Weighted-DAG are more active compared to their baseline.

but less extreme effect. Nodes at the beginning and in the middle of the grid are active more frequently compared to the baseline, which indicates less positional bias. This could may be the reason why WEIGHTED-NORMAL is able to achieve similar results with less nodes compared to NORMAL.

6.4.2 Node Inactivity. In regard to the limited search space exploration of DAG, their results show interesting trends. Figure 5 shows the percentage a node is active during the training of Multiply, comparing the baseline with WEIGHTED-DAG. Interestingly, nodes of the baseline are active during 50 % to 80 % of the training iterations whereas the nodes of WEIGHTED-DAG are active during 80 % to 90 %. Furthermore, it is interesting to note a decreased variance in the percentage of active nodes during training considering each node. This may indicates that there are no nodes in the graph which are more active or inactive compared to the others.

In addition, our experiments show that across all benchmarks, the maximal number of inactivity reduces by about 30 %. Furthermore, the frequency of such long inactivity periods should occur less, too. Given the number of iterations a node is inactive during a whole training, we counted the outliers outside of a 95 % HPDI interval. Comparing the number of outliers between WEIGHTED-DAG and DAG, WEIGHTED-DAG is able to reduce the number of outliers—and the number of nodes being inactive for long periods—by about 91 %. Hence, long periods of inactivity as described in Section 4 should occur less. Combined with the better performance of WEIGHTED-DAG, it is possible that a higher percentage of nodes being active leads to the improvement. However, as more nodes are active, more computational time is needed to calculate an output because more nodes contribute to such. Consequently, WEIGHTED-DAG leads to a trade-off. Less iterations are needed until a solution is found but each iteration is probably more expensive computationally.

On another note, another insight for DAG is the number of nodes being inactive for only one iteration. We found that nodes are more likely to be inactive for only one iteration during the training of DAG. On average, during the training of the baseline, nodes are 10,352 times inactive for only one iteration. On the other hand, this number decreases for WEIGHTED-DAG to an average of only 784. Given the same gamma distribution-based model for comparing non-negative data from Section 6.2, we build a model based on the number of iterations nodes spend inactive for DAG and WEIGHTED-DAG, respectively. Here, we found that nodes of the baseline are 7,700 to 11,700 times more inactive for only one iteration during their training compared to WEIGHTED-DAG. It is possible that this effect is another factor for WEIGHTED-DAGs improved performance. However, this remains only a conjecture as no conclusions can be drawn based on our experiments.

7 CONCLUSION

Long periods of node inactivity and the possibility of nodes never becoming active is a limitation in the exploration of the search space of Cartesian Genetic Programming (CGP). We hypothesize that these inactivity periods could decrease the exploration of CGPs search space. To combat this potential drawback, we introduced WEIGHTED-CGP, wherein we present a new, guided mutation operator for CGP. By utilizing weights during the mutation of connection genes, specific nodes have a higher probability of becoming active. Furthermore, we hypothesize that these weights are able to decrease the impact of the positional bias of CGP: nodes near input nodes are more likely to be active than nodes near output nodes—which

decreases CGPs exploration of its search space. This extensions was then tested and evaluated on several Boolean benchmark problems and on two different CGP variants: the standard CGP variant (NORMAL) and DAG.

When the number of nodes are not changed, WEIGHTED-NORMAL performs equally to NORMAL in most cases. Only in 1 out of 4 tasks, WEIGHTED-NORMAL definitely outperforms NORMAL. When DAG is compared to WEIGHTED-DAG, the latter is able to solve all benchmarks with less iterations needed given the right configuration.

We evaluate the influence of WEIGHTED-CGP on the number of nodes needed to train a problem, too. In this case, we can confirm that less nodes are needed which can save space, computational complexity and potentially decrease the size of the search space. Furthermore, we investigate the effect of WEIGHTED-CGP on the positional bias of NORMAL as well as the activity of DAG. This leads to new insights, as WEIGHTED-CGP is indeed able to decrease NORMALS positional bias. Finally, for DAG, nodes are more active during their training which potentially leads to less training iterations needed.

As for future work, there is the possibility to analyse the concept and workings of CGP and WEIGHTED-CGP in more detail. For example, a more detailed evaluation of the hyperparameters w_{max} and w_{step} could be conducted to better understand their interplay.

Furthermore, the analysis of active node distributions and the periods of inactivity is not complete. We found that, in the context of DAG, nodes are over 10 times more likely to be inactive for only one iteration. The real impact of this phenomenon remains unknown and further—more in-depth—experiments should be conducted.

Most of the current research of neutral genetic drift in CGP describes the importance of many inactive nodes. However, with WEIGHTED-DAG, most nodes are active during 90 % of the training and are able to outperform DAG which has less active nodes. A similar tendency can be seen with WEIGHTED-NORMAL, too. WEIGHTED-NORMAL and NORMAL perform equally in terms of iterations needed until a solution is found. However, WEIGHTED-NORMAL needs less nodes, with more nodes being active, too. Both of those insights, in turn, could go against the current research direction of neutral genetic drift in CGP and could be topic of more in-depth analysis. Nonetheless, there is still the possibility of an interplay between active and inactive nodes. With nodes being active more frequently, it may give modified inactive nodes the opportunity to be tested. This would, in turn, be another argument in favour of neutral genetic drift.

ACKNOWLEDGMENTS

The authors would like to thank the German Federal Ministry of Education and Research (BMBF) for supporting the project SaMoA within VIP+.

REFERENCES

[1] Peter J. Bentley and Soo Ling Lim. 2017. Fault tolerant fusion of office sensor data using cartesian genetic programming. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 1–8.

[2] Borja Calvo, Josu Ceberio, and Jose A. Lozano. 2018. Bayesian Inference for Algorithm Ranking Analysis. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Kyoto, Japan) *(GECCO '18)*. Association for Computing Machinery, New York, NY, USA, 324–325. https://doi.org/10.1145/3205651.3205658

[3] M. Collins. 2005. Finding Needles in Haystacks is Harder with Neutrality. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation* (Washington DC, USA) *(GECCO '05)*. Association for Computing Machinery, New York, NY, USA, 1613–1618. https://doi.org/10.1145/1068009.1068282

[4] Henning Cui, Andreas Margraf, and Jörg Hähner. 2022. Refining Mutation Variants in Cartesian Genetic Programming. In *Bioinspired Optimization Methods and Their Applications*, Marjan Mernik, Tome Eftimov, and Matej Črepinšek (Eds.). Springer International Publishing, Cham, 185–200.

[5] Brian W. Goldman and William F. Punch. 2013. Length Bias and Search Limitations in Cartesian Genetic Programming. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation* (Amsterdam, The Netherlands) *(GECCO '13)*. Association for Computing Machinery, New York, NY, USA, 933–940. https://doi.org/10.1145/2463372.2463482

[6] Brian W. Goldman and William F. Punch. 2013. Reducing Wasted Evaluations in Cartesian Genetic Programming. In *Genetic Programming*, Krzysztof Krawiec, Alberto Moraglio, Ting Hu, A. Şima Etaner-Uyar, and Bin Hu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 61–72.

[7] Brian W. Goldman and William F. Punch. 2015. Analysis of Cartesian Genetic Programming's Evolutionary Mechanisms. *IEEE Transactions on Evolutionary Computation* 19, 3 (2015), 359–373. https://doi.org/10.1109/TEVC.2014.2324539

[8] Wolfgang Hörmann, Josef Leydold, and Gerhard Derflinger. 2004. *Discrete Distributions*. Springer Berlin Heidelberg, Berlin, Heidelberg, 215–241. https://doi.org/10.1007/978-3-662-05946-3_10

[9] Jakub Husa and Roman Kalkreuth. 2018. A Comparative Study on Crossover in Cartesian Genetic Programming. In *Genetic Programming*, Mauro Castelli, Lukas Sekanina, Mengjie Zhang, Stefano Cagnoni, and Pablo García-Sánchez (Eds.). Springer International Publishing, Cham, 203–219.

[10] Roman Kalkreuth. 2022. Phenotypic Duplication and Inversion in Cartesian Genetic Programming Applied to Boolean Function Learning. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Boston, Massachusetts) *(GECCO '22)*. Association for Computing Machinery, New York, NY, USA, 566–569. https://doi.org/10.1145/3520304.3529065

[11] Roman Kalkreuth, Günter Rudolph, and Andre Droschinsky. 2017. A New Subgraph Crossover for Cartesian Genetic Programming. In *Genetic Programming*, James McDermott, Mauro Castelli, Lukas Sekanina, Evert Haasdijk, and Pablo García-Sánchez (Eds.). Springer International Publishing, Cham, 294–310.

[12] Paul Kaufmann and Roman Kalkreuth. 2017. An Empirical Study on the Parametrization of Cartesian Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Berlin, Germany) *(GECCO '17)*. Association for Computing Machinery, New York, NY, USA, 231–232. https://doi.org/10.1145/3067695.3075980

[13] Paul Kaufmann and Roman Kalkreuth. 2020. On the Parameterization of Cartesian Genetic Programming. In *2020 IEEE Congress on Evolutionary Computation (CEC)*. 1–8. https://doi.org/10.1109/CEC48606.2020.9185492

[14] John K. Kruschke. 2013. Bayesian estimation supersedes the t test. *Journal of Experimental Psychology: General* 142, 2 (2013), 573.

[15] Andreas Margraf, Anthony Stein, Leonhard Engstler, Steffen Geinitz, and Jorg Hahner. 2017. An evolutionary learning approach to self-configuring image pipelines in the context of carbon fiber fault detection. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 147–154.

[16] Julian Miller and P. Thomson. 2000. Cartesian Genetic Programming. In *Proc. European Conference on Genetic Programming*, Vol. 1802. Springer, 121–132.

[17] Julian Miller, P. Thomson, and T. Fogarty. 1999. Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study. *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science* (10 1999).

[18] Julian F. Miller. 2011. *Cartesian Genetic Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-17310-3_2

[19] Julian Francis Miller. 2020. Cartesian genetic programming: its status and future. *Genetic Programming and Evolvable Machines* 21, 1 (01 Jun 2020), 129–168. https://doi.org/10.1007/s10710-019-09360-6

[20] Andrew J. Payne and Susan Stepney. 2009. Representation and structural biases in CGP. In *2009 IEEE Congress on Evolutionary Computation*. 1064–1071. https://doi.org/10.1109/CEC.2009.4983064

[21] Wenzel Pilar von Pilchau, David Pätzel, Anthony Stein, and Jörg Hähner. 2023. Deep Q-Network Updates for the Full Action-Space Utilizing Synthetic Experiences. In *2023 International Joint Conference on Neural Networks (IJCNN), to Appear*.

[22] David Pätzel. 2023. cmpbayes. https://github.com/dpaetzel/cmpbayes. commit = 4de0abc37ee28b35267db173d32bb96ca9e69236.

[23] Masanori Suganuma, Masayuki Kobayashi, Shinichi Shirakawa, and Tomoharu Nagao. 2020. Evolution of Deep Convolutional Neural Networks Using Cartesian Genetic Programming. *Evolutionary Computation* 28, 1 (03 2020), 141–163. https://doi.org/10.1162/evco_a_00253 arXiv:https://direct.mit.edu/evco/article-pdf/28/1/141/2020362/evco_a_00253.pdf

[24] Ali Torabi, Arash Sharifi, and Mohammad Teshnehlab. 2022. Using Cartesian Genetic Programming Approach with New Crossover Technique to Design Convolutional Neural Networks. *Neural Processing Letters* (01 Dec 2022). https://doi.org/10.1007/s11063-022-11093-0
[25] Andrew James Turner and Julian Francis Miller. 2015. Neutral genetic drift: an investigation using Cartesian Genetic Programming. *Genetic Programming and Evolvable Machines* 16, 4 (01 Dec 2015), 531–558. https://doi.org/10.1007/s10710-015-9244-6
[26] Vesselin K. Vassilev and Julian F. Miller. 2000. The Advantages of Landscape Neutrality in Digital Circuit Evolution. In *Evolvable Systems: From Biology to Hardware*, Julian Miller, Adrian Thompson, Peter Thomson, and Terence C. Fogarty (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 252–263.
[27] James Alfred Walker and Julian Francis Miller. 2008. The Automatic Acquisition, Evolution and Reuse of Modules in Cartesian Genetic Programming. *IEEE Transactions on Evolutionary Computation* 12, 4 (2008), 397–417. https://doi.org/10.1109/TEVC.2007.903549
[28] Ingo Wegener. 2005. *Complexity theory: exploring the limits of efficient algorithms.* Springer Science & Business Media.
[29] David White, James Mcdermott, Mauro Castelli, Luca Manzoni, Brian Goldman, Gabriel Kronberger, Wojciech Jaśkowski, Una-May O'Reilly, and Sean Luke. 2013. Better GP benchmarks: Community survey results and proposals. *Genetic Programming and Evolvable Machines* 14 (03 2013), 3–29.
[30] Dennis G. Wilson, Julian F. Miller, Sylvain Cussat-Blanc, and Hervé Luga. 2018. Positional cartesian genetic programming. *arXiv preprint arXiv:1810.04119* (2018).
[31] Tina Yu and Julian Miller. 2001. Neutrality and the Evolvability of Boolean Function Landscape. In *Genetic Programming*, Julian Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 204–217.
[32] Tina Yu and Julian Miller. 2002. Finding needles in haystacks is not hard with neutrality. In *European Conference on Genetic Programming*. Springer, 13–25.

A STATISTICAL MODEL TO COMPARE TWO CGP CONFIGURATIONS

We compare the *sets of number of iterations until a solution is found* (NIS) for two different CGP solutions (in our example, a WEIGHTED-CGP configuration compared to the baseline) for a given task. In the following, for each NIS, a random variable T_i with $i \in \{\text{WEIGHTED-CGP}, baseline\}$ is denoted.

For that purpose, a model inspired by the model described by Kruschke [14] is used. Our model differs, as it is based on two gamma distributions, one for each T_i—instead of the *Student's t-distribution* used by Kruschke [14]. We differ from the t-distribution, as it supports negative values. The mean NIS, however, can not be negative, so false assumptions would be made with the Student's t-distribution. Hence, we rely on a gamma distribution, as its support $x \in (0, \infty)$ is non-negative and can be mapped to our data more precisely.

The gamma distributions are parametrized for a mean parameter μ_i (the shape parameter) and a rate parameter β_i. As a prior for μ_i, a broad exponential distribution is used, which is parametrized such that 90 % of means lie in $[0, 1{,}500{,}000]$. Here, the upper bound is two times the maximum number of iterations for a CGP training. Furthermore, a mean NIS can not be negative or zero, so the exponential distribution is shifted by one hundredth of the minimum of the measured NIS. As for the rate parameter β_i, its prior is a uniform distribution chosen such that the variance of the distribution is at least l times, but at most u times the variance of the data. In accordance with the work from Kruschke [14], we choose l as one-thousandth, and u as thousand times the standard deviation of the pooled data.

We treat runs that were aborted after 750,000 iterations (i.e. runs that did not solve the task in time) as *censored data*. In our setting, we treat censored data as missing data, which is constrained to fall in the censored range. Therefore, we explicitly represent our censored data as a parameter $T_{i,cens}$. This value is sampled with the same β_i and μ_i parameters. However, all imputed values for the censored data will be greater than our maximal number of iterations.

In accordance with the description, we define a full model for two instances of $\{T_i, T_{i,cens}\}$ with $i \in \{\text{WEIGHTED-CGP}, baseline\}$ as follows:

$$T_i \sim \text{Gamma}(\mu_i, \beta_i) \tag{3}$$

$$T_{i,cens} \sim \text{Gamma}(\mu_i, \beta_i), T_{i,cens} > 750{,}000 \tag{4}$$

$$\mu_i - \frac{min(T_i)}{100} \sim Exp(\lambda_i) \tag{5}$$

$$\lambda_i = -\frac{ln(i - 0.9)}{u_\mu} \tag{6}$$

$$\beta_i \mid \mu_i \sim \mathcal{U}\left(\frac{\mu_i}{u \cdot Var(T_i)}, \frac{\mu_i}{l \cdot Var(T_i)}\right) \tag{7}$$

$$l = 1000^{-1}, \quad u = 1000, \quad u_\mu = 1{,}500{,}000 \tag{8}$$

To compare two CGP configurations, the parameters of interest are μ_i as well as their difference to estimate their ordering and magnitude of any effects. The model provides a posterior distribution for μ_i. As a result, they can be sampled, randomly paired and then subtracted to obtain a sample of the distribution of the difference of the mean values of WEIGHTED-CGP and the baseline. Given that the samples are large enough, it is possible to reason about the underlying distribution itself and to compute probabilities of the difference being positive or negative.

Please note that the *cmpbayes* [22] library uses Markov Chain Monte Carlo sampling to obtain the samples for the model. Here, four independent chains of 10,000 samples are sampled, each after a warm-up phase of 1,000. A more detailed description can be found in the work of Pilar von Pilchau et al. [21].

B RESULTS: TRAINING ITERATIONS

The results of each WEIGHTED-NORMAL and WEIGHTED-DAG configuration are stated in Table 5. Each configuration is compared to the baselines given in Table 2, with the same number of nodes. They extend Table 3 to judge the performance of WEIGHTED-NORMAL and WEIGHTED-DAG configurations.

Table 5: Results of Weighted-Normal and Weighted-Dag on the benchmark problems, given in intervals. We compute for each considered hyperparameter configuration $(w_{\max}, w_{\mathbf{step}})$ the 95 % *highest posterior density intervals* (HPDI) of the distributions of μ_1/μ_2 with $\mu_1 := \mu_{w_{\text{step}}, w_{\max}}$ and $\mu_2 := \mu_{\text{baseline}}$ where μ_1 and μ_2 are random variables corresponding to the respective *mean numbers of iterations until solution.* We report the lower and upper bounds of the 95% HPDI of the distribution of μ_1/μ_2. A 95 % HPDI interval $[l, u]$ can be read as $p(l \leq \mu_1/\mu_2 \leq u) = 95\,\%$, that is, *the ratio between the modified algorithm with the given parametrization and the baseline lying between the two bounds has a probability of 95 %*

		Weighted-Normal $w_{\max}$				Weighted-Dag $w_{\max}$			
	w_{step}	250	500	750	1,000	250	500	750	1,000
Parity	10	[0.7, 3.5]	[0.9, 6.6]	[1.3, 8.2]	[0.8, 4.8]	[0.3, 1.3]	[0.5, 1.8]	[0.6, 2.3]	[0.5, 2.0]
	25	[0.4, 2.0]	[0.6, 4.0]	[0.4, 1.9]	[1.0, 6.7]	[0.3, 1.3]	[0.5, 1.7]	[0.7, 2.4]	[0.4, 1.7]
	50	[0.7, 3.9]	[1.2, 8.0]	[0.3, 1.4]	[0.9, 6.1]	[0.5, 2.1]	[0.5, 2.2]	[0.7, 2.7]	[0.5, 1.8]
	100	[0.5, 2.1]	[0.6, 4.1]	[0.5, 2.8]	[0.5, 2.4]	[0.5, 2.3]	[0.6, 2.0]	[0.7, 2.8]	[0.7, 2.7]
Encode	10	[0.6, 2.4]	[0.8, 4.6]	[0.8, 3.1]	[1.0, 4.3]	[0.7, 2.0]	[0.5, 1.3]	[0.7, 1.6]	[0.5, 1.0]
	25	[0.8, 4.3]	[1.0, 4.8]	[0.5, 1.7]	[0.7, 3.2]	[0.7, 1.7]	[0.5, 1.3]	[0.6, 1.6]	[0.5, 1.1]
	50	[0.6, 2.6]	[0.5, 1.4]	[0.8, 3.4]	[1.6, 6.6]	[0.6, 1.6]	[0.7, 1.7]	[0.6, 1.6]	[0.5, 1.4]
	100	[0.5, 1.6]	[0.5, 1.8]	[0.4, 1.4]	[0.6, 2.1]	[0.7, 1.7]	[0.6, 1.5]	[0.7, 1.5]	[0.6, 1.6]
Decode	10	[0.8, 1.5]	[1.0, 2.1]	[0.8, 1.5]	[1.2, 2.5]	[0.7, 1.2]	[0.6, 1.0]	[0.6, 1.0]	[0.8, 1.3]
	25	[0.9, 1.6]	[0.7, 1.3]	[0.9, 1.5]	[1.0, 2.0]	[0.8, 1.4]	[0.8, 1.3]	[0.7, 1.3]	[0.8, 1.3]
	50	[0.8, 1.4]	[1.0, 1.9]	[1.1, 1.7]	[1.2, 2.5]	[0.7, 1.1]	[0.7, 1.1]	[0.7, 1.2]	[0.6, 1.2]
	100	[0.8, 1.5]	[0.8, 1.5]	[0.9, 1.7]	[1.1, 2.3]	[0.7, 1.3]	[0.8, 1.4]	[0.8, 1.7]	[0.9, 1.5]
Multiply	10	[0.3, 1.2]	[0.5, 1.7]	[0.5, 1.9]	[0.9, 3.5]	[0.6, 1.3]	[0.9, 2.0]	[0.7, 1.5]	[0.8, 1.8]
	25	[0.4, 1.7]	[0.7, 2.7]	[0.7, 2.4]	[0.4, 1.4]	[0.9, 2.1]	[0.6, 1.1]	[0.7, 1.3]	[0.8, 2.2]
	50	[0.3, 0.7]	[0.8, 2.8]	[0.7, 2.4]	[0.3, 1.2]	[0.7, 1.4]	[0.7, 1.3]	[0.7, 1.6]	[0.8, 1.9]
	100	[0.4, 1.4]	[0.7, 2.5]	[0.4, 1.8]	[0.8, 3.0]	[0.7, 1.4]	[0.6, 1.2]	[0.8, 1.5]	[0.7, 1.3]