

# Towards Phenotypic Duplication and Inversion in Cartesian Genetic Programming

Roman Kalkreuth<sup>1</sup>

Natural Computing Research Group, Leiden Institute of Advanced Computer Science, Leiden University,

**Keywords:** Cartesian Genetic Programming, Mutation, Duplication, Inversion, Boolean Function Learning.

**Abstract:** The search performance of Cartesian Genetic Programming (CGP) relies to a large extent on the sole use of genotypic point mutation in combination with extremely large redundant genotypes. Over the last years, steps have been taken to extend CGP's variation mechanisms by the introduction of advanced methods for recombination and mutation. One branch of these contributions addresses phenotypic variation in CGP. Besides demonstrating the effectiveness of various phenotypic search operators, corresponding analytical experiments backed evidence that phenotypic variation is another approach for achieving effective evolutionary-driven search in CGP. However, recent comparative studies have demonstrated the limitations of phenotypic recombination in Boolean function learning and highlighted the effectiveness of the mutation-only approach. Especially the use of the  $1 + \lambda$  selection strategy with neutral genetic drift has been found superior to recombination-based approaches in this problem domain. Therefore, in this work, we further explore phenotypic mutation in CGP by the introduction and evaluation of two phenotypic mutation operators that are inspired by chromosomal rearrangement. Our initial findings show that our proposed methods can significantly improve the search performance of CGP on various single- and multiple-output Boolean function benchmarks by reducing the number of fitness evaluations needed to find the ideal solution.

## 1 INTRODUCTION


Genetic programming (GP) can be considered as a search heuristic for computer program synthesis that is inspired by neo-Darwinian evolution. First work on GP has been done by Forsyth (Forsyth, 1981), Cramer (Cramer, 1985) and Hicklin (Hicklin, 1986). Later work by Koza (Koza, 1990; Koza, 1992; Koza, 1994) significantly popularized the field of GP. GP as proposed by Koza is traditionally used with trees as program representation. In contrast, Cartesian Genetic Programming is a well established graph-based GP variants. First work towards CGP was done by Miller, Thompson, Kalganova, and Fogarty (Miller et al., 1997; Kalganova, 1997; Miller, 1999) by the introduction of a graph encoding model based on a two-dimensional array of functional nodes. CGP can be seen as an extension to the traditional tree-based GP representation model since its representation allows many graph-based applications such as Boolean circuit design, image processing (Sekanina, 2002), and neural networks (Turner and Miller, 2013).

Even though standard CGP was introduced over two decades ago, it is still predominantly used only with a probabilistic point mutation operator for genetic variation.

The predominant use of the mutation-only CGP approach originates from failures of various genotypic crossover operators to improve its search performance. This motivated the introduction of two phenotypic approaches to recombination called sub-graph crossover (Kalkreuth et al., 2017) and block crossover (Husa and Kalkreuth, 2018). Although initial experiments with phenotypic recombination led to promising initial results, recent comparative studies (Kaufmann and Kalkreuth, 2017; Husa and Kalkreuth, 2018; Kalkreuth, 2020; Kalkreuth, 2021b) gave more evidence about the effectiveness of mutation-only CGP for Boolean function learning. However, the introduction of phenotypic recombination operators inspired later work on phenotypic mutation in CGP and led to the introduction of two operators called *insertion* and *deletion* (Kalkreuth, 2019; Kalkreuth, 2021a).

Considering recent findings on the limitations of re-

---

<sup>1</sup> <https://orcid.org/0000-0003-1449-5131>

combination in CGP and the fact that advanced variation is still underdeveloped when compared to the number of contributions in tree-based GP, this work aims to further exploration of phenotypic mutation. We introduce two new operators called *duplication* and *inversion* and evaluate these methods on a diverse set of Boolean function problems. Boolean function learning can be seen as one of the most popular problem domains of CGP since evolving solutions for this kind of problem has been a major motivation for its introduction (Miller, 1999).

The document is structured as follows: Section 2 of this work describes CGP. Related work on phenotypic variation in CGP is reviewed in Section 3. In Section 4, we introduce our new methods. Section 5 is devoted to the formulation of the research question, description of our experiments, and the presentation of our results. Based on our experimental findings, the formulated research question is analyzed in Section 6. Points of criticism are discussed in Section 7. Finally, Section 8 gives a conclusion and outlines our future work.

## 2 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. In this way, CGP can be seen as an extension of the traditional tree-based GP approach. The CGP representation model is based on a rectangular grid or row of nodes. A definition of a cartesian genetic program (CP) is given in Definition 2.1. Each CP is encoded in the genotype of an individual and is decoded to its corresponding phenotype. Originally, the structure of the graph was represented by a rectangular grid of  $n_r$  rows and  $n_c$  columns, but later work focused on a representation with one row. The CGP decoding procedure processes groups of genes, and each group refers to a node of the graph, except the last one, which represents the outputs of the phenotype. Each node is represented by two types of genes that 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. A definition of a function node is given in Definition 2.2. The number of input genes depends on the maximum arity  $n_a$  of the function set.

**Definition 2.1** (Cartesian Genetic Program). A cartesian genetic program is an element of the Cartesian product  $\mathcal{N}_i \times \mathcal{N}_f \times \mathcal{N}_o \times \mathcal{F}$  :

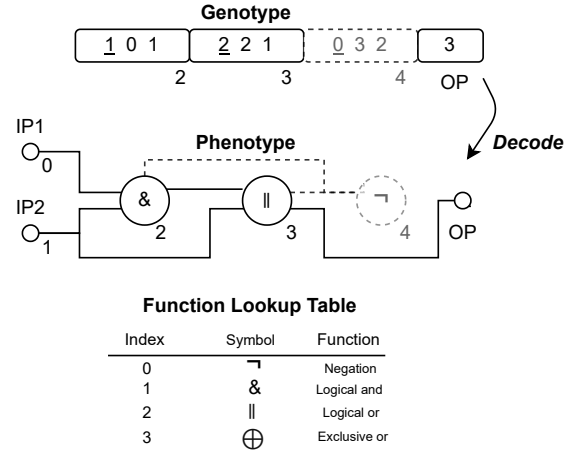


Figure 1: Example of the decoding procedure of a CGP genotype to its corresponding phenotype. The identifiers IP1 and IP2 stand for the two input nodes with node index 0 and 1. The identifier OP stands for the output node of the graph.

- $\mathcal{N}_i$  is a finite non-empty set of input nodes
- $\mathcal{N}_f$  is a finite set of function nodes
- $\mathcal{N}_o$  is a finite non-empty set of output nodes
- $\mathcal{F}$  is a finite non-empty set of functions

**Definition 2.2** (Function Node). Given the number of inputs  $n_i$ , the number of function nodes  $n_f$ , and the maximum arity  $n_a$ , a function node  $\theta_f$  is defined as a tuple  $\theta_f = (x_f, g_f, g_c^0, \dots, g_c^{n_a-1})$  of dimension  $n_a + 2$ :

- $x_f \in \{n_i, \dots, n_i + n_f - 1\}$  is the number of the function node
- $g_f \in \{x \in \mathbb{N}^0 \mid 0 \leq x \leq |\mathcal{F}| - 1\}$  is the function gene
- $g_c^0, \dots, g_c^{n_a-1} \in \{x \in \mathbb{N}^0 \mid 0 \leq x \leq x_f - 1\}$  are the connection genes

A backward search is conducted to decode the corresponding phenotype. The decoding procedure is done for all output genes. An example of the backward search of the most popular one-row integer representation is illustrated in Figure 1. The backward search starts from the program output and processes all nodes which are linked in the genotype. In this way, only active nodes are processed during evaluation. The genotype in Figure 1 is grouped by its function nodes. The first (underlined) gene of each group refers to the function number in the corresponding function set. The non-underlined genes represent the input connections of the node. Inactive function nodes are shown in gray color and with dashed lines.

The number of inputs  $n_i$ , outputs  $n_o$ , and the length of the genotype is fixed. Every candidate program is represented with  $n_r * n_c * (n_a + 1) + n_o$  integers. Even if the length of the genotype is fixed for each candidate program, the length of the corresponding phenotype in CGP is variable, which can be considered as an advantage of the CGP representation. CGP is traditionally used with a  $(1 + \lambda)$  evolutionary algorithm (EA). The  $(1 + \lambda)$ -EA is often used with a selection strategy called *neutrality*, which is based on the idea that genetic drift yields to diverse individuals having equal fitness. The genetic drift is implemented into the selection mechanism in a way that individuals which have the same fitness as the normally selected parent are determined, and one of these same-fitness individuals is returned uniformly at random. The new population in each generation consists of the best individual of the previous population and the  $\lambda$  created offspring. The breeding procedure is mostly done by a point mutation that swaps genes in the genotype of an individual in the valid range by chance. Another point mutation is the flip of the functional gene, which changes the functional behavior of the corresponding function node.

### 3 PHENOTYPIC VARIATION IN CGP

Phenotypic variation in CGP can be described as a type of chromosomal alteration whereby only the phenotype of an individual is varied by recombination or mutation. It originates from an investigation of the length bias and the search limitation of CGP (Goldman and Punch, 2013; Goldman and Punch, 2015). In their work, Goldman and Punch presented a modified version of the point mutation operator which exactly mutates one active gene. This so-called single active-gene mutation strategy (SAGMS) has been found beneficial for the search performance of CGP. The SAGMS can be seen as a form of phenotypic variation since it mutates only active genes.

Phenotypic mutation inspired the introduction of two phenotypic recombination operators called subgraph crossover (Kalkreuth et al., 2017) and block crossover (Husa and Kalkreuth, 2018). The subgraph crossover exchanges and links subgraphs of active function nodes between two selected parents and the block crossover exchanges blocks of active function genes. In recent comparative studies, its use has been found beneficial for several symbolic regression benchmarks since it led to a significant decrease in the number of fitness evaluations needed to find the ideal solution (Kalkreuth, 2020; Kalkreuth, 2021b).

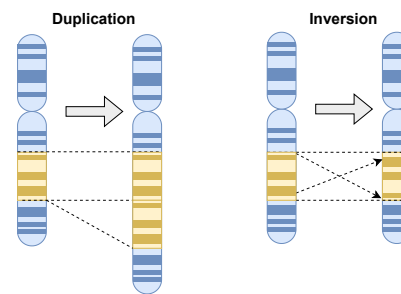


Figure 2: Exemplification of chromosomal duplication and inversion in nature which served as inspiration for this work. Duplication results in the formation of extra copies of a region within the chromosome. Inversion rearranges a segment of a chromosome by reversing it end-to-end.

However, the gain of the search performance was considerably lower for the tested Boolean function problems. Moreover, the results of the experiments clearly showed that the subgraph crossover failed to improve the search performance on some of the tested Boolean benchmarks when compared to the results of the traditional  $1 + \lambda$  selection strategy. Similar findings were reported for the block crossover (Kalkreuth, 2021b). Kalkreuth (Kalkreuth, 2019) adapted two phenotypic mutation operators called *insertion* and *deletion* that are inspired by frameshift mutation. *Insertion* selects and activates one inactive function node of a CP, where the inactive node is selected by chance. In contrast, *deletion* deactivates the first active function node of a CP. Experiments with Boolean and symbolic regression benchmarks demonstrated that the proposed mutations can contribute to the search performance of CGP on the tested problems. Češka et al. (Češka et al., 2021) proposed a combination of the classical single active gene mutation and a node deactivation operation for circuit design called *SagTree*. The deactivation operation of *SagTree* aims at eliminating a part of the circuit and forming a tree from an active gate.

Phenotypic mutation also led to the proposal of semantic mutation in CGP. Manfrini et al. (Manfrini et al., 2016) extended SAGMS to the so-called *Biased Single-Active Mutation*, that records the frequencies of beneficial function gene mutations during the evolutionary run which are then used to calculate probabilities for function transitions. Hodan et al. (Hodan et al., 2020) proposed a semantically oriented form of mutation that operates similar to the SAGMS but additionally uses semantics to determine the best value for each gene mutation.

## 4 THE PROPOSED METHODS

The proposed methods for phenotypic mutation in CGP are inspired by two major single-chromosome mutations which occur in nature and are studied in the field of molecular evolution. Duplication is a form of mutation that causes the generation of one or more copies of a gene or section of a chromosome. Another terminology that is common for this type of mutation is *gene amplification*. Inversion means that the order within a section of the chromosome is reversed. Figure 2 exemplifies chromosomal inversion and duplication in nature on a part of a chromosome. Both duplication and inversion count to the group of chromosomal rearrangement mutations, and inversion was originally adapted in the field of genetic algorithms (GA) by Holland (Holland et al., 1975). For instance, inversion in combination with the path representation for GA has been found useful for its application to the traveling salesman problem (Larrañaga et al., 1999). In CGP, gene duplication has been originally adapted for a variant of the representation model by Ebner (Ebner, 2012) to evolve sub-detectors for object detection tasks. We adapt the two mutations for CGP with phenotypic function variation which is also utilized by the block crossover. This means that function genes of active function nodes are selected and the values are rearranged by means of duplication and inversion. To regulate the strength of both mutations for the evolutionary process, the methods are utilized with a respective duplication and inversion probability. An implementation of both methods for the CGP extension package of the Java Evolutionary Computation Research System (ECJ) (Scott and Luke, 2019) is provided in the ECJ GitHub repository<sup>1</sup>.

### 4.1 Duplication

We adapt chromosomal duplication for CGP by selecting an active function gene and replacing the values of a sequence of successive active function genes with the value of the selected gene. This type of adaption does not cause any structural enlargement as it happens in nature. The reason for this is that such an operation is quite complex when the CGP representation model is used. According to Kalkreuth (Kalkreuth, 2021c), even the activation of a single inactive node requires several rearrangements of connection genes to ensure that other inactive nodes are not affected by the operation. With our adaption by replacement, we avoid this computational effort but at the same time, increase the frequency of the selected function within the phenotype. In this

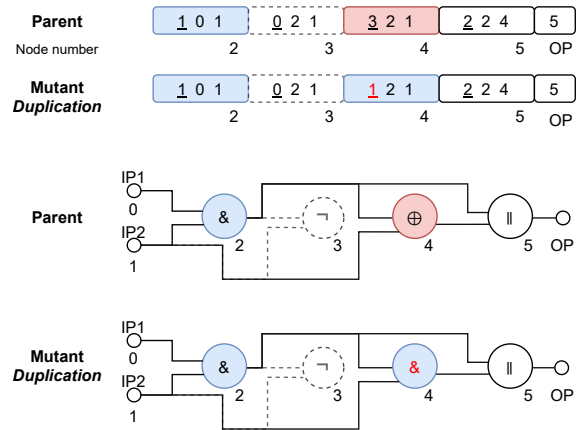


Figure 3: Exemplification of duplication in CGP: The value of the function gene of node 2 is duplicated to node 4 by the replacement of the function gene. In this example, the duplication depth was set to one. The duplicated function gene value is highlighted in red color.

way, we phenotypically amplify the occurrence of the selected function. The procedure is discarded if the CP has less than two active function nodes since at least two active function nodes are necessary. Before the duplication of the function gene is performed, a duplication depth is determined in accordance with the predefined maximum duplication depth and the number of active function nodes. When the number of active nodes is less than the predefined maximum depth, the depth is adjusted to the number of active function nodes. The maximum is then used to choose the duplication depth by chance. Afterward, the index of an active function node, whose position allows the duplication of its function gene of the determined depth, is selected. The function gene is finally duplicated and the altered genotype is returned.

A definition of duplication is given in Definition 4.1. The algorithmic implementation of the determination of the duplication depth and starting node is described in Algorithm 1 and 2. The algorithmic implementation of the duplication procedure is described in Algorithm 3. An exemplification is illustrated in Figure 3.

**Definition 4.1** (Duplication). Let  $d$  be the duplication depth and let  $\mathcal{D} = (g_f^0, \dots, g_f^d)$  be a tuple of active function gene values which have been selected for the duplication procedure. Let  $g_f^0 \in \mathcal{D}$  be the first active function gene value and let  $\tilde{\mathcal{D}} = (g_{f_0}^0, g_{f_1}^0, \dots, g_{f_d}^0)$  be a tuple with duplicated function gene values which is obtained after the duplication procedure. Duplication is a variation operator for CGP that adapts chromosomal duplication by replacing the active function genes  $\mathcal{D}$  by  $\tilde{\mathcal{D}}$ .

<sup>1</sup><https://github.com/GMUEClab/ecj>

Algorithm 1: Determination of the stochastic depth.

---

**Arguments**  
*D*: Maximum depth  
 $|N_a|$ : Number of active function nodes  
**Return**  
depth: Determined stochastic depth

```

1: function StochasticDepth(D,  $|N_a|$ )
2:   ▷ If the number of active nodes is less than the maximum depth
3:   if  $|N_a| \leq D$  then
4:     ▷ Adjust max to the maximum possible depth
5:      $\max \leftarrow |N_a| - 1$ 
6:   else
7:     ▷ Set max to the predefined maximum depth
8:      $\max \leftarrow D$ 
9:   end if
10:  ▷ Determine the depth by chance in the interval [1, max]
11:  depth  $\leftarrow$  RandomInteger(1, max)
12:  return depth
13: end function

```

---

Algorithm 2: Determination of the start index.

**Arguments**  
depth: Previously determined stochastic depth  
 $|N_a|$ : Number of active function nodes  
**Return**  
start: start index in the list of active function nodes

```

1: function StartIndex(depth,  $|N_a|$ )
2:   ▷ Set start initially to zero
3:   start  $\leftarrow$  0
4:   ▷ Calculate the max possible start index
5:    $\max \leftarrow |N_a| - \text{depth} - 1$ 
6:   ▷ If the maximum is greater zero
7:   if  $\max > 0$  then
8:     ▷ Replace start index with a random one in the interval [0, max]
9:     start  $\leftarrow$  RandomInteger(0, max)
10:  end if
11:  return start
12: end function

```

---

## 4.2 Inversion

Inversion permutes the function gene values of a set of successive active function nodes. The permutation is performed by reversing the order of the function gene values. The procedure is also discarded if the CP has less than two active function nodes. Before the inversion is performed, the inversion depth and node index are determined in the same way as for the duplication procedure (Algorithm 1 and 2). A definition of inversion is given in Definition 4.2 and the algorithmic implementation is described in Algorithm 4. An exemplification of the inversion procedure is illustrated in Figure 4.

**Definition 4.2** (Inversion). Let  $d$  be the inversion depth and let  $I = (g_f^0, \dots, g_f^d)$  be a tuple of active

Algorithm 3: The duplication procedure.

---

**Arguments**  
*G*: Original genome  
 $N_a$ : List of active function node numbers  
 $D_d$ : Maximum duplication depth  
**Return**  
 $\bar{G}$ : Mutated genome

```

1: function Duplication(G,  $N_a$ ,  $D_d$ )
2:   ▷ If less than two active nodes are active
3:   if  $|N_a| < 2$  then
4:     ▷ Discard procedure by returning the original genome
5:     return G
6:   end if
7:   ▷ Determine the depth stochastically and oriented with the maximum
8:   depth  $\leftarrow$  StochasticDepth( $D_d$ ,  $|N_a|$ )
9:   ▷ Determine start index in  $N_a$ 
10:  start  $\leftarrow$  StartIndex(depth,  $|N_a|$ )
11:  ▷ Calculate endpoint for the iteration
12:  end  $\leftarrow$  start + depth
13:  ▷ Get first function gene value
14:  func  $\leftarrow$  GetFunction( $N_a[\text{start}]$ )
15:  ▷ Initialize loop counter
16:   $i \leftarrow$  start + 1
17:  ▷ Iterate over the duplication depth
18:  while  $i \leq \text{end}$  do
19:    ▷ Get the next active function node number
20:    node  $\leftarrow N_a[i]$ 
21:    ▷ Determine the genome position
22:    pos  $\leftarrow$  PositionFromNodeNumber(node)
23:    ▷ Duplicate the first function
24:     $G[\text{pos}] \leftarrow$  func
25:    ▷ Loop counter increment
26:     $i \leftarrow i + 1$ 
27:  end while
28:  return G
29: end function

```

---

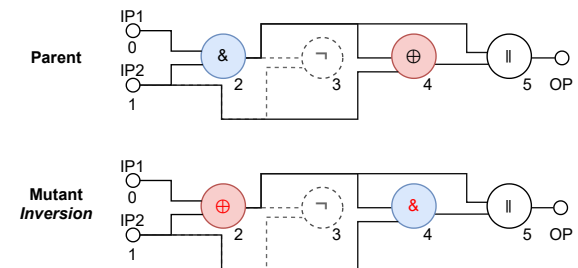
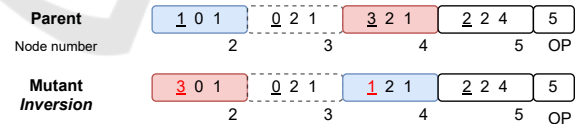


Figure 4: Exemplification of inversion in CGP: The function genes of node 2 and node 4 are inverted by a swap of the gene values. In this example, the inversion depth was set to one. The inverted function gene values are highlighted in red color.

Algorithm 4: The inversion procedure.

---

**Arguments**  
 $G$ : Original genome  
 $N_a$ : List of active function node numbers  
 $D_i$ : Maximum inversion depth  
**Return**  
 $\tilde{G}$ : Mutated genome

```

1: function Inversion( $G, N_a, D_i$ )
2:    $\triangleright$  If less than two active nodes are active
3:   if  $|N_a| < 2$  then
4:      $\triangleright$  Discard procedure by returning the original genome
5:     return  $G$ 
6:   end if
7:    $\triangleright$  Determine the depth stochastically and oriented with the maximum
8:    $\text{depth} \leftarrow \text{StochasticDepth}(D_d, |N_a|)$ 
9:    $\triangleright$  Determine start index in  $N_a$ 
10:   $\text{start} \leftarrow \text{StartIndex}(\text{depth}, |N_a|)$ 
11:   $\triangleright$  Calculate endpoint for the iteration
12:   $\text{end} \leftarrow \text{start} + \text{depth}$ 
13:   $\triangleright$  Division by two and ceiling for the iteration
14:   $\text{div} \leftarrow \lceil \frac{\text{depth}}{2} \rceil$ 
15:   $\triangleright$  Initialize loop counter
16:   $i \leftarrow 0$ 
17:  while  $i < \text{div}$  do
18:     $\triangleright$  Node number incremented from start
19:     $n_1 \leftarrow N_a[\text{start} + i]$ 
20:     $\triangleright$  Node number decremented from end
21:     $n_2 \leftarrow N_a[\text{end} - i]$ 
22:     $\triangleright$  Swap function gene values within the genome
23:     $G \leftarrow \text{SwapFunctionGenes}(G, n_1, n_2)$ 
24:     $\triangleright$  Loop counter increment
25:     $i \leftarrow i + 1$ 
26:  end while
27:  return  $G$ 
28: end function

```

---

function gene values which have been selected for the inversion procedure. Let  $\tilde{I} = (g_f^{d-0}, g_f^{d-1}, \dots, g_f^{d-i})$ , where  $i = d$  be a tuple of reversed function gene values which is obtained after the inversion procedure. Inversion is a variation operator for CGP that adapts chromosomal inversion by reversing the order of  $d + 1$  sequential active function genes  $I$  to  $\tilde{I}$ .

## 5 EVALUATION

### 5.1 Formulation of Research Question

The experiments of this work focus on the examination of the following research question:

**Research Question 1 (RQ1).** Can the use of duplication and inversion significantly contribute to the search performance of integer-based CGP for the tested Boolean function problems?

### 5.2 Benchmarks

For our experiments, we composed a diverse set of popular Boolean function problems. Since our experiments focus on the Boolean function domain, we want to ensure diversity for our evaluation. Therefore, our benchmark set covers five categories of Boolean functions which are often implemented in digital circuits. The benchmark set is shown in Table 1. In addition to the commonly used arithmetic and parity problems, we also included combinational and comparative problems such as the digital demultiplexer and comparator. Moreover, besides merely evaluating common single function problems, we also included underrepresented multi-function problems such as the adder/subtractor and the arithmetic logic unit (ALU). The function set of the ALU benchmark consisted of three logical and two arithmetic functions and is shown in Table 2. Since the majority of the problems are typically implemented with XOR gates, we decided to use a reduced function set  $\mathcal{F}_\ominus = \{\text{AND}, \text{OR}, \text{NAND}, \text{NOR}\}$  which increases the difficulty of these problems. An exception has been made for the four-bit digital multiplier problem for which we used an extended function set  $\mathcal{F}_\oplus = \{\text{AND}, \text{OR}, \text{NAND}, \text{NOR}, \text{NOT}, \text{XOR}, \text{XNOR}\}$ . All benchmarks used in our experiments are implemented for the ECJ CGP extension package and are available in the ECJ GitHub repository<sup>2</sup>. For the generation of the truth tables of the ALU benchmark, we developed a C++ based tool called *Boolean Benchmark Builder* which is also available on GitHub<sup>3</sup>. With our effort, we intend to promote the use of multi-function benchmarks in the Boolean problem domain. Overall, the emphasis of our benchmark set lies on multiple-output problems since the graph representation model of CGP is well-suited for this sort of problem. Another reason is the reported overuse of single-output benchmarks such as the parity even problem (McDermott et al., 2012; White et al., 2013). According to White et al. (White et al., 2013) multiple-output problems are considered to be a suitable replacement in the Boolean function domain.

### 5.3 Experimental Setup

To evaluate the search performance of the tested algorithms, we measured the number of fitness evaluations until the CGP algorithm terminated (*fitness-evaluations-to-termination*) as recommended by McDermott et al. (McDermott et al., 2012). Termination

<sup>2</sup><https://github.com/GMUEClab/ecj>

<sup>3</sup><https://github.com/RomanKalkreuth/boolean-benchmark-builder/>

Table 1: Benchmark set used for the experiments.

Arithmetic		
Problem	# Inputs	# Outputs
Adder 2-Bit	5	3
Adder 3-Bit	7	4
Adder 4-Bit	9	5
Subtractor 2-Bit	4	3
Adder/Subtractor 3-Bit	4	3
Multiplier 2-Bit	4	4
Multiplier 3-Bit	6	6
Multiplier 4-Bit	8	8
Combinational		
Demultiplexer 1x8-Bit (DeMUX-1x8)	3	8
Demultiplexer 1x16-Bit (DeMUX-1x16)	4	16
Comparative		
Comparator 3x1-Bit	3	9
Comparator 4x1-Bit	4	18
Parity		
Parity-Even 8-Bit	8	1
Parity-Even 9-Bit	9	1
Arithmetic/Logical		
Arithmetic Logic Unit 2-Bit (ALU 2-Bit)	7	3
Arithmetic Logic Unit 3-Bit (ALU 3-Bit)	9	4

Table 2: Function set of the arithmetic logic unit benchmark.

Opcode	Function	Description
000	&	Logical and
001		Logical or
010	$\oplus$	Exclusive or
011	+	Addition
100	-	Subtraction

was triggered when an ideal solution was found or a predefined budget of fitness evaluation was exceeded. For very complex problems, where it is likely that an algorithm does not find the ideal solution within a large budget of fitness evaluations, we additionally calculated the success rate. To evaluate the fitness, we defined the fitness value of an individual as the number of different bits to the corresponding truth table. When this number became zero, the algorithm successfully terminated. In addition to the mean values of the measurements, we calculated the standard deviation (SD), median (Q2) as well as lower and upper quartile (Q1 and Q3). The levels back parameter  $l$  was set to  $\infty$ . We performed 100 independent runs with different random seeds, except the complex and computing-intensive four-bit digital multiplier problem for which we only performed 30 runs. Due to its complexity, we defined a limit of  $10^8$  fitness evaluations. Meta-optimization experiments have been performed with the intention to compare the algorithms fairly and are described in more detail in the following subsection. All algorithms were compared on the same number of function nodes and the same setting of the  $\lambda$  parameter to exclude conditions, which can distort the search performance comparison. All algo-

Table 3: Identifiers for the tested CGP algorithms.

Identifier	Description
$(1 + \lambda)$ -CGP	$1 + \lambda$ selection strategy with neutral genetic drift
$(1 + \lambda)$ -CGP-DP	$(1 + \lambda)$ -CGP with gene duplication
$(1 + \lambda)$ -CGP-IN	$(1 + \lambda)$ -CGP with gene inversion
$(1 + \lambda)$ -CGP-DP/IN	$(1 + \lambda)$ -CGP with duplication and inversion

rithms which used our proposed methods are tested against the standard  $(1 + \lambda)$ -CGP which we declared as the baseline algorithm for our experiments. In our experiments, we exclusively used the single-row standard integer-based representation of CGP.

Since we cannot guarantee normally distributed values in our samples, we used the nonparametric two-tailed Mann-Whitney  $U$  test (Mann and Whitney, 1947) to evaluate statistical significance. More precisely, we tested the null hypothesis that two samples come from the same population (i.e. have the same median). In addition to the  $p$  values, the average number of fitness evaluations is denoted with  $a^\dagger$  if the significance level is  $p < 0.05$  or  $a^\ddagger$  if the significance level is  $p < 0.01$ .

## 5.4 Meta-optimization

We tuned relevant parameters for all tested CGP algorithms on the set of benchmark problems. Moreover, we used the meta-optimization toolkit of ECJ. The parameter space for the respective CGP algorithms, explored by meta-optimization, is presented in Table 4. For the meta-level, we used a canonical GA equipped with intermediate recombination and point mutation. Since GP benchmark problems can be very noisy in terms of finding the ideal solution, we oriented the meta-optimization with a common approach that has been used in previous studies (Kaufmann and Kalkreuth, 2017; Husa and Kalkreuth, 2018; Kalkreuth, 2021b). The meta-evolution process was repeated several times, and the most effective settings were compared to find the best setting. Finally, empirical fine-tuning of the respective parameters was performed.

## 5.5 Results

The results of our meta-optimization and search performance evaluation are presented in Table 6 and it is visible that either the use of duplication or inversion reduces the number of fitness evaluations to termination for the majority of our tested problems. Violin plots are provided in Figure 5. Please note that we evaluate the performance of the four-bit multiplier problem by the respective success rates which are presented in Table 5. For this kind of problem, we are primarily interested in how many solutions were found

Table 4: Parameter space explored by meta-optimization for the tested CGP algorithms.

(1 + $\lambda$ )-CGP		
$\lambda$	number of offspring	[1, 16]
$N$	number of nodes	[10, 10000]
$M_p$	point mutation rate[%]	[0.1, 5.0]

(1 + $\lambda$ )-CGP-DP		
$M_p$	point mutation rate[%]	[0.1, 5.0]
$M_d$	duplication rate[%]	[1.0, 20.0]
$D_d$	duplication depth	[1, 20]

(1 + $\lambda$ )-CGP-IN		
$M_p$	point mutation rate[%]	[0.1, 5.0]
$M_i$	inversion rate[%]	[1.0, 20.0]
$D_i$	inversion depth	[1, 20]

(1 + $\lambda$ )-CGP-DP/IN		
$M_p$	point mutation rate[%]	[0.1, 5.0]
$M_d$	duplication rate[%]	[1.0, 20]
$D_d$	duplication depth	[1, 20]
$M_i$	inversion rate[%]	[1.0, 20.0]
$D_i$	inversion depth	[1, 20]

Table 5: Success rates of the evaluation of the four-bit multiplier problem.

Algorithm	Success Rate [%]
(1 + $\lambda$ )-CGP	37
(1 + $\lambda$ )-CGP-DP	53
(1 + $\lambda$ )-CGP-IN	40
(1 + $\lambda$ )-CGP-DP/IN	23

at all by the tested algorithms since it has extraordinary complexity when compared to other benchmarks. The evolved parametrization pattern for the (1 +  $\lambda$ )-CGP are coherent with former findings (Kaufmann and Kalkreuth, 2017; Kalkreuth, 2021b). A setting of  $\lambda = 1$  turned out to be a general effective choice which is also coherent with former findings (Kalkreuth, 2021b).

## 6 ANALYSIS OF RESEARCH QUESTION

Our experiments demonstrate that the use of phenotypic inversion and duplication can significantly contribute to the search performance of standard integer-based CGP for our tested Boolean function problems. Besides observing a reduced number of fitness evaluations to termination, we also observed a clearly increased success rate on the complex four-bit multiplier problem for the duplication method. However, on some of our tested problems we did not observe a statistically significant result for a certain method or combination. Moreover, on certain problems, only the simultaneous use of our proposed methods led to a significant result. But considered holistically, our

initial results demonstrate the effectiveness of phenotypic duplication and inversion for the tested problems since the use of at least one method led to a significant reduction of the fitness evaluations or a clear increased success rate on every problem.

## 7 DISCUSSION

The initial results of this work allow certain points of criticism that are worthy of discussion. The first point which should be addressed is that on some problems a certain method or the simultaneous use failed to improve the search performance significantly. At this point, we can only hypothesize that the underlying semantics of certain problems are possibly suitable for one of our methods. For instance, duplication performed more effectively than inversion on the demultiplexer and comparator problems. In contrast, we achieved significant results with inversion on the multi-function problems such as the adder/subtractor and the ALU benchmark. Regarding the simultaneous use of our methods, we can only carefully hypothesize that it may add too much mutation strength to the evolutionary process of certain problems and therefore leads to no significant improvement. These points perhaps indicate the limitations of our methods but could open the opportunity to study and understand the search behavior of CGP on different problems in more detail. Since our methods require further parametrization which seems to be very problem-specific, a rule of thumb is difficult to determine. However, based on the results of our meta-optimization experiments we can narrow down the parametrization pattern. We observed that higher mutation rates greater than 12 % and depths greater than 6 function nodes were not successful on the tested problems. Moreover, on the most complex problems very low depths have been successful. Besides ensuring fair conditions for our comparative experiments, the motivation for the problem-specific tuning of the parameters of our methods was also to achieve more insight into the respective mutation strengths, composed of mutation rate and depth, that are required to achieve search performance enhancements. Another major motivation for the chosen experimental setup was to lay a foundation that can be used for further analysis of the search behavior in the future.

As a first step, our experiments focused on the evaluation of search effectiveness rather than runtime efficiency. However, we considered computational complexity for the design of our methods since the phenotype length is not altered. Consequently, our methods can be used without redetermination of active func-





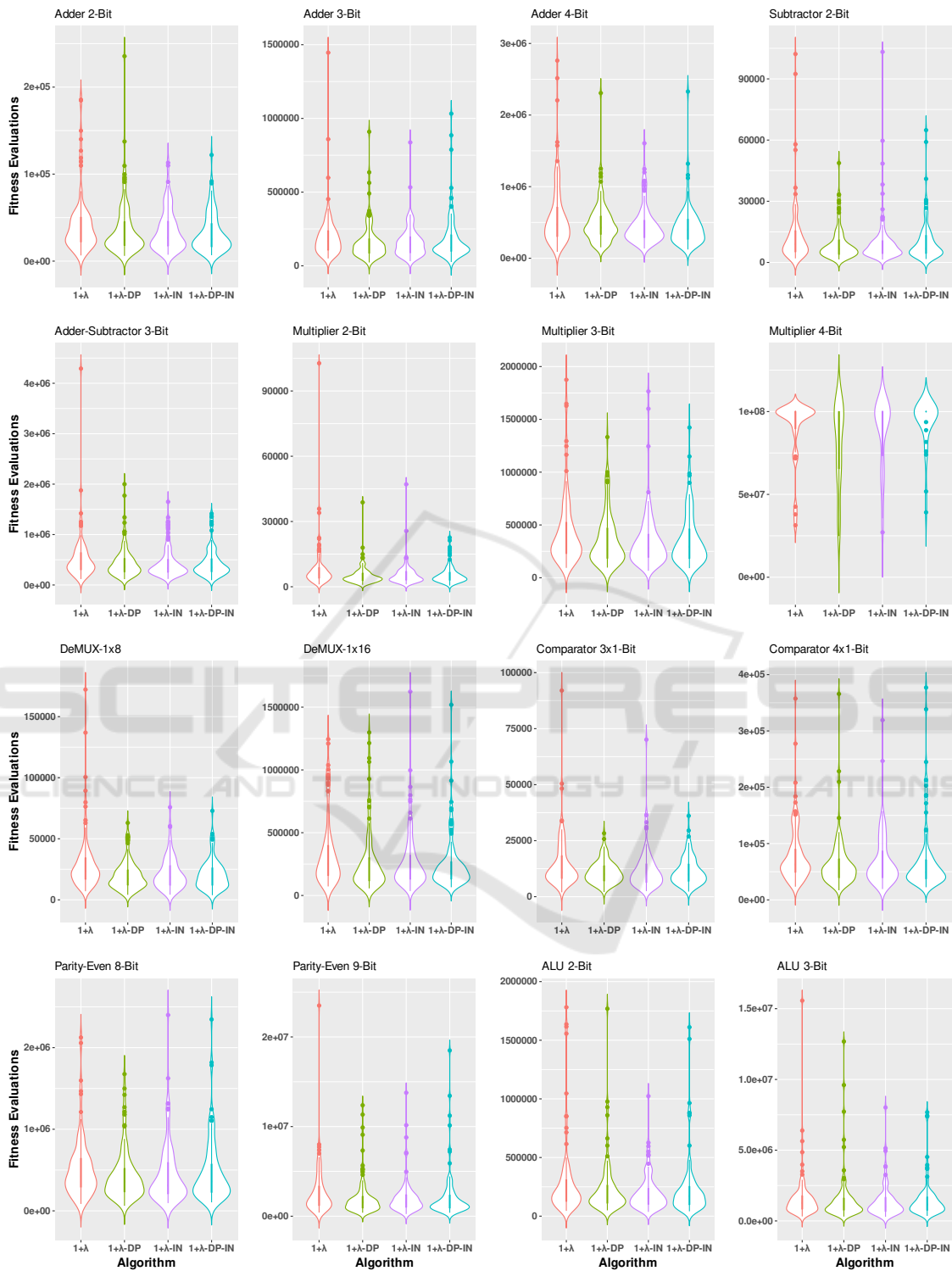


Figure 5: Violin plots for all tested problems and algorithms of our experiments.

multiple output topologies as well as multi-function problems. Overall, our results indicate that the use of our proposed methods can be beneficial for learn-

ing Boolean functions. Since our experiments primarily focused on the evaluation of the search performance and ensuring fair conditions through meta-

optimization, the next step is to study and analyze our methods with analytical experiments. Therefore, future work will focus on phenotype space and semantic analysis. Another part of our future work is devoted to the development of self-adaptation mechanisms to reduce parametrization and to increase effectiveness further.

## ACKNOWLEDGEMENTS

A large part of this work was carried out at the Department of Computer Science of the Technical University Dortmund. We thank Paul Kaufmann from the Westphalian University of Applied Sciences for providing the four bit digital adder and multiplier benchmark and sharing his empirical experience about the complexity of both problems. We also thank Andre Droschinsky, Marco Pleines and Fabian Ostermann from the Technical University Dortmund for reviewing the formalism and proofreading.

## REFERENCES

- Cramer, N. L. (1985). A representation for the adaptive generation of simple sequential programs. In Grefenstette, J. J., editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187, Carnegie-Mellon University, Pittsburgh, PA, USA.
- Ebner, M. (2012). Evolving concepts using gene duplication. In *Computational Modelling of Objects Represented in Images - Fundamentals, Methods and Applications III, Third International Symposium, CompIMAGE 2012, Rome, Italy, September 5-7, 2012*, pages 69–74. CRC Press.
- Češka, M., Matyáš, J., Mrázek, V., Sekanina, L., Vašíček, Z., and Vojnar, T. (2021). Sagtree: Towards efficient mutation in evolutionary circuit approximation. *Swarm and Evolutionary Computation*, page 100986.
- Forsyth, R. (1981). BEAGLE a Darwinian approach to pattern recognition. *Kybernetes*, 10(3):159–166.
- Goldman, B. W. and Punch, W. F. (2013). Length bias and search limitations in cartesian genetic programming. In *GECCO '13: Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 933–940, Amsterdam, The Netherlands. ACM.
- Goldman, B. W. and Punch, W. F. (2015). Analysis of cartesian genetic programming's evolutionary mechanisms. *IEEE Trans. Evol. Comput.*, 19(3):359–373.
- Hicklin, J. (1986). Application of the genetic algorithm to automatic program generation. Master's thesis, University of Idaho.
- Hodan, D., Mrazek, V., and Vasicek, Z. (2020). Semantically-oriented mutation operator in cartesian genetic programming for evolutionary circuit design. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference, GECCO '20*, page 940–948, New York, NY, USA. Association for Computing Machinery.
- Holland, J., Mahajan, M., Kumar, S., and Porwal, R. (1975). Adaptation in natural and artificial systems, the university of michigan press, ann arbor, mi. 1975. In *Applying genetic algorithm to increase the efficiency of a data flow-based test data generation approach*, pages 1–5.
- Husa, J. and Kalkreuth, R. (2018). A comparative study on crossover in cartesian genetic programming. In Castelli, M., Sekanina, L., Zhang, M., Cagnoni, S., and Garcia-Sanchez, P., editors, *EuroGP 2018: Proceedings of the 21st European Conference on Genetic Programming*, volume 10781 of LNCS, pages 203–219, Parma, Italy. Springer Verlag.
- Kalaganova, T. (1997). Evolutionary approach to design multiple-valued combinational circuits. In *Proceedings of the 4th International conference on Applications of Computer Systems (ACS'97)*, pages 333–339, Szczecin, Poland.
- Kalkreuth, R. (2019). Two new mutation techniques for cartesian genetic programming. In Guervós, J. J. M., Garibaldi, J. M., Linares-Barranco, A., Madani, K., and Warwick, K., editors, *Proceedings of the 11th International Joint Conference on Computational Intelligence, IJCCI 2019, Vienna, Austria, September 17-19, 2019*, pages 82–92. ScitePress.
- Kalkreuth, R. (2020). A comprehensive study on subgraph crossover in cartesian genetic programming. In Guervós, J. J. M., Garibaldi, J. M., Wagner, C., Bäck, T., Madani, K., and Warwick, K., editors, *Proceedings of the 12th International Joint Conference on Computational Intelligence, IJCCI 2020, Budapest, Hungary, November 2-4, 2020*, pages 59–70. SCITEPRESS.
- Kalkreuth, R. (2021a). *An Empirical Study on Insertion and Deletion Mutation in Cartesian Genetic Programming*, pages 85–114. Springer International Publishing, Cham.
- Kalkreuth, R. (2021b). *Reconsideration and extension of Cartesian genetic programming*. PhD thesis, Technical University of Dortmund, Germany.
- Kalkreuth, R., Rudolph, G., and Droschinsky, A. (2017). A new subgraph crossover for cartesian genetic programming. In Castelli, M., McDermott, J., and Sekanina, L., editors, *EuroGP 2017: Proceedings of the 20th European Conference on Genetic Programming*, volume 10196 of LNCS, pages 294–310, Amsterdam. Springer Verlag.
- Kalkreuth, R. T. (2021c). *Reconsideration and Extension of Cartesian Genetic Programming*. PhD thesis.
- Kaufmann, P. and Kalkreuth, R. (2017). An empirical study on the parametrization of cartesian genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '17*, pages 231–232, New York, NY, USA. ACM.
- Koza, J. (1990). Genetic Programming: A paradigm for genetically breeding populations of computer programs

- to solve problems. Technical Report STAN-CS-90-1314, Dept. of Computer Science, Stanford University.
- Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.
- Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts.
- Larrañaga, P., Kuijpers, C. M. H., Murga, R. H., Inza, I., and Dizdarevic, S. (1999). Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artif. Intell. Rev.*, 13(2):129–170.
- Manfrini, F. A. L., Bernardino, H. S., and Barbosa, H. J. C. (2016). A novel efficient mutation for evolutionary design of combinational logic circuits. In *Parallel Problem Solving from Nature - PPSN XIV - 14th International Conference, Edinburgh, UK, September 17-21, 2016, Proceedings*, pages 665–674.
- Mann, H. B. and Whitney, D. R. (1947). On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, 18(1):50 – 60.
- McDermott, J., White, D. R., Luke, S., Manzoni, L., Castelli, M., Vanneschi, L., Jaskowski, W., Krawiec, K., Harper, R., De Jong, K., and O'Reilly, U.-M. (2012). Genetic programming needs better benchmarks. In *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 791–798, Philadelphia, Pennsylvania, USA. ACM.
- Miller, J. F. (1999). An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1135–1142, Orlando, Florida, USA. Morgan Kaufmann.
- Miller, J. F., Thomson, P., and Fogarty, T. (1997). Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study. In *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*, pages 105–131. Wiley.
- Scott, E. O. and Luke, S. (2019). ECJ at 20: toward a general metaheuristics toolkit. In López-Ibáñez, M., Auger, A., and Stützle, T., editors, *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, pages 1391–1398. ACM.
- Sekanina, L. (2002). Image filter design with evolvable hardware. In *Applications of Evolutionary Computing, EvoWorkshops 2002: EvoCOP, EvoIASP, EvoS-TIM/EvoPLAN, Kinsale, Ireland, April 3-4, 2002, Proceedings*, pages 255–266.
- Turner, A. J. and Miller, J. F. (2013). Cartesian genetic programming encoded artificial neural networks: a comparison using three benchmarks. In *GECCO '13: Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 1005–1012, Amsterdam, The Netherlands. ACM.
- White, D. R., McDermott, J., Castelli, M., Manzoni, L., Goldman, B. W., Kronberger, G., Jaskowski, W., O'Reilly, U.-M., and Luke, S. (2013). Better GP Benchmarks: Community Survey Results and Proposals. *Genetic Programming and Evolvable Machines*, 14(1):3–29.