A Comparative Study on Crossover in Cartesian Genetic Programming

Jakub Husa[1(✉)] and Roman Kalkreuth[2]

[1] Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic
ihusa@fit.vutbr.cz

[2] Department of Computer Science, TU Dortmund University, Dortmund, Germany
Roman.Kalkreuth@tu-dortmund.de

Abstract. Cartesian Genetic Programming is often used with mutation as the sole genetic operator. Compared to the fundamental knowledge about the effect and use of mutation in CGP, the use of crossover has been less investigated and studied. In this paper, we present a comparative study of previously proposed crossover techniques for Cartesian Genetic Programming. This work also includes the proposal of a new crossover technique which swaps block of the CGP phenotype between two selected parents. The experiments of our study open a new perspective on comparative studies on crossover in CGP and its challenges. Our results show that it is possible for a crossover operator to outperform the standard $(1 + \lambda)$ strategy on a limited number of tasks. The question of finding a universal crossover operator in CGP remains open.

Keywords: Cartesian Genetic Programming · Crossover
Comparative study

1 Introduction

Genetic Programming (GP) as popularized by Koza [1–3] uses syntax trees as program representation. Cartesian Genetic Programming (CGP) as introduced by Miller et al. [4] offers a novel graph-based representation which in addition to standard GP problem domains, makes it easy to be applied to many graph-based applications such as electronic circuits, image processing, and neural networks. CGP is mainly used with mutation as the only genetic operator. The reason for this is that previous work on crossover in CGP has provided mixed results and comparative results about the use of crossover are missing.

Tree-based GP was originally introduced with a sub-tree crossover technique which swaps randomly chosen sub-branches of the parent trees to produce new offsprings. Koza considered crossover as the dominant genetic operator as a result of his experiments [2,3]. However, later research with more comprehensive and detailed experiments found that the beneficial effects of crossover cannot be generalized in GP [5–7].

In contrast to fundamental knowledge about crossover in tree-based GP, the state of knowledge in CGP appears to be comparatively weak. Furthermore, the potential and understanding of crossover in CGP seem to be an open and remaining question. In this paper, we present the results of a first comparative study on crossover in CGP which includes the comparison of formerly proposed crossover techniques. Furthermore, we introduce a new method of crossover for CGP, called Block crossover, which is also investigated in our study.

Section 2 of this paper describes CGP briefly and surveys previous work on crossover in CGP. This section also surveys former attempts of comparative crossover studies in tree-based GP and reviews its contribution to the understanding of GP. In Sect. 3 we introduce our new form of crossover for CGP. Section 4 is devoted to the experimental results of our study and the description of our experiments. In Sect. 5 we discuss the results of our experiments. Finally, Sect. 6 gives a conclusion and outlines future work.

2 Related Work

2.1 Cartesian Genetic Programming

In contrast to tree-based GP, CGP represents a genetic program via genotype-phenotype mapping as an indexed, acyclic and directed graph. Originally the structure of the graphs was a rectangular grid of N_r rows and N_c columns, but later work also focused on a representation with just one row. The genes in the genotype are grouped, and each group refers to a node of the graph, except the last group which represents the outputs of the phenotype. Each node is represented by two types of genes which index the function number in the GP function set and the node inputs. These nodes are called *function nodes* and execute functions on the input values. The number of input genes depends on the maximum arity N_a of the function set. The last group in the genotype represents the indexes of the nodes which lead to the outputs.

A backward search is used to decode the corresponding phenotype. The backward search starts from the outputs and processes the linked nodes in the genotype. In this way, only active nodes are processed during the evaluation. The number of inputs N_i, outputs N_o and the length of the genotype is fixed. Every candidate program is represented with $N_\mathrm{r} * N_\mathrm{c} * (N_\mathrm{a} + 1) + N_\mathrm{o}$ integers. Even when the length of the genotype is fixed for every candidate program, the length of the corresponding phenotype in CGP is variable which can be considered as a significant advantage of the CGP representation.

CGP traditionally operates with a $(1+\lambda)$ evolutionary algorithm (EA) in which λ is often chosen with a size of four. The new population in each generation consists of the best individual of the previous population and the λ created offspring. The breeding procedure is mostly done by a point mutation which creates offsprings by changing a small number of randomly selected genes from the parent genotype to a random value within the permissible range. One of the most important techniques is a special rule for the selection of the new parent. In the case when two or more individuals can serve as the parent, an individual which has not served as the parent in the previous generation will be selected as a new parent. This strategy is important because it ensures the diversity of the population and has been found highly beneficial for the search performance of CGP.

2.2 Previous Work on Crossover in CGP

Some of the first experiments on crossover in CGP included the investigation of four variations of crossover which were tested on the simple regression problem $x^2 + 2x + 1$. Clegg et al. [8] reported that all tested variations of crossover techniques influenced the convergence of CGP negatively. In comparison to the mutation-only CGP algorithm, the addition of the crossover techniques hindered the performance of CGP. The crossover techniques were applied to the standard integer-based representation of CGP.

For instance, the genetic material was recombined by swapping parts of the genotypes of the parent individuals or randomly exchanging selected nodes. Clegg et al. [8] stressed that merely swapping the integers (in whatever manner) on a genotypic level in CGP disrupts the performance.

This was the motivation for a new form of crossover which has been introduced by Clegg et al. [8] and is based on a real-valued representation. This variation of CGP represents the graph as a fixed length list of real-valued numbers in the interval [0,1]. The genes are decoded to the integer-based representation with the help of normalization values (e.g. the number of functions or maximum input range). The recombination of two genotypes is performed with a standard Arithmetic crossover operation which uses a random weighting factor and can also be found in the field of real-valued Genetic Algorithms. The experiments of Clegg et al. showed that the new representation in combination with crossover improves the convergence behavior of CGP. However, for the convergence behavior in the later generations, Clegg et al. showed that the use of crossover in real-valued CGP leads to disruptive effects on one of the two tested problems. The improved convergence of the Arithmetic crossover was evaluated in the domain of symbolic regression and has been found useful in this problem domain [8].

Slaný et al. [9] analyzed the fitness landscapes of functional-level CGP on image operator design problems. Slaný et al. analyzed single and multi-point crossover operators. It was demonstrated that the mutation operator and the single-point crossover operator generate the smoothest landscapes for the tested problems.

For a multi-chromosome approach to CGP, Walker et al. [10] investigated a multi-chromosome crossover operator which joins the best chromosome parts from all individuals. This crossover technique was found useful for problems with multiple outputs and independent fitness assignment.

A beneficial effect of crossover in CGP was obtained by the use of an implicit context representation for CGP in which recombination is useful for the Even Parity-3 problem [11].

CGP has been extended for the automatic definition and reuse of functions by Walker et al. [12] and Kaufmann et al. [13]. Kaufmann et al. adopted the module creation mechanisms for a cone- and age-based CGP crossover [13]. Cone-based crossover showed good results for functions with repetitive inner patterns, while age-based crossover excels for randomized inner structures.

Recently, a new form of crossover has been introduced by Kalkreuth et al. [14]. The subgraph crossover recombines random parts of the CGP phenotype of two former selected individuals. This crossover technique has been found beneficial for the performance of CGP on symbolic regression, Boolean functions, and image operator design problems.

To the best of our knowledge, the most recent work on crossover in CGP has been done by Kalkreuth et al. However, while some crossover operators for standard CGP have been introduced and investigated, comprehensive comparative studies are still missing. This has been the motivation for our work.

3 The Block Crossover

The Block crossover is a new method of crossover for standard CGP. The method is mainly inspired by the cone-based crossover of Kaufmann et al. [13] for Embedded CGP, which integrates selected modules of a donor genotype into a receptor genotype. Since Kaufmann et al. have been successful with this crossover technique for specific boolean functions, our motivation for the proposal of the Block crossover is to adapt this mechanism for standard CGP. The Block crossover is also inspired by the subgraph crossover which has been introduced by Kalkreuth [14]. Since CGP suffers from a lack of a diverse and effective set of crossover techniques, the introduction and investigation of new crossover technique is significant.

The Block crossover technique focuses on the one-dimensional representation of CGP where the number of rows is limited to one. Given a previously selected genotypes of two individuals serving as parents, the Block crossover generates a list of all blocks of nodes that meet the following criteria:

- The block contains a desired number of nodes.
- All nodes in the block are directly linked through their inputs or outputs.
- All nodes in the block are part of the genotype's active path.

In our implementation, we have chosen to use blocks consisting of three nodes. To fulfill the other criteria, we have constructed the blocks by evaluating the genotype's active path, and selecting active nodes who's inputs were two distinct nodes and not primary inputs of the genotype. The time complexity of this simple method is linear, and it is performed along with the standard evaluation of the genotype's active path that precedes its evaluation.

The Block crossover then randomly selects one block from each list and swaps them between the genotypes. The position of the nodes transferred as part of the block may change inside the new genotype. However, their mutual links are preserved and the function performed by the block stays the same. Therefore, the created offsprings retain the same active path but performs a new operation. If either parent contains no swap-able blocks, no crossover operation is performed and the offsprings are simply cloned from their parents. The crossover operation is then followed by the standard point mutation.

Figure 1 illustrates the crossover procedure. First the active paths are determined, and the swap-able blocks are stored in the lists M_1 and M_2. Then, two blocks N_1 and N_2 are chosen from their respective lists. In order to produce the first offspring O_1, the first parent P_1 is cloned, and the function nodes inside the selected block N_1, are replaced by nodes from block N_2. Nodes $(2, 5, 6)$ have been moved to position $(2, 3, 4)$, but by maintaining them in the same order within the one-row genotype, we can ensure their mutual connection, and their logical function stay the same. The second offspring O_2 is produced in the same way but the roles of the parents P_1 and P_2 are reversed.

4 Experiments

4.1 Experimental Setup

We have performed experiments on problems from the symbolic regression and Boolean function domains. To evaluate the search performance, we measured the best fitness value found after a predefined number of fitness function evaluations (*best-fitness-of-run*). For all problems, the fitness was to be minimized. Our comparison has focused on four crossover operators. Standard One-point crossover, Subgraph crossover, Arithmetic crossover and our newly proposed Block crossover.

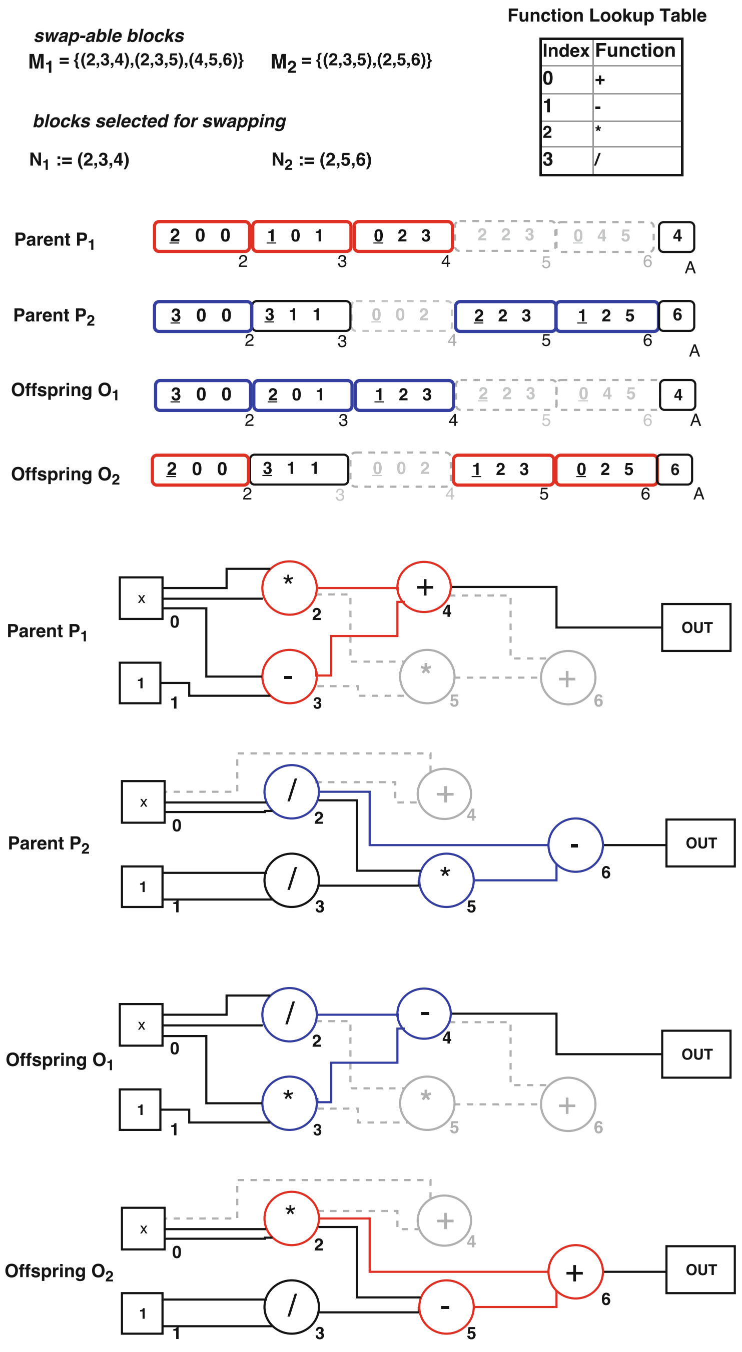
The evolution used a generational model. The initial population was randomly generated. Parent genomes for the next generation are picked using two separate tournaments, which allow for the same individual to be picked multiple times. The parents and a crossover operator are used ot create two offsprings, which are then mutated. This process is repeated until a sufficient number of offsprings has been created. Next generation consists of offsprings and a certain percentage of the best individuals (elites) from the previous generation.

In addition, two more evolutionary setups were added for comparison. The None crossover uses the same evolutionary scheme, but the offsprings it creates are identical clones of their parents, leaving mutation as the only active genetic operator. The $(1 + \lambda)$ setup forgoes the above described setup and implements the traditional CGP algorithm.

Our experiments have focused on examining the following hypothesis.

Hypothesis 1. *The* $(1 + \lambda)$ *CGP algorithm performs better than the crossover operators in all domains.*

In order to test this hypothesis, we first performed two rounds of meta-evolutionary experiments in order to determine which evolutionary parameters were critical, so that the crossover operators can all use their optimal setting, and be compared in a fair way. The two most important parameters were then subject to a parameter sweep, and for every crossover operator the best performing combination of parameters has been selected for comparison. To classify the significance of our results, we have used the Mann-Whitney U Test, to compare the standard $(1 + \lambda)$-CGP with all other crossover operators.



Index	Function
0	+
1	-
2	*
3	/

Fig. 1. The block crossover technique.

The implementation was done in Java, using the ECJ Evolutionary Computation Research System. All experiments were performed on a computing cluster with the following hardware configuration: 2 x Intel Xeon E5-2680v3 processor, 2.5 GHz, 12 cores; 128 GB RAM, 5.3 GB cache per core, DDR4@2133 MHz; InfiniBand FDR56 network connection.

4.2 Meta-evolution

For the meta-level, we used a basic canonical GA to tune five parameters we considered most important to the evolutionary process. Meta-evolution is very costly in terms of the computational effort necessary to find an optimal parameter setting. Furthermore, since GP benchmark problems can be very noisy in terms of finding the ideal solution, the evaluation of the evolved individuals is repeated multiple times, with fitness defined as the mean result.

During the first round of meta-evolution, all problems used the same setting, and the evolved parameters have been limited to discrete values, as seen in Table 1. During the second round, the granularity and range were modified to better fit each individual problem. Because the $(1 + \lambda)$ scheme does not use tournament selection nor elitism, the two parameters have been ignored during its meta-evolution.

Table 1. Configuration of the first round of the meta-evolutionary GA.

Property	Setting	Evolved parameter	Possible values
Maximum generations	50	Mutation rate	0.01 – 0.20
Population size	10	Elitism rate	0, 0.05 – 0.50
Mutation probability	0.5	Population size	2 – 1024*
Mutation type	Random walk	Genotype length	2 – 1024*
Tournament size	2	Tournament size	2 – 1024*
Number of trials	5		

* {2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, 128, 192, 256, 384, 512, 768, 1024}.

Results of the first round of meta-evolution have revealed that the tournament size parameter behaves wildly and does not converge to any specific value for any problem nor type of crossover. In some cases, it even significantly outgrew the population size. This caused the tournaments to include the entire population, resulting in a crossover of the best individual with itself, and wholly defeated the purpose of the crossover operator. To prevent this from happening, the tournament size has not been included in the second round of meta-evolution and its value has been fixed to four.

Table 2 shows the results of the second round of meta-evolution which were used to set up the ensuing parameter sweep. Because the computational effort required to perform a parameter sweep grows exponentially with the number of parameters, only the two most important parameters, mutation rate and population size, were included in the sweep.

Table 2. Results of the second round of the meta-evolutionary GA. The table shows the best-performing combination of the four tuned parameters.

Problem	Algorithm	Mutation rate	Elitism rate	Population size	Genotype length
Adder	$(1+\lambda)$	0.01	–	4	512
	None	0.01	0.08	4	384
	Block	0.01	0.10	4	1536
	Subgraph	0.01	0.06	6	768
	One-point	0.02	0.24	6	96
	Arithmetic	0.025	0.26	6	96
Multiplier	$(1+\lambda)$	0.05	–	3	24
	None	0.02	0.20	4	96
	Block	0.035	0.22	4	128
	Subgraph	0.04	0.04	4	64
	One-point	0.035	0.02	4	64
	Arithmetic	0.01	0.06	6	384
Bent	$(1+\lambda)$	0.14	–	24	128
	None	0.09	0.20	24	512
	Block	0.045	0.22	3	128
	Subgraph	0.04	0.20	12	256
	One-point	0.10	0.24	12	256
	Arithmetic	0.05	0.20	6	256
Resilient	$(1+\lambda)$	0.07	–	2	64
	None	0.07	0.20	32	2048
	Block	0.12	0.26	3	96
	Subgraph	0.09	0.26	3	96
	One-point	0.035	0.20	192	512
	Arithmetic	0.025	0.28	6	256
Koza-3	$(1+\lambda)$	0.08	–	24	64
	None	0.15	0.10	64	64
	Block	0.19	0.22	96	32
	Subgraph	0.07	0.20	14	16
	One-point	0.09	0.08	16	24
	Arithmetic	0.12	0.28	12	32
Nguyen-4	$(1+\lambda)$	0.07	–	24	192
	None	0.05	0.14	192	1024
	Block	0.11	0.08	6	96
	Subgraph	0.17	0.16	32	96
	One-point	0.18	0.16	6	128
	Arithmetic	0.05	0.10	16	128
Nguyen-7	$(1+\lambda)$	0.13	–	64	32
	None	0.11	0.18	12	96
	Block	0.10	0.10	6	48
	Subgraph	0.22	0.10	6	192
	One-point	0.09	0.28	64	256
	Arithmetic	0.16	0.12	4	48
Pagie-1	$(1+\lambda)$	0.05	–	2	384
	None	0.10	0.10	64	768
	Block	0.10	0.20	4	1536
	Subgraph	0.09	0.06	4	256
	One-point	0.05	0.08	8	1024
	Arithmetic	0.09	0.22	32	512

The ideal elitism rate was similar across all problems and types of crossover. For the sweep, it has been set to the overall average of 15%. Combined with the fixed tournament size of four, this means that during the sweep, there would be 52.2% chance none of the individuals in a tourney would be elites from the previous generation. The ideal genotype length was highly variable and largely depended on the problem, rather than the type of crossover used. For the sweep, the genotype length was set up individually for each problem.

4.3 Boolean Functions

We have chosen to evolve both single and multiple output Boolean functions. 2-bit digital adder and multiplier were used as our multiple output problems. Former work by White et al. [15] proposed these, as suitable alternatives to the overused parity problems. Their fitness was defined as a hamming distance between the resulting truth table, and the ideal solution. To increase the speed of the evaluation, we have used compressed truth tables.

For single output problems, we used 8-bit bent and 1-resilient Boolean functions. These functions find their use in cryptography, where they can provide an LFSR based key-stream generator of a stream cipher with resistance to linear and correlation attacks [16].

Bent Boolean functions possess the maximum possible degree of nonlinearity, defined as the Hamming distance between the truth table of a given function, and truth tables of all linear function and their negations. For an 8-bit function, maximum degree of nonlinearity is 120 [17]. We defined their fitness, as the difference between its actual degree of nonlinearity and the optimal value.

1-resilient functions are highly nonlinear functions that are balanced and have correlation immunity of the first degree. Balancedness means that the function's truth table contains the same number of ones and zeros. Correlation immunity, means that if the truth table was split in half based on the value of a specific input, the two halves of the truth table would each remain balanced. To the best of our knowledge, the maximum possible nonlinearity of an 8-bit 1-resilient

Table 3. Configuration of the Boolean function parameter sweep.

Property	Adder	Multiplier	Bent	Resileint
Input bits	5	4	8	8
Output bits	3	4	1	1
Genotype length	512	96	256	192
Mutation rate	0.002 – 0.02	0.005 – 0.05	0.01 – 0.1	0.01 – 0.1
Population size	2 – 48*	2 – 48*	2 – 48*	2 – 48*
Fitness evaluations	10000	5000	2000	5000
Tournament size	2	2	2	2
Percentage of elites	0.15	0.15	0.15	0.15

*\{2, 3, 4, 6, 8, 12, 16, 24, 32, 48\}.

Table 4. Results of the parameter sweep for Boolean functions.

Problem	Crossover type	Mutation rate	Pop. size	Mean fitness	**SD**	Q1	**Median**	Q3
Adder	$(1+\lambda)$	0.010	2	**4.26**	3.3923	1	4	6
	None	0.008	3	6.61[b]	3.4638	4	6	9
	Block	0.010	3	6.88[b]	3.2358	5	7	8
	Subgraph	0.010	3	6.60[b]	3.9029	4	6	9
	One-point	0.014	2	6.99[b]	3.5604	4.75	6.5	8.25
	Arithmetic	0.010	3	6.96[b]	3.0975	5	7	9
Multiplier	$(1+\lambda)$	0.035	2	**1.13**	1.0016	0	1	2
	None	0.020	4	2.09[b]	1.4777	1	2	3
	Block	0.035	3	2.14[b]	1.5441	1	2	3
	Subgraph	0.015	3	1.85[b]	1.4240	1	2	3
	One-point	0.020	4	2.03[b]	1.4997	1	2	3
	Arithmetic	0.025	2	2.23[b]	1.5166	1	2	3
Bent	$(1+\lambda)$	0.05	2	**2.92**	3.8604	0	0	8
	None	0.06	24	4.10	4.3705	0	4	8
	Block	0.04	8	3.89	4.0098	0	4	8
	Subgraph	0.05	32	4.28	4.1974	0	4	8
	One-point	0.05	16	4.04	3.9182	0	4	8
	Arithmetic	0.05	3	4.88	4.0931	0	8	8
Resilient	$(1+\lambda)$	0.07	2	16.89	19.6612	4	4	20
	None	0.07	4	**5.84**[b]	5.0667	4	4	4
	Block	0.08	6	6.64[b]	5.6916	4	4	4
	Subgraph	0.04	6	6.24[b]	5.4627	4	4	4
	One-point	0.09	4	6.12[b]	5.2863	4	4	4
	Arithmetic	0.04	3	8.48[a]	6.9464	4	4	14

[a] p-value is less than 0.05. [b] p-value is less than 0.01.

function is not known, but it can not be higher than 116 [18]. We defined the fitness, as the difference between the actual degree of nonlinearity and the optimal value, and if the evolved function was not resilient, its fitness was further penalized by 58, half the known limit.

Table 3 shows the setting used for the parameter sweep of Boolean functions. Each setting was run one hundred times, for every problem and type of crossover. All problems used the following function set {AND, OR, XOR, AND with one input inverted}. Because the best performing population size was usually very small, we have reduced the tournament size to two, to avoid repeating the issue from the first round of meta-evolution. For problems where the optimized setting was routinely able to find the ideal solution, we have also reduced the number of fitness function evaluations to get more telling results.

Table 4 shows the results of the parameter sweep. For each problem and crossover operator, we have selected combination of mutation rate and population size which provided the best mean fitness over the hundred runs. Operators that performed significantly different from $(1+\lambda)$ have their mean values marked. The table also shows standard deviation (SD) and three quantiles.

Figure 2 provides visual comparison using box plots. The Arithmetic crossover, originally intended for use in symbolic regression, performs the worst when used for Boolean function design. For adder and multiplier problems, the $(1+\lambda)$ strategy has significantly surpassed all other approaches. However, for the bent function, there was no statistically significant difference, and for the 1-resilient function, the $(1+\lambda)$ has performed significantly worse than the other options. Here, even with an optimal setting, some of the runs failed to produce a resilient function, resulting in significant deterioration of the average fitness.

4.4 Symbolic Regression

For symbolic regression, we have chosen four problems from the work of Clegg et al. [8] and McDermott et al. [19] for better GP benchmarks, and the Pagie-1 one problem which has been proposed by White et al. [15] as an alternative to the heavily overused Koza-1 ("quartic") problem. The analytic functions of the problems are shown in Table 5. The training data set U$[a,b,c]$ refers to c uniform random samples drawn from a to b inclusive and E$[a,b,c]$ refers to a grid of points evenly spaced with an interval of c, from a to b inclusive.

The fitness of the individuals was represented by a cost function value, defined as the sum of the absolute differences between the correct function values and

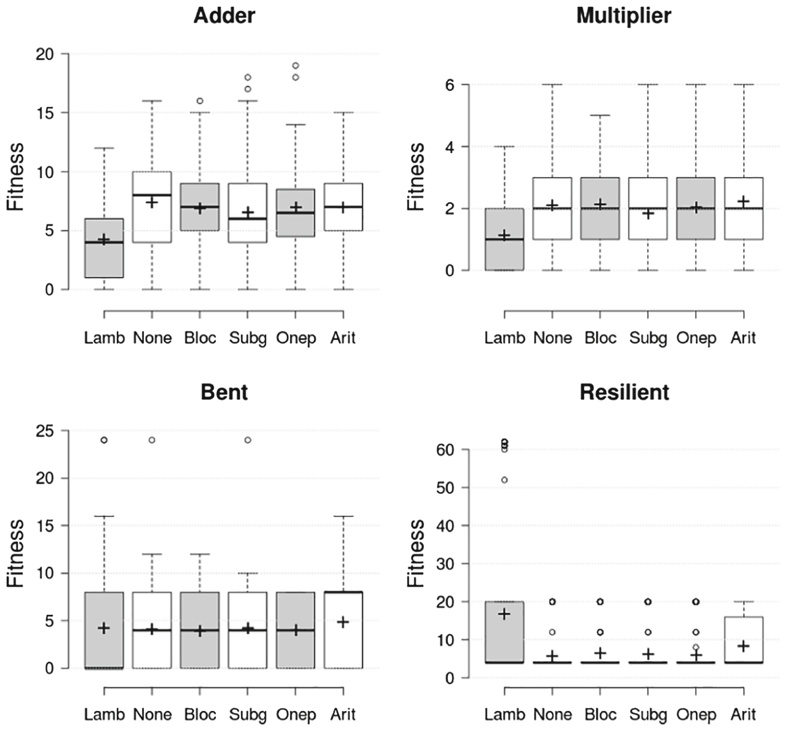


Fig. 2. Comparison of crossover operators for Boolean functions.

Table 5. Symbolic regression problems used in the experiment.

Problem	Objective function	Vars	Training set
Koza-3	$x^6 - 2x^4 + x^2$	1	U[−1, 1, 20]
Nguyen-4	$x^6 + x^5 + x^4 + x^3 + x^2 + x$	1	U[−1, 1, 20]
Nguyen-7	$ln(x+1) + ln(x^2+1)$	1	U[0, 2, 20]
Pagie-1	$1/(1+x^{-4}) + 1/(1+y^{-4})$	2	E[−5, 5, 0.4]

Table 6. Configuration of the symbolic regression parameter sweep.

Property	Koza-3	Nguyen-4	Nguyen-7	Pagie-1
Genotype length	48	128	128	512
Mutation rate	0.02 – 0.2	0.02 – 0.2	0.02 – 0.2	0.02 – 0.2
Population size	4 – 96*	4 – 96*	4 – 96*	4 – 96*
Fitness evaluations	10000	10000	10000	10000
Tournament size	4	4	4	4
Percentage of elites	0.15	0.15	0.15	0.15

* {4, 6, 8, 12, 16, 24, 32, 48, 64, 96}.

Table 7. Results of the parameter sweep for symbolic regression.

Problem	Crossover type	Mutation rate	Pop. size	Mean fitness	SD	Q1	Median	Q3
Koza-3	$(1+\lambda)$	0.16	24	0.0664	0.0774	0.0119	0.0504	0.0839
	None	0.12	16	0.0642	0.0815	0.0092	0.0376	0.0849
	Block	0.06	12	0.0636	0.0755	0.0200	0.0455	0.0784
	Subgraph	0.16	64	0.0692	0.0819	0.0168	0.0483	0.0852
	One-point	0.14	96	0.0617	0.0640	0.0154	0.0333	0.0878
	Arithmetic	0.12	12	**0.0435**	0.0405	0.0146	0.0311	0.0764
Nguyen-4	$(1+\lambda)$	0.12	6	**0.3120**	0.2658	0.1574	0.2478	0.3745
	None	0.10	8	0.3307	0.2326	0.1672	0.2865	0.4130
	Block	0.08	16	0.3485	0.2800	0.1800	0.2850	0.4125
	Subgraph	0.10	6	0.3709[a]	0.2692	0.1940	0.3393	0.4652
	One-point	0.10	12	0.3282	0.2351	0.1579	0.2864	0.4219
	Arithmetic	0.08	8	0.3231	0.2305	0.1560	0.2558	0.4318
Nguyen-7	$(1+\lambda)$	0.18	64	**0.6722**	0.4215	0.4364	0.5935	0.7682
	None	0.10	6	0.6871	0.3736	0.4464	0.6055	0.8073
	Block	0.12	24	0.7601[a]	0.3352	0.5522	0.7101	0.9163
	Subgraph	0.12	32	0.7724[a]	0.4461	0.5090	0.7155	0.9613
	One-point	0.16	16	0.7136	0.3741	0.4405	0.6984	0.8439
	Arithmetic	0.14	6	0.8132[a]	0.4978	0.5502	0.7027	0.8288
Pagie-1	$(1+\lambda)$	0.08	8	130.8812	48.2214	93.7397	122.7972	160.8945
	None	0.06	96	134.5053	46.6960	96.3143	140.6268	170.5598
	Block	0.04	96	126.1124	45.7809	87.4737	122.3703	161.1563
	Subgraph	0.08	64	150.4739[b]	46.9169	115.0119	161.9589	181.6550
	One-point	0.06	8	130.6106	48.9600	96.4414	122.4861	169.5678
	Arithmetic	0.04	8	**120.1536**	45.7169	84.6019	114.4325	152.5632

[a] p-value is less than 0.05. [b] p-value is less than 0.01.

the values of an evaluated individual. The configuration of the experiment is shown in Table 6. All problems used the following set of mathematical functions $\{+, -, *, /, \sin, \cos, \ln(|n|), e^n\}$.

Table 7 shows the parameter sweep results. Same as before, the primary selected criterion was the best average fitness over one hundred runs. Crossover operators that performed significantly different from $(1+\lambda)$ have their mean values marked. As can be seen in Fig. 3, the arithmetic crossover performs very well, when used for symbolic regression, as originally designed.

5 Discussion

In our meta-evolutionary experiments, we dealt with significant problems in order to make a fair comparison. We were able to determine optimal parameter settings for the $(1+\lambda)$-CGP as the tuning consists of only three parameters: population size, mutation rate, and genotypic length. However, determining optimal parameter settings for the canonical crossover algorithms is more complex. There are three additional parameters to contend with: crossover rate, elitism rate, and tournament size, which makes obtaining an optimal parameter setting for the respective problems significantly more difficult.

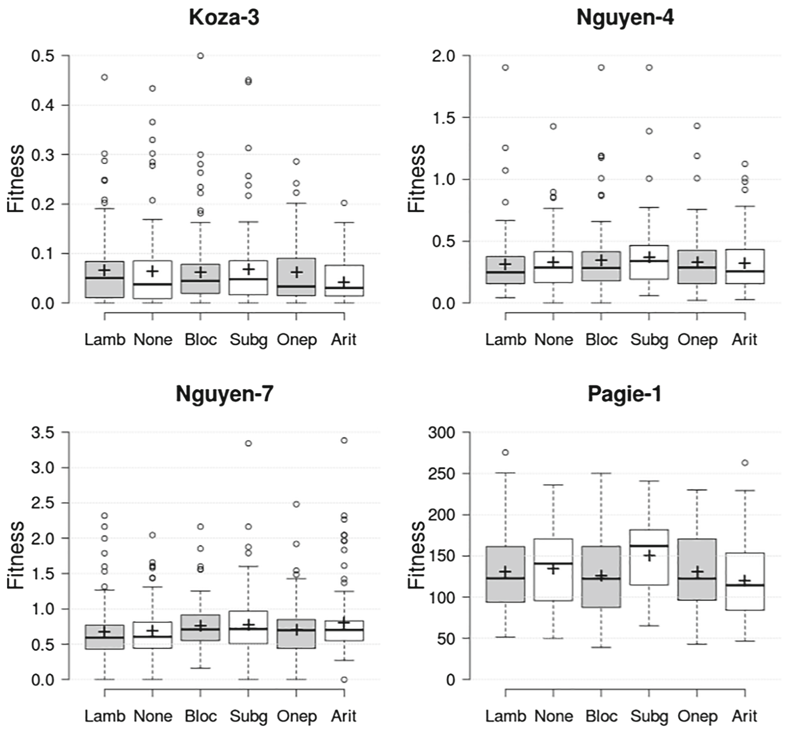


Fig. 3. Comparison of crossover operators for symbolic regression.

Furthermore, former studies on the traditional $(1 + \lambda)$-CGP algorithm have shown that large genotypes are very effective for the performance of CGP for certain problems. Consequently, we have to deal with a big parameter space in CGP in order to determine the optimal parameters and to make a fair comparison.

For this paper, we only used the meta-evolution framework of the Java Evolutionary Computation Toolkit (ECJ)[1]. However, we think that including other *state-of-the-art* methods for parameter tuning of evolutionary algorithms, like Iterated Race for Automatic Algorithm Configuration (IRace)[2] or Sequential-parameter-optimization (SPOT)[3], can provide more insight into well-performing algorithm settings in CGP, and help to provide fair and profound comparisons.

Another point which should be discussed is the observation that each type of crossover works best with different settings. Our findings indicate that there exists no general parameterization pattern for CGP when the crossover is in use. We think it should be investigated if there are similar behaviors like exploration abilities which could be obtained by fitness and search space analyses.

The results of our study show that the parameter settings vary for different problems in the respective problem domain, and indicate that there is no general pattern to parametrize the $(1+\lambda)$-CGP in a well-performing way. These findings also open up a new question, which conditions or types of problems have the need for bigger or smaller population sizes. A preliminary assumption could be that the fitness landscape of certain problems requires more exploration abilities in order to overcome local optima.

Our results indicate that bigger populations perform well in the symbolic regression domain. This finding is consistent with a recent study on mutation-only CGP by Kaufmann et al. [20] which also indicates that bigger populations perform best in the symbolic regression domain.

Since our experiments validate Kaufmann et al. results, this behavior should be investigated through more detailed experiments. Furthermore, we think that these findings offer a good opportunity to get more understanding of how CGP works in detail and can significantly contribute to the overall knowledge of fitness landscape analysis in CGP.

Specifically, Kaufmann et al. show that a mutational $(\mu + \lambda)$ evolutionary algorithm with big population size can be very effective. Therefore, we think it should be investigated whether the Block crossover can be used with a $(\mu + \lambda)$ evolutionary algorithm, as a part of our attempts to proceed towards more precise comparative studies in CGP.

5.1 Analysis of Hypothesis

The results of our comparative study show that the traditional $(1+\lambda)$-CGP algorithm can not be stated as the universally predominant algorithm for CGP. While it is often a good choice, the outcome of our study gives a significant evidence

[1] https://cs.gmu.edu/~eclab/projects/ecj/.
[2] http://iridia.ulb.ac.be/irace/.
[3] http://www.spotseven.de/category/sequential-parameter-optimization/.

that the $(1 + \lambda)$-CGP can not be considered as the most efficient CGP algorithm in the boolean function domain. The experiments on 1-resilient Boolean function proves that the $(1 + \lambda)$-CGP may indeed be significantly inferior to the other CGP algorithms.

6 Conclusion and Future Work

The first comprehensive comparative study on crossover in CGP has been proposed. We also proposed a new Block crossover technique, inspired by embedded CGP, for use in standard CGP. We have performed a comparative study using our new crossover technique, two evolutionary methods that only use mutation, and three other crossover operators that have been suggested in the literature. Simple One-point Crossover, Arithmetic crossover, used in the field of real-valued Genetic Algorithms, and Subgraph Crossover that recombines parts of the parent chromosome phenotypes.

We have formulated a hypothesis that the traditional $(1 + \lambda)$-CGP algorithm would not perform significantly worse than the crossover operators. We performed a comparison on eight selected tasks from the areas of Boolean function design and symbolic regression. We have used meta-evolution to determine the most important evolutionary parameters and find common values for the parameters of lower importance.

Next, we have performed a series of parameter sweeps, to determine the settings most suitable for every type of crossover and every task, and performed a comparison. Finally, we have performed a non-parametric statistical test to prove our hypothesis false, and shown that the $(1 + \lambda)$-CGP is significantly outperformed by all other approaches, when designing 1-resilient Boolean functions.

Our results show, that it is possible for crossover operators to outperform the standard $(1 + \lambda)$ strategy. However, if both methods have their parameters fine-tuned, the $(1 + \lambda)$ strategy often remains as the overall best strategy. The question of finding a universal crossover operator is CGP therefore remains open.

Our study opens a new perspective on comparative studies on the use of crossover in CGP and its challenges. The experiments with meta-evolution in CGP have shown that it is difficult to obtain well-performing parameter settings for crossover algorithms in CGP.

These results are the first step toward a fair comparison and a more clear understanding of the function of crossover in CGP. Our future work will focus on exploring ways to make comparisons between crossover techniques and algorithms in CGP more fair, including the investigation of suitable parameter optimization techniques for CGP, widening the spectrum of problem domains on which comparison is made, and using crossover operators from other areas. We will especially focus on investigating the possibility of combining the Block crossover with the $(\mu + \lambda)$ evolutionary algorithm, and on exploring the domain of cryptographically significant boolean functions, where the $(1 + \lambda)$ algorithm faces great difficulty.

Acknowledgments. This work was supported by the Czech science foundation project 16-17538S.

References

1. Koza, J.: Genetic programming: a paradigm for genetically breeding populations of computer programs to solve problems. Technical Report STAN-CS-90-1314, Department of Computer Science. Stanford University, June 1990
2. Koza, J.R.: Genetic programming: on the programming of computers by means of natural selection. MIT Press, Cambridge (1992)
3. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge (1994)
4. Miller, J.F., Thomson, P.: Cartesian genetic programming. In: Poli, R., Banzhaf, W., Langdon, W.B., Miller, J., Nordin, P., Fogarty, T.C. (eds.) EuroGP 2000. LNCS, vol. 1802, pp. 121–132. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-540-46239-2_9
5. Luke, S., Spector, L.: A Comparison of Crossover and Mutation in Genetic Programming. In: Proceedings of the Second Annual Conference on Genetic Programming 1997, pp. 240–248. Morgan Kaufmann, Stanford University, CA, USA, 13–16 July (1997)
6. Luke, S., Spector, L.: A revised comparison of crossover and mutation in genetic programming. In: Proceedings of the Third Annual Conference on Genetic Programming 1998, pp. 208–213. Morgan Kaufmann, University of Wisconsin, Madison, Wisconsin, USA, 22–25 July (1998)
7. 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
8. Clegg, J., Walker, J.A., Miller, J.F.: A new crossover technique for cartesian genetic programming. In: GECCO 2007: Proceedings of the 9th annual Conference on Genetic and Evolutionary Computation, vol. 2, pp. 1580–1587. ACM Press, London, 7–11 July (2007)
9. 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.) EuroGP 2007. LNCS, vol. 4445, pp. 311–320. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71605-1_29
10. Walker, J.A., Miller, J.F., Cavill, R.: A multi-chromosome approach to standard and embedded cartesian genetic programming. In: GECCO 2006: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, vol. 1, pp. 903–910. ACM Press, Seattle, 8–12 July (2006)
11. 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.) EuroGP 2006. LNCS, vol. 3905, pp. 351–360. Springer, Heidelberg (2006). https://doi.org/10.1007/11729976_32
12. Walker, J.A., Miller, J.F.: Evolution and acquisition of modules in cartesian genetic programming. In: Keijzer, M., O'Reilly, U.-M., Lucas, S., Costa, E., Soule, T. (eds.) EuroGP 2004. LNCS, vol. 3003, pp. 187–197. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24650-3_17

13. Kaufmann, P., Platzner, M.: Advanced techniques for the creation and propagation of modules in cartesian genetic programming. In: GECCO 2008: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation, pp. 1219–1226. ACM, Atlanta, 12–16 July (2008)
14. 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.) EuroGP 2017. LNCS, vol. 10196, pp. 294–310. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-55696-3_19
15. White, D.R., McDermott, J., Castelli, M., Manzoni, L., Goldman, B.W., Kronberger, G., Jaskowski, W., O'Reilly, U.M., Luke, S.: Better GP benchmarks: community survey results and proposals. Genet. Program Evolvable Mach. **14**(1), 3–29 (2013)
16. Carlet, C.: Boolean functions for cryptography and error correcting codes. Boolean Models Methods Math. Comput. Sci. Eng. **2**, 257–397 (2010)
17. Picek, S., Jakobovic, D., Miller, J.F., Batina, L., Cupic, M.: Cryptographic boolean functions: One output, many design criteria. Appl. Soft Comput. **40**, 635–653 (2016)
18. Sarkar, P., Maitra, S.: Nonlinearity bounds and constructions of resilient boolean functions. In: Bellare, M. (ed.) CRYPTO 2000. LNCS, vol. 1880, pp. 515–532. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44598-6_32
19. McDermott, J., White, D.R., Luke, S., Manzoni, L., Castelli, M., Vanneschi, L., Jaśkowski, W., Krawiec, K., Harper, R., Jong, K.D., O'Reilly, U.M.: Genetic programming needs better benchmarks. In: Proceedings of the 14th International Conference on Genetic and Evolutionary Computation Conference, GECCO 2008, pp. 791–798. ACM, Philadelphia (2012)
20. Kaufmann, P., Kalkreuth, R.: Parametrizing cartesian genetic programming: an empirical study. In: Kern-Isberner, G., Fürnkranz, J., Thimm, M. (eds.) KI 2017. LNCS (LNAI), vol. 10505, pp. 316–322. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67190-1_26