Positional Bias Does Not Influence Cartesian Genetic Programming with Crossover

Henning Cui(✉), Michael Heider, and Jörg Hähner

University of Augsburg, 86159 Augsburg, Germany
henning.cui@uni-a.de

Abstract. The recombination operator plays an important role in many evolutionary algorithms. However, in Cartesian Genetic Programming (CGP), which is part of the aforementioned category, the usefulness of crossover is contested. In this work, we investigate whether CGP's positional bias actually influences the usefulness of the crossover operator negatively. This bias describes a skewed distribution of CGP's active and inactive nodes, which might lead to destructive behaviours of standard recombination operators. We try to answer our hypothesis by employing one standard CGP implementation and one without the effects of positional bias. Both versions are combined with one of four standard crossover operators, or with no crossover operator. Additionally, two different selection methods are used to configure a CGP variant. We then analyse their performance and convergence behaviour on eight benchmarks taken from the Boolean and symbolic regression domain. By using Bayesian inference, we are able to rank them, and we found that positional bias does not influence CGP with crossover. Furthermore, we argue that the current research on CGP with standard crossover operators is incomplete, and CGP with recombination might not negatively impact its evolutionary search process. On the contrary, using CGP with crossover improves its performance.

Keywords: Cartesian Genetic Programming · CGP · Crossover · Recombination · Positional Bias

1 Introduction

Cartesian Genetic Programming (CGP) is a form of *Genetic Programming* (GP), based on *directed, acyclic and feed–forward graphs* whose nodes are arranged in a two-dimensional grid. CGP mainly relies on mutation and selection operators for its evolutionary search process.

For GP, numerous different crossover algorithms exist [17] but its overall utility is doubted [26,32]. Similarly, since its inception in 1999 by Miller [21], the use of a crossover operator in CGP is an active research topic. To this day, its advantage is questioned as an effective crossover operator relies on extensions to the CGP formula and highly depends on the specific use case [10]. Furthermore, it is still unclear as to why crossover raises issues in the context of CGP [4,11, 21,22].

Our hypothesis is to assume that the *positional bias* influences CGP with a recombination algorithm, an effect which potentially limits CGP to fully explore its search space [8]. It describes a problem in which nodes contributing to an output have a non-uniform distribution in the graph. This in turn might lead to problems for standard recombination operators like the point-, multi-n-, or uniform-crossover. These operators do not consider semantic structures, like such non-uniform distributions of nodes. To counteract positional bias, Cui et al. [5] introduced an operator to fully eliminate it. We will take advantage of this operator to test our hypothesis. If positional bias plays a role in CGP's issues with crossover, using a recombination algorithm with this extension might boost its performance or lead to interesting results. By empirically analysing its performance and behaviour, a better understanding of this overall issue could be gained.

We provide a quick overview of the core principles of CGP, reintroduce the concept of positional bias and how to mitigate it in the following Sect. 2. Afterwards, Sect. 3 gives an overview of related work. We will then explain our hypothesis and state our research questions in Sect. 4. Subsequently, we discuss our experimental setup in Sect. 5, and the results and its discussion is given in Sect. 6. At last, Sect. 7 summarizes our findings and discusses future research directions.

2 Cartesian Genetic Programming

In this section, we reintroduce the supervised learning algorithm called *Cartesian Genetic Programming* (CGP) [21]. An additional emphasis will be made on CGP's *positional bias* and one method on how to reduce it.

2.1 Representation

Standard CGP is represented by a directed, acyclic and feed-forward graph. It contains partially connected *nodes*, which are arranged in a grid. This allows for an arbitrary amount of program inputs and any desired amount of outputs. Originally, CGP used a $c \times r$ grid with $c \in \mathbb{N}_+$ and $r \in \mathbb{N}_+$. With today's standards, a CGP model consists of only one row for most applications [20].

The set of nodes in a graph defined by CGP can be divided into input-, output- and computational nodes. *Input nodes* receive the program input directly, and relay these values to computational and/or output nodes. Similarly, *output nodes* define the program's final output by redirecting the output of an input- or computational node. They are represented by a single connection gene. *Computational nodes* are represented by one function- and m connection genes, with $m \in \mathbb{N}_+$ being the maximum *arity* of one function in the whole function set. The function gene encodes the computational function of a node. It gets its input from previous nodes via its connection genes. They define a path between a previous and the current node.

Computational nodes can also be categorized into *active* and *inactive* nodes. The former are part of a path to one or multiple output nodes—hence they contribute to the program's final output. Inactive nodes are not part of a path to output nodes and do not contribute to an output. Still, by allowing such inactive nodes to persist throughout the training process, it improves CGP's evolutionary search through neutral genetic drift [18,29].

An illustrative example of a graph defined by CGP can be seen in Fig. 1. It depicts the genotype with two input-, three computational- and one output node. Active nodes are drawn with a solid line, while inactive nodes are marked by dashed lines. It contains two input-, three computational-, and one output node. 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 is not part of a path to an output node, and is therefore inactive.

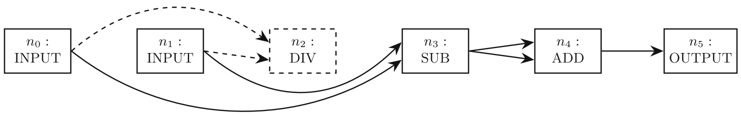

Fig. 1. Example graph defined by a CGP genotype. The dashed node and connections are *inactive* due to not contributing to the output.

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, and additional input and output nodes corresponding to the given learning task. Furthermore, to improve readability and clarity, a graph defined by this representation will be called Standard for the remainder of this work.

2.2 Positional Bias

CGP enforces a feed-forward grid. Goldman and Punch [8] found that this leads to *positional bias*. This issue describes a non-uniform distribution of nodes being active throughout the whole graph. Computational nodes near input nodes have a *higher chance* of being active compared to the ones near the output nodes. This negatively impacts CGP's evolutionary search process, as it increases the difficulty to solve certain tasks and decreases its performance [9,23].

The reason behind positional bias is CGP's grid-structure combined with feed-forward connections. Nodes near input nodes have more nodes to their right. This means, more nodes can mutate a connection to them—which in turn increases the probability of them becoming active. Nodes near output nodes, however, have less nodes to their right. Hence, their chance to be part of a path to an output node decreases, too.

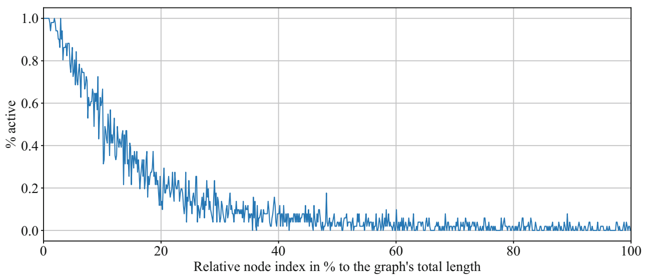

Fig. 2. Distribution of active nodes over a graph defined by CGP.

Figure 2 shows a visualization of positional bias for a better understanding. It depicts a plot of 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 computational node in its graph is given. The y-axis indicates its probability of being active. A clear example of positional bias can be seen here. About the first quarter of computational nodes have a (very) high probability of being active, and the remaining node's probabilities are minimal.

One solution to solve positional bias is to use the *levels back* parameter from the original CGP [19], which restricts the connectivity of a node. However, its usage is not recommended as it negatively impacts CGP's performance [20]. This is why we will also refrain from using it.

To fully mitigate positional bias, Cui et al. [5] introduced the *Equidistant-Reorder* (E-Reorder) operator, which was inspired by the original *Reorder* operator from Goldman and Punch [8]. It works by generating a new genome G', which is initialized by inheriting all input and output nodes from the original genome G because these do not change their positions. In addition, G' reserves enough space for computational nodes to fit all computational nodes of G. Afterwards, all active nodes from G are placed in the same sequence and equidistantly apart into G'. All inactive nodes from G are then placed in the same sequence into the next genome location of G' where no active node was placed. Finally, G is replaced by G'. Because G' and G have the same ordering of active nodes, the genotype of G' changes but the phenotype stays the same. As a result, Cui et al. were able to fully eliminate positional bias, which lead to an increase in CGP's performance. For a more in-depth algorithmic explanation, we refer to their work [6].

To improve readability, we will refer to a CGP version with the E-Reorder operator as E-Reorder.

3 Related Work

In the context of CGP, the effects and behaviours of crossover operators have been investigated by various previous authors. Regarding the influence of positional bias on CGP's crossover operator, to the best of our knowledge, the work of Cui et al. [6] is the only one who investigated it. In this work, we expand their perspective and analyses.

Still, various other previous works laid out the foundation to our investigation. The earliest work regarding CGP and crossover was done by Miller [21]. While his original work did not argue against using crossover, he later claimed that the one-point crossover showed a disruptive behaviour [20].

Cai et al. [2] argued that CGP is not positional independent. This means, CGP's components and their workings depend on their position in the graph. Problems may arise now with crossover, as they do not consider such dependencies. Hence, useful structures are destroyed. To counteract this problem, the authors introduced a new crossover operator which also considers such dependencies. However, our work differs from theirs as they did not consider the distribution of active nodes or positional bias in their work.

Kalkreuth et al. [13] also argued that simply swapping genes randomly does not improve CGP. In their work, they introduced a subgraph-crossover operator which only recombines active nodes. A comprehensive study was done [11] and they showed a beneficial behaviour in two problem domains. However, contrary to this work, they also did not consider positional bias.

Similarly to the just described work, Husa and Kalkreuth [10,11] introduced the block-crossover, which swap blocks of consecutive active nodes. It is based upon the subgraph crossover [13] and embedded CGP [15].

Normally, CGP's representation is integer based, similarly to standard genetic programming. Clegg et al. [4] changed it to a floating-point based representation. This allowed them to introduce specialized crossover operators. Wilson et al. [33] based their work upon this aspect and further analysed a floating-point based representation of CGP.

One use case of CGP is that image processing as filter pipelines can be evolved with it. Slaný et al. [25] used the original grid-structure of CGP for this task. They also included single-point and multi-point crossover without modifying CGP in their studies. By observing the fitness landscape of their solutions, the authors concluded that the single-point crossover improved the evolutionary search.

Another domain specific approach was done by Torabi et al. [28]. They used CGP in the context of neural architecture search and adapted a specialized crossover mechanism to design convolutional neural networks.

A more recent method, developed from Kalkreuth [12], describes an operator which recombines only active nodes.

4 Positional Bias and Crossover

We originally assumed that *positional bias* might influence CGP's issues, as discovered in previous studies (cf. Sect. 3), with recombination operators. Our rationale is that active nodes accumulate near input nodes, while inactive nodes concentrate near output nodes. As such, because active nodes contribute to the program's output, their clustering near input nodes can be viewed as important node–structures. However, traditional crossover operators like the $n-$point or uniform crossover do not consider such clusters. It is possible that, by applying crossover, *important semantic structures* might get destroyed. Other methods from Husa and Kalkreuth [10], Cai et al. [2] or Kalkreuth et al. [11–13] do however consider such structures. This may be one explanation for their performance gain.

To summarize: By applying traditional crossover operators with Standard, important structures of active nodes near the input nodes might get destroyed. Such clusterings are due to positional bias. With E-Reorder, positional bias can be fully circumvented. As a result, by applying the E-Reorder operator *before* crossover, these stated negative effects might be weakened, too.

Thus, in this work we will focus on answering two research questions:

Q1: Has positional bias an effect on CGP with a crossover operator?
Q2: Is there a difference in behaviour when different CGP variants with and without crossover are analysed?

5 Experimental Design

In order to gauge the effects of positional bias on CGP with crossover, we conducted an *empirical study.* We give a brief introduction into Bayesian inference to evaluate and rank different CGP configurations. They are used to compare multiple CGP configurations, which we use for our statistical analysis. Afterwards, we describe CGP's configuration and the benchmarks used[1].

5.1 Bayesian Data Analysis

To answer our research questions, a fair comparison of algorithms and a qualitatively sound evaluation must be ensured. For this task each CGP variant must be ranked to find the best solution. For Boolean benchmarks we only examine the number of *training iterations until a solution is found* (*I2S*). Thus, for these types of benchmarks, algorithms are ranked according to their *I2S*. Considering symbolic regression benchmarks, the final goal is to minimize their fitness value—which is why they are ranked according to their *final test fitness value*. For both benchmark types these numbers cannot be negative. Hence, other common distributions such as *Student's t distributions* can not be expected to model the data well [16]. This is why we performed a *Bayesian data analysis* for the posterior distributions of our results[2]. The model to compare the algorithms is based on the *Plackett-Luce model* described by Calvo et al. [3]. 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 the *I2S* of Boolean benchmarks, we report the 95 % *highest posterior density intervals* (HPDI) of the distribution of μ_{config}, where μ_{config} is

[1] The source code can be found at: https://github.com/CuiHen/CGP_with_Crossover_Strategies.

[2] We utilized the Python library *cmpbayes* [24] for all statistical models.

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 $p(l \leq \mu_{config} \leq u) = 95\,\%$. This means, the probability of the algorithms results lying between the bounds $l \in \mathbb{N}_+$ and $u \in \mathbb{N}_+$ is 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 [16] and Pätzel [24].

5.2 CGP Variants and Their Configuration

To answer our hypothesis, a broad set of configurations must be evaluated and compared. For basic baselines, we use the following two configurations: No crossover, Standard with a $(1 + 4)$ Evolutionary Strategy (ES); and no crossover, E-Reorder with a $(1 + 4)$-ES. Both variants are commonly found in literatures—that means, CGP uses an elitist $(\mu + \lambda)$-ES with $\mu = 1$ and $\lambda = 4$, and no crossover [20,29].

These baselines are then compared with the combinations of different components. We employ two different CGP variants: Standard and E-Reorder. For the selection method, a *standard tournament selection with elitists* or $(\mu + \lambda)$-ES is used. Considering the crossover operators, we tested four operators: *1-point*, *2-point*, *3-point*, and *uniform* crossover. A crossover rate of 0.9 was used for all configurations. Furthermore, the option of *no recombination* must also be evaluated. Including our baselines, this leads to 22 different algorithmic combinations. To further clarify the combination of non-baseline modules, we list the components in Table 1.

Table 1. Different components from which a CGP configuration is created. We tested the combination of all categories and two baselines.

Crossover Operator	CGP Variant	Selection Strategy
1-point crossover	$(\mu + \lambda)$-ES	Standard
2-point crossover	Tournament selection with elitists	E-Reorder
3-point crossover		
Uniform crossover		
No crossover		

To mutate the genotypes, we use *Single* [9]. This operator works by mutating random genes until a gene corresponding to an active node is mutated. This enforces a measurable change in the phenotype—compared to the standard point mutation which may only mutate inactive nodes, which makes it impossible to gauge the quality of the genotypical change [7,8]. Furthermore, it has the benefit that it does not need a mutation probability and achieves similar results compared to a standard point mutation.

For the baselines, their number of computational nodes N must be optimized. Considering other CGP configurations, we additionally optimized μ and λ when the $(\mu + \lambda)$-ES is used. For tournament selection, we included the tournament size, population size, and the number of elitists into the hyperparameter optimization process. For each parameter, we investigated the following possibilities: $N \in \{50, 100, 150, \cdots, 2000\}$, $\mu \in \{2, 4, 6, 8, 10\}$, $\lambda \in \{10, 12, 14, \cdots, 50\}$, tournament size $\in \{2, 4, 6, 8\}$, population size $\in \{10, 12, 14, \cdots, 50\}$, number of elitists $\in \{2, 4, 6, 8, 10\}$.

To find the best hyperparameters, we used a Tree-structured Parzen Estimator[3]. All configurations were tested 20 times with independent repetitions and completely random seeds. For our final results, each CGP version used the best set of hyperparameters found for a given benchmark and were run again for 50 times with independent repetitions and different random seeds.

5.3 Benchmarks

To evaluate our hypothesis, Boolean and symbolic regression benchmarks were tested. 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 [31]. However, it was commonly used as a benchmark in literature [14,21]. This is why 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 and recommended by White et al. [31]. For these benchmark problems we used their standard Boolean function set: *AND*, *OR*, *NAND* and *NOR*. As their fitness function, we employed a standard one, too. It is defined by the ratio of correctly mapped inputs.

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*)*.

In terms of *symbolic regression benchmarks* we again adhered to the recommendations from the GP community [31] and previous works [11]. Four different benchmarks were used: *Keijzer-6*, *Koza-3*, *Nguyen-7* and *Pagie-1*. Their definitions are shown in Table 2.

The function set consists of eight mathematical functions: addition, subtraction, multiplication, protected division, sine, cosine, natural logarithm and the

[3] For the hyperparameter search, we utilized the Python library Optuna [1].

Table 2. Symbolic regression benchmarks used. $U[a, b, c]$ means that c uniform random samples are drawn from a to b, inclusive. $E[a, b, c]$ defines a grid of points from a to b, with c being the spacing.

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

exponential function. As for the fitness function, the *mean absolute error* over the whole benchmark with n entries was used:

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. Again, we only limit their budged based on training iterations. In this way, all configurations have a chance of convergence—which might not be the case when we limit their budged depending on their population size.

6 Evaluation

With our experimental setup explained, we now focus on answering our research questions. On that account, we report *the top three parametrizations* regarding the performance. We do not list all combinations of modules per benchmark, as there are a total of 22 different configurations. Combined with eight different benchmarks, presenting all results would take up too much space. For a complete list of all results, we refer to *zenodo*: https://doi.org/10.5281/zenodo.10830014. Furthermore, in the complete list we report the mean fitness, standard deviation of the fitness, number of active nodes, total number of nodes, population size, number of elitists, tournament size, success rate, HPDI and the probability of one solution being the best.

6.1 Performance of Different Module Combinations

We report selected results on Boolean and symbolic regression benchmarks in Table 3 and Table 4. Again, we refer to https://doi.org/10.5281/zenodo.10830014 for a complete view of our results.

Discussion: Boolean Benchmarks. For the Boolean benchmarks, tournament selection does not perform well in this setting. It leads to the worst results on all benchmarks except for Parity. On Multiply, all configurations with this selection method even reaches a success rate of zero.

Regarding our original research question Q1, the positional bias *does probably not affect CGP* in the context of Boolean benchmarks. On the contrary,

Table 3. Selected results on Boolean benchmarks. We report the mean fitness ($mean(I2S)$), HPDI, number of active nodes (# active), total number of nodes (# nodes), and the probability of one solution being the best (p_{best}). Entries are sorted according to p_{best}.

CGP variant	$mean(I2S)$	HPDI	# active	# nodes	p_{best}
Parity					
1-p. cr., Standard, $(4+50)$	137	$(98, 189)$	48	650	0.07
2-p. cr., E-Reorder, $(8+38)$	690	$(416, 1146)$	431	800	0.07
uni. cr., Standard, $(2+48)$	136	$(101, 182)$	36	450	0.06
...	...	...	...	...	...
Encode					
2-p. cr., Standard, $(2+44)$	2,192	$(1878, 2541)$	63	150	0.13
1-p. cr., Standard, $(2+50)$	2,564	$(2146, 3044)$	87	400	0.12
no cr., Standard, $(2+50)$	2,288	$(1919, 2718)$	77	300	0.11
...	...	...	...	...	...
Decode					
2-p. cr., Standard, $(8+50)$	3,229	$(2871, 3643)$	126	250	0.21
3-p. cr., Standard, $(8+46)$	3,415	$(2997, 3884)$	162	450	0.15
no cr., E-Reorder, $(8+50)$	3,414	$(2946, 3960)$	208	250	0.15
...	...	...	...	...	...
Multiply					
1-p. cr., Standard, $(2+44)$	35,519	$(29547, 42588)$	96	350	0.15
no cr., E-Reorder, $(2+50)$	53,593	$(41036, 69020)$	247	300	0.13
uni. cr., Standard, $(2+48)$	42,120	$(34939, 50555)$	95	350	0.13
...	...	...	...	...	...

in most cases Standard *with recombination* seems to outperform E-Reorder *with no crossover* both in terms of mean fitness and p_{best}; while E-Reorder outperforms Standard when both variants *do not use crossover*. For Multiply, Standard with an $(\mu+\lambda)$-ES finds solutions with less active nodes compared to E-Reorder with $(\mu+\lambda)$-ES, which indicates a more compact solution with less redundant computations. This is contrasted by the general trend that the Standard baseline normally has more active nodes compared to the E-Reorder baseline. Regarding the other benchmarks, such a behaviour cannot be observed. There is only a slight tendency towards Standard with crossover having less active nodes than E-Reorder.

Interestingly, a 1-point or 2-point crossover leads to the best results in most cases. We believe that, in this setting, Standard with a 1- or 2-point crossover should drastically change its geno- and phenotype. In addition, Boolean benchmarks display a *deceptive* fitness landscape [30]. In this context, this means that a multitude of different solutions lead to the same fitness value. Hence, due to

Table 4. Selected results on symbolic regression benchmarks. We report the mean *I2S* (*mean*(*I2S*)), mean fitness ($mean(fit)$), number of active nodes (# active), total number of nodes (# nodes), the success rate (s-rate), and the probability of one configuration being the best in terms of test fitness (p_{best}). Entries are sorted according to p_{best}.

CGP variant	*mean*(*I2S*)	$mean(fit)$	# active	# nodes	s-rate	p_{best}
Keijzer-6						
uni. cr., E-Reorder, $(10+40)$	2	0.000	9	50	1.00	0.14
2-p. cr., E-Reorder, $(10+48)$	1	0.000	8	50	1.00	0.10
3-p. cr., E-Reorder, $(8+32)$	2	0.000	9	50	1.00	0.09
...	...	...	...	...	...	...
Koza-3						
3-p. cr., E-Reorder, tour.	1059	0.003	19	50	1.00	0.12
uni. cr., E-Reorder, tour.	323	0.006	21	50	1.00	0.08
uni. cr., E-Reorder, $(10+30)$	5893	0.006	16	50	1.00	0.08
...	...	...	...	...	...	...
Nguyen-7						
1-p. cr., E-Reorder, tour.	14063	0.009	31	50	1.00	0.08
1-p. cr., Standard, tour.	24621	0.010	14	50	1.00	0.07
uni. cr., Standard, tour.	28423	0.009	16	150	1.00	0.07
...	...	...	...	...	...	...
Pagie-1						
1-p. cr., Standard, tour.	482817	0.033	32	100	0.08	0.08
uni. cr., Standard, $(6+46)$	459114	0.034	97	850	0.12	0.08
no cr., Standard, $(8+50)$	451652	0.036	91	450	0.16	0.08
...	...	...	...	...	...	...

great changes in the genotype introduced by recombinations, more regions in the fitness landscape might be explored. This, in turn, might improve its evolutionary search process—and lead to less *I2S*.

Discussion: Symbolic Regression Benchmarks. Compared to Boolean benchmarks, a tournament selection does not completely impair its performance. Still, a CGP configuration with crossover leads to the best results in all cases again.

Additionally, for the first three out of four benchmarks, E-Reorder with crossover probably leads to the best results in terms of mean *I2S* and mean fitness value. The greatest difference in *I2S* can be seen with Nguyen-7. The lowest *I2S* is achieved with 2-point crossover, E-Reorder and tournament selection. This configuration achieves a mean *I2S* of 2,078 with a mean test fitness value of 0.011—compared to its counterpart: 2-point crossover, Standard and tournament selection, which needs 19,000 iterations and achieves a mean fitness of 0.010.

According to the test fitness value and *I2S*, Pagie-1 is the hardest symbolic regression benchmark for CGP. When only its mean test fitness is considered, Standard with or without recombination always outperforms E-Reorder. Furthermore, 1-point crossover with Standard and tournament selection has the highest p_{best}. This notion contradicts our hypothesis. However, when the success rate combined with its mean *I2S* is considered, a 2-point crossover E-Reorder variant comes on top with a success rate of 0.16.

All in all, E-Reorder with crossover should lead to better results for symbolic regression benchmarks. However, to answer Q1 in the context of symbolic regression benchmarks, the *positional bias does probably not* affect CGP with crossover.

6.2 Convergence Behaviour

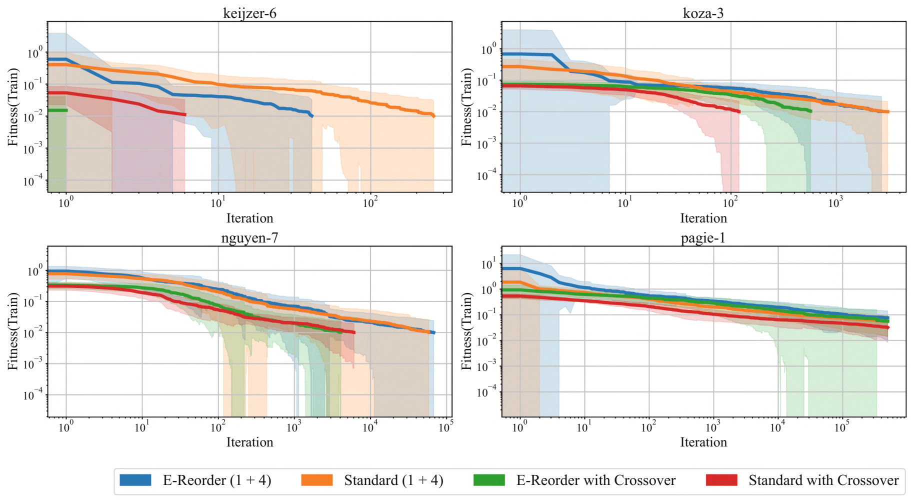

Fig. 3. Convergence plots for each regression benchmark. For better visualization, the x- and y-axis have a logarithmic scale and we cut them off after they reach a mean fitness of less than 0.01. The shaded area indicates their respective standard deviation.

To better understand the effects of crossover on CGP, we depict convergence plots for symbolic regression. We also try to classify their convergence behaviour according to Stegherr et al. [27]. Please note that we do not include convergence plots for Boolean benchmarks. Because of their *deceptive* fitness landscape [30], they all show the exact same convergence behaviour by design: *Fast to Slow*. Hence, no additional value can be gained by including them.

Considering the plots for symbolic regression benchmarks, including the behaviour of all 22 configurations will make it impossible to analyse. This is why we will only include the behaviour of four CGP configurations: No Crossover, Standard, $(1+4)$-ES; No Crossover, E-Reorder, $(1+4)$-ES; The best Standard and E-Reorder configuration with crossover. For these plots, we averaged the convergence of 50 runs. Furthermore, to easier see the differences, the x- and y-axis have a logarithmic scale. In addition, we included their respective standard deviation (shaded area).

As can be seen in Fig. 3, they all depict a similar convergence behaviour—which can be classified into *Fast to Slow*. Within the first few iterations, a relatively low 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. Nevertheless, when only convergence plots are examined, all algorithms behave very similar. Thus, it can be concluded that the behaviour of CGP does not change when a crossover method is included.

6.3 General Discussion

An unusual observation in our work is the general lack of performance issues for Standard with crossover. Contrary to some previous works, we could not replicate the negative impact of recombination on CGP. We believe that this might be due to several reasons.

One of the earliest works on CGP with crossover was done by Miller [21]. His work differs greatly from ours, as he used a $c \times r$ grid, with $c > 1$ and $r > 1$. Compared to our work, we used $r = 1$ which is also the recommended number of rows nowadays [20]. Furthermore, he only considered tournament selection, which, in our setting, lead to the worst results in most cases for Boolean benchmarks independent of using crossover or not.

Clegg et al. [4] also argued against the use of standard crossover techniques. They analysed CGP's convergence behaviour, tested on a single symbolic regression problem. Four crossover operators were compared against a standard CGP implementation, and they found that these recombination operators increased the *I2S*. However, according to their description, they did not optimize CGP's hyperparameters or tested different selection operators. Additionally, their statement that crossover negatively impairs CGP is only based on one fairly simple problem, which can also be criticised.

To the best of our knowledge, Husa and Kalkreuth [10] were the first to present a comparative study on crossover for CGP. In their setting, they found that CGP with crossover can outperform the $(1+\lambda)$-ES without crossover for an arbitrary configuration. When the hyperparameters for each configuration are optimized, though, the $(1+\lambda)$-ES without crossover always outperforms CGP with crossover. Nonetheless, they only tested Boolean benchmarks and combined recombination operators with tournament selection. In our work, this selection operator impaired CGP's performance on Boolean benchmarks which is why we have to treat their conclusion with reservation.

Considering our results, we believe that the use of a recombination operator does not always impair CGP's performance. On the contrary, with the right configuration and parametrization, CGP might even profit from it. As we presented in Sect. 6.1, Standard with crossover is almost always able to achieve better results compared to their baseline. Even when their convergence behaviour is visually analysed in Sect. 6.2, we do not see a negative effect caused by crossover.

To finally give a definite answer to our research questions:

Q1 & A1: Has positional bias an effect on CGP with a crossover operator employed? *No*, positional bias does not affect CGP with a recombination algorithm in our setting.

Q2 & A2: Is there a difference in behaviour when different CGP variants with and without crossover are analysed? *No*, there is no difference in convergence behaviour in our setting.

7 Conclusion

In this work, we investigated the effect of the *positional bias* on CGP with a *crossover operator employed*. On that account, we compared two different CGP versions, one with and one without positional bias. A standard CGP variant (Standard) was used, which suffers from the negative effects of positional bias. To fully mitigate this problem, we employed a CGP variant with the *Equidistant-Reorder* operator (E-Reorder). These variants were then evaluated with two different selection strategies, four different recombination algorithms, or with no crossover operator employed—leading to a comparison of 22 unique algorithmic configurations of CGP. To gauge the effects of positional bias on crossover, four Boolean and four symbolic regression benchmarks were used. We optimized the hyperparameters for each combination of benchmark and CGP version, ranked and discussed the optimized configurations.

In our testing, we found that positional bias has *no effect* on CGP with crossover. On the contrary, by adding a recombination algorithm, the performance of Standard could be improved in all cases. While previous works suggested that recombination does not or negatively impact CGP, we believe that this decision was made too early. Using a tournament selection, for example, lead to the worst results on Boolean benchmarks independent of using crossover or not. However, most authors used CGP with tournament selection and crossover on Boolean benchmarks without considering other selection operators in the past. This might lead to an overall worse performance—and to draw a conclusion that CGP does not profit from crossover.

As for future works, other selection and crossover operators should be paired with CGP. Furthermore, more benchmark categories should be tested before giving a final answer to the question: Does crossover impact CGP negatively? In addition, positional bias does probably not influence CGP with crossover negatively, according to our results. However, more configurations can be tested. We used an Equidistant-Reorder method, which fully eliminates positional bias. Nevertheless, more extensions should be tested, like the standard Reorder method [8].

It would also be possible to use the extension called DAG [8], which allows arbitrary node connections as long as no cycles form. This graph can be reordered into a feed-forward graph without changing the sequence of any operations. In this way, the positional bias is mitigated and crossover operators can be applied and tested, too.

Acknowledgements. The authors would like to thank the German Federal Ministry of Education and Research (BMBF) for supporting the project SaMoA within VIP+ (grant number 03VP09291).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Akiba, T., Sano, S., Yanase, T., Ohta, T., Koyama, M.: Optuna: a next-generation hyperparameter optimization framework. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2019)
2. Cai, X., Smith, S.L., Tyrrell, A.M.: Positional independence and recombination in cartesian genetic programming. In: Collet, P., Tomassini, M., Ebner, M., Gustafson, S., Ekárt, A. (eds.) Genetic Programming, pp. 351–360. Springer, Heidelberg (2006). https://doi.org/10.1007/11729976_32
3. Calvo, B., Ceberio, J., Lozano, J.A.: Bayesian inference for algorithm ranking analysis. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO 2018), pp. 324–325. Association for Computing Machinery, New York (2018). https://doi.org/10.1145/3205651.3205658
4. Clegg, J., Walker, J.A., Miller, J.F.: A new crossover technique for cartesian genetic programming. In: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO 2007), pp. 1580–1587. Association for Computing Machinery, New York (2007). https://doi.org/10.1145/1276958.1277276
5. Cui, H., Margraf, A., Hähner, J.: Equidistant reorder operator for cartesian genetic programming. In: van Stein, N., Marcelloni, F., Lam, H.K., Cottrell, M., Filipe, J. (eds.) Proceedings of the 15th International Joint Conference on Computational Intelligence - ECTA, 13–15 November 2023, Rome, pp. 64 – 74 (2023). https://doi.org/10.5220/0012174100003595
6. Cui, H., Margraf, A., Heider, M., Hähner, J.: Towards understanding crossover for cartesian genetic programming. In: van Stein, N., Marcelloni, F., Lam, H.K., Cottrell, M., Filipe, J. (eds.) Proceedings of the 15th International Joint Conference on Computational Intelligence - ECTA, 13–15 November 2023, Rome, pp. 308 – 314 (2023). https://doi.org/10.5220/0012231400003595
7. Goldman, B.W., Punch, W.F.: Analysis of cartesian genetic programming's evolutionary mechanisms. IEEE Trans. Evolution. Comput. **19**(3), 359–373 (2015). https://doi.org/10.1109/TEVC.2014.2324539
8. Goldman, B.W., Punch, W.F.: Length bias and search limitations in cartesian genetic programming. In: Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation (GECCO 2013), pp. 933–940. Association for Computing Machinery, New York (2013). https://doi.org/10.1145/2463372.2463482

9. Goldman, B.W., Punch, W.F.: Reducing wasted evaluations in cartesian genetic programming. In: Krawiec, K., Moraglio, A., Hu, T., Etaner-Uyar, A.Ş., Hu, B. (eds.) Genetic Programming, pp. 61–72. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37207-0_6
10. Husa, J., Kalkreuth, R.: A comparative study on crossover in cartesian genetic programming. In: Castelli, M., Sekanina, L., Zhang, M., Cagnoni, S., García-Sánchez, P. (eds.) Genetic Programming, pp. 203–219. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77553-1_13
11. Kalkreuth, R.: A comprehensive study on subgraph crossover in cartesian genetic programming. In: Proceedings of the 12th International Joint Conference on Computational Intelligence (IJCCI 2020) - ECTA, pp. 59–70. INSTICC, SciTePress (2020). https://doi.org/10.5220/0010110700590070
12. Kalkreuth, R.: Towards discrete phenotypic recombination in cartesian genetic programming. In: Rudolph, G., Kononova, A.V., Aguirre, H., Kerschke, P., Ochoa, G., Tušar, T. (eds.) Parallel Problem Solving from Nature – PPSN XVII, pp. 63–77. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-14721-0_5
13. Kalkreuth, R., Rudolph, G., Droschinsky, A.: A new subgraph crossover for cartesian genetic programming. In: McDermott, J., Castelli, M., Sekanina, L., Haasdijk, E., García-Sánchez, P. (eds.) Genetic Programming, pp. 294–310. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-55696-3_19
14. Kaufmann, P., Kalkreuth, R.: An empirical study on the parametrization of cartesian genetic programming. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO 2017), pp. 231–232. Association for Computing Machinery, New York (2017). https://doi.org/10.1145/3067695.3075980
15. Kaufmann, P., Platzner, M.: Advanced techniques for the creation and propagation of modules in cartesian genetic programming. In: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation (GECCO 2008), pp. 1219–1226. Association for Computing Machinery, New York (2008). https://doi.org/10.1145/1389095.1389334
16. Kruschke, J.K.: Bayesian estimation supersedes the t test. J. Exp. Psychol. Gen. **142**(2), 573–603 (2013). https://doi.org/10.1037/a0029146
17. Langdon, W.B., Poli, R., McPhee, N.F., Koza, J.R.: Genetic Programming: An Introduction and Tutorial, with a Survey of Techniques and Applications, pp. 927–1028. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78293-3_22
18. Miller, J., Smith, S.: Redundancy and computational efficiency in cartesian genetic programming. IEEE Trans. Evol. Comput. **10**(2), 167–174 (2006). https://doi.org/10.1109/TEVC.2006.871253
19. Miller, J., Thomson, P., Fogarty, T.: Designing electronic circuits using evolutionary algorithms. arithmetic circuits: a case study. In: Genetic Algorithms and Evolution Strategies in Engineering and Computer Science (1999)
20. Miller, J.F.: Cartesian Genetic Programming. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-17310-3_2
21. Miller, J.F.: An empirical study of the efficiency of learning Boolean functions using a cartesian genetic programming approach. In: Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation (GECCO 1999), vol. 2, pp. 1135–1142. Morgan Kaufmann Publishers Inc., San Francisco (1999)
22. Miller, J.F.: Cartesian genetic programming: its status and future. Genet. Program Evolvable Mach. **21**(1), 129–168 (2020)

23. Payne, A.J., Stepney, S.: Representation and structural biases in CGP. In: 2009 IEEE Congress on Evolutionary Computation, pp. 1064–1071 (2009). https://doi.org/10.1109/CEC.2009.4983064
24. Pätzel, D.: cmpbayes. https://github.com/dpaetzel/cmpbayes
25. Slaný, K., Sekanina, L.: Fitness landscape analysis and image filter evolution using functional-level cgp. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) Genetic Programming, pp. 311–320. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71605-1_29
26. Spears, W.M., Anand, V.: A study of crossover operators in genetic programming. In: Ras, Z.W., Zemankova, M. (eds.) Methodologies for Intelligent Systems, pp. 409–418. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-54563-8_104
27. Stegherr., H., Heider., M., Hähner., J.: Assisting convergence behaviour characterisation with unsupervised clustering. In: Proceedings of the 15th International Joint Conference on Computational Intelligence - ECTA, pp. 108–118. INSTICC, SciTePress (2023). https://doi.org/10.5220/0012202100003595
28. Torabi, A., Sharifi, A., Teshnehlab, M.: Using Cartesian genetic programming approach with new crossover technique to design convolutional neural networks. Neural Process. Lett. **55**(5), 5451–5471 (2023). https://doi.org/10.1007/s11063-022-11093-0
29. Turner, A.J., Miller, J.F.: Neutral genetic drift: an investigation using cartesian genetic programming. Genet. Program. Evol. Mach. **16**(4), 531–558 (2015). https://doi.org/10.1007/s10710-015-9244-6
30. Vasicek, Z.: Bridging the gap between evolvable hardware and industry using Cartesian genetic programming. In: Stepney, S., Adamatzky, A. (eds.) Inspired by Nature. ECC, vol. 28, pp. 39–55. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-67997-6_2
31. White, D.R., et al.: Better GP benchmarks: community survey results and proposals. Genet. Program. Evol. Mach. **14**(1), 3–29 (2013)
32. White, D.R., Poulding, S.: A rigorous evaluation of crossover and mutation in genetic programming. In: Vanneschi, L., Gustafson, S., Moraglio, A., De Falco, I., Ebner, M. (eds.) EuroGP 2009. LNCS, vol. 5481, pp. 220–231. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01181-8_19
33. Wilson, D.G., Miller, J.F., Cussat-Blanc, S., Luga, H.: Positional cartesian genetic programming (2018)