OPEN ACCESS
OPUS AUGSBURG

Universität Augsburg
Universitätsbibliothek

Refining mutation variants in Cartesian genetic programming

Henning Cui, Andreas Margraf, Jörg Hähner

Angaben zur Veröffentlichung / Publication details:

Cui, Henning, Andreas Margraf, and Jörg Hähner. 2022. "Refining mutation variants in Cartesian genetic programming." *Lecture Notes in Computer Science* 13627: 185–200. https://doi.org/10.1007/978-3-031-21094-5_14.

Nutzungsbedingungen / Terms of use:

licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under these conditions:
Deutsches Urheberrecht
Weitere Informationen finden Sie unter: / For more information see:
https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publiz/

Refining Mutation Variants in Cartesian Genetic Programming

Henning Cui[1][0000−0001−5483−5079](✉), Andreas Margraf[2][0000−0002−2144−0262], and Jörg Hähner[1][0000−0003−0107−264X]

[1] University of Augsburg, 86159 Augsburg, Germany
{henning.cui,joerg.haehner}@uni-a.de
[2] Fraunhofer IGCV, Institution for Casting, Composite and Processing Technology, 86159 Augsburg, Germany
andreas.margraf@igcv.fraunhofer.de

Abstract. In this work, we improve upon two frequently used mutation algorithms and therefore introduce three refined mutation strategies for Cartesian Genetic Programming. At first, we take the probabilistic concept of a mutation rate and split it into two mutation rates, one for active and inactive nodes respectively. Afterwards, the mutation method *Single* is taken and extended. *Single* mutates nodes until an active node is hit. Here, our extension mutates nodes until more than one but still predefined number n of active nodes are hit. At last, this concept is taken and a decay rate for n is introduced. Thus, we decrease the required number of active nodes hit per mutation step during CGP's training process. We show empirically on different classification, regression and boolean regression benchmarks that all methods lead to better fitness values. This is then further supported by probabilistic comparison methods such as the Bayesian comparison of classifiers and the Mann-Whitney-U-Test. However, these improvements come with the cost of more mutation steps needed which in turn lengthens the training time. The third variant, in which n is decreased, does not differ from the second mutation strategy listed. [3]

Keywords: Cartesian Genetic Programming · Genetic Programming · Evolutionary Algorithm · Mutation Strategy.

1 Introduction

Cartesian genetic programming (CGP) is a form of genetic programming and a nature inspired search heuristic. It can be used to automatically generate

[3] This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: https://doi.org/10.1007/978-3-031-21094-5_14. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms

programs and was first introduced by Miller [18] in 1999. Since its introduction it has been used for a multitude of applications like evolving electronic circuits [17], image processing [14] or evaluation of sensor data [3]. CGP employs a directed, acyclic graph-based representation. This means that these graphs consists of nodes which are arranged in a two-dimensional grid. These nodes can in turn be active or inactive, meaning that they do or do not contribute to an output. While some versions of CGP utilize or suggest the usage of a crossover operator, as is seen in Kalkreuth et al. [11] or Wilson et al. [22], they remain unused in standard CGP as it does not universally profit from this operation [10,16]. Thus, mutation is oftentimes the only genetic operation used. In the literature, there are two different mutation strategies which protrude and are frequently applied: a probabilistic approach or the *Single* mutation by Goldman and Punch [6]. As for the first one, every node has a chance of mutation while in the latter one, nodes are mutated until an active one is hit.

While the mutation function and the mutation rate, if needed, are very important for the success of CGP, there is little research performed trying to improve upon these two operations. Because of that, this work focuses on improving the probabilistic and *Single* mutation approach. At first, the probabilistic approach is further discussed. Normally, only a single mutation rate is used. However, we hypothesize that utilizing a different mutation rate for both active and inactive genes may lead to faster convergence. As for one *Single* mutation step, nodes are mutated until one active node is altered. This notion is suspended now and we allow the mutation of nodes to take place until n active nodes are mutated. To evaluate this concept, we experiment with different n values. We also investigate the effect of decreasing the mutation rate and number of active nodes changed over time.

We show on multiple datasets that our extensions achieve statistically significantly better fitness values on these datasets compared to the standard ones for the cost of a slower convergence rate.

We follow this introduction with Section 2, which gives a brief overview of related work. Afterwards, we present an introduction into CGP in Section 3. In Section 4, the refined mutation concepts are introduced. Furthermore, a short theoretical explanation is given as to why these concepts should lead to better fitness values. Afterwards, the experimental design is introduced in Section 5 and Section 6 presents and discusses our experimental results. Finally, we conclude our work in Section 7.

2 Related Work

In this work, the focus lies on extending the probabilistic mutation strategy as well as *Single*.

Previously, there has been other mutation strategies employed as well. Goldman and Punch [6] created other mutation algorithms alongside *Single*. Both algorithm extends the probabilistc mutation and the first one skips redundant evaluations, which leads to less computational time needed. The second algo-

rithm is more complex and not recommended by the authors, as it does not improve upon existing strategies.

In other studies, there has been some other work done investigating the effect of mutating inactive genes. The authors Turner and Miller [20] did an in depth investigation about genetic drift and genetic redundancy. Their findings, included but are not limited to, are that performance significantly worsens when there is no mutation of inactive genes. Furthermore, they hypothesized that a single mutation rate may be inferior to utilizing two mutation rates, for active and inactive genes respectively. By utilizing a mutation rate for inactive genes of up to 100%, they believe that this may enhance genetic drift and therefore CGP's performance.

Kaufmann and Kalkreuth [12] examined, among other things, the effects of *Single* mutation but mutated every inactive node while doing one mutation step. By doing so, they found an improvement and other, better suited mutation methods as well. One of these improvements would be to turn off the mutation of function genes when boolean regression benchmarks are used. Thus, they laid out the first steps for this work.

3 Cartesian Genetic Programming

This section gives a brief overview of CGP and its mutation algorithm.

3.1 Introduction to Cartesian Genetic Programming

CGP is traditionally represented as a directed, acyclic and feed-forward graph. It consists of *nodes* which are arranged in a $n_c \times n_r$ grid, wherein n_c declares the grid's number of columns and n_r indicates its number of rows. It takes one or multiple program inputs and feeds it forward through partially connected nodes before writing final values to output nodes.

Each node consists of a number of genes, namely a function gene, two connection genes and a parameter gene. The function gene addresses the encoded computational function of the node. If this function depends on a parameter, its value is taken from the parameter gene. The required input is taken from its connection genes, as they indicate where the node gets its data from. This can either be a program input or the output of a previous node. However, a node cannot get its input from an arbitrary former node, as the hyperparameter *levels-back* l restricts the connectivity of a node. Furthermore, the nodes are partitioned into two groups: active and inactive nodes. Active nodes contribute to a program output, while inactive nodes do not. The parameter l defines the number of columns to the nodes left it can receive its input from. Oftentimes, l is equal to n_c meaning that every node receives its input from every prior node.

In our work, a slightly modified version of CGP inspired by Harding et al. [8] and Leitner et al. [13] is used. Here, the handling of input and output are different from regular CGP and are adopted from *Self-Modifying CGP* [9]. Traditionally, if a program has n_i many inputs and requires n_o outputs, CGP contains $n_i + n_o$

additional genes. Each additional gene represents an address to its respective program input or output node. In our work though, no further input and output genes are used. The function set used is extended by four special functions taken from Harding et al. [9], indicating which node serves as an input or output.

An illustrative example of a genotype can be seen in Figure 1 as it shows a graph with $n_r = 1$ and $n_c = 7$. The function of the first two nodes are 'INPUT', indicating that the next program input is to be read. Afterwards, both input values are added in the third node. However, it does not link to a node with an output function, rendering it inactive. In the fourth node, the same inputs are subtracted and then added in node six with an additional program input taken from node five. At last, node seven indicates a program output.

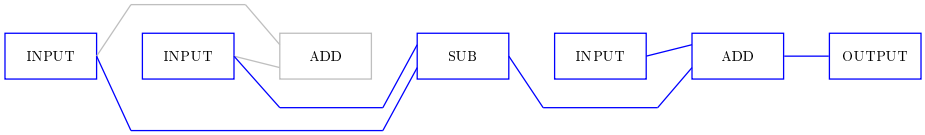

Fig. 1. An example genotype of CGP.

As is the case for many CGP algorithms, the standard $(1 + \lambda)$ Evolutionary Algorithm (EA) is used where the individual with the highest fitness is chosen as the next parent. This parent individual is taken to evolve λ offsprings. Additionally, neutral search is performed. This means that, when an offspring and the parent have the same fitness score, the offspring is always chosen as the next parent, even in the rare case that both offspring and parent are identical. This allows to generate better offsprings [20,23].

As is described by Miller [19], inactive nodes are part of *non coding* genes which are not used to provide an output. Such nodes can be exemplified by the third node in Figure 1 as it is part of the genotype but not contributing to the genotypes output.

By having non coding genes, genetic drift is allowed to occur as mutating them does not affect the fitness of the phenotype. These inactive nodes have the possibility to later be changed to active nodes when connection genes are mutated [7]. The works of Turner and Miller [20] and Yu and Miller [23] show that such neutral genes are highly beneficial for the evolutionary process of Evolutionary Algorithms and CGP as they help escape local optima and the evolutionary search of CGP. Albeit this theory is doubted by Goldman and Punch [5].

3.2 Mutation Algorithm

The most common mutation algorithms are of probabilistic nature [19] and *Single* [6].

As for the first one, the nodes are mutated according to a predefined mutation probability p. Furthermore, there is a distinction between *point mutation* and *probabilistic mutation*. For the point mutation, p percent of all nodes are randomly chosen and mutated. Concerning the probabilistic mutation, we iterate through every node with each node mutating with a probability p. However, with a probabilistic mutation strategy, it is possible that only inactive genes are mutated. Thus, it does not change the fitness value and nothing can be said about the quality of the changes.

Single, on the other hand, takes random nodes and mutates them until an active node is mutated. This offers the benefit to generate good results without setting a mutation rate. Goldman and Punch [6] evaluated *Single* on four boolean benchmarks and found that *Single* is preferred when the optimal mutation rate is unknown. Another benefit is that there is a guaranteed change in the phenotype. This avoids wasted evolutions where the phenotype stays the same. However, When the mutation rate can be optimized, a probabilistic mutation method is able to outperform *Single*.

4 Further Changes in the Mutation Algorithm

The following section gives an overview about the refined mutation strategies utilized in this work. Afterwards, a short theoretical explanation is given to motivate and reason their respective effectiveness.

4.1 Probabilistic Mutation

A caveat of the probabilistic mutation is that it does not differentiate between active and inactive nodes. We hypothesize that this could lead to a slower convergence and/or worse fitness. The main motivation is established by the findings of Turner and Miller [20]. This work addresses neutral drift in the context of CGP. They found that, among other things, not mutating inactive genes leads to a worse evolutionary search and in turn worse results overall. The authors Miller and Turner [20] also hypothesize that it could be favorable to explicitly use a higher mutation rate for inactive genes; and perhaps change it as high as up to 100%, meaning that every inactive node in the genotype is mutated after a single evaluation step. As having neutral drift is highly beneficial for the evolution of the CGP phenotype, such high mutation rates for inactive genes could lead to lower convergence rates.

To test this hypothesis, we split the mutation rate into two. A user defined probability p_i for the mutation of inactive nodes is introduced as well as a mutation rate p_a for active nodes. We compare different combinations of p_i and p_a for different classification and regression datasets as well as some selected boolean regression benchmarks.

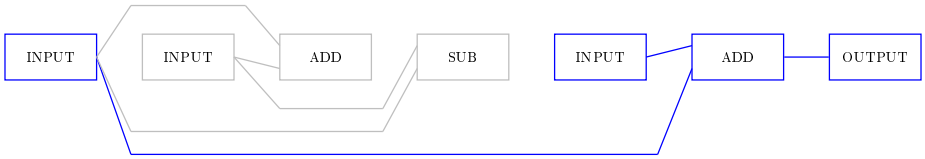

Fig. 2. An example of a changed phenotype as compared by Fig 1. By changing the connection gene of the sixth node, it is possible to alter the phenotype greatly. The computation does not rely on the second and fourth node anymore.

4.2 Single and Multiple Mutation

Single randomly changes inactive genes until one active gene is hit. Thus, only incremental changes are possible. However, these changes may affect the phenotype greatly by changing a connection gene as is exemplified in Fig. 2. The connection gene of the sixth node is mutated and now merely depends on the first and third input. Compared to the previous phenotype version in Fig 1, the computation of the program output does not rely on the second and fourth node anymore. Nevertheless, the single incremental change may, on the other hand, not affect the phenotype at all or only slightly. As is shown by Kaufmann and Kalkreuth [12], changing connection genes can improve the fitness at most and is the most meaningful mutation according to their estimation. Looking at the CGP implementation used in this work, every node has four possible genes which can be mutated: two connection genes, one function and one parameter gene. However, only the first connection gene guarantees a change in the phenotype. The second connection gene is only used if the corresponding function requires two inputs, which is needed by 6 out of 30 functions in our case. This leads to a chance of mutating a meaningful connection gene at about 29%. Albeit one could argue to exclusively mutate connection genes, only selected problems benefit from solely changing the in-going connections rendering this procedure not viable for every problem [12].

As is hypothesized by Goldman et al. [6], the incremental change of *Single* may perform worse on problems where larger changes per evaluation step are necessary. Thus, we introduce a modified version of *Single*: *Multi-n* and *Decreasing Multi-n* (*DMulti-n*).

Multi-n takes the concept of *Single* but mutates the genotype until n active nodes are mutated. It should be noted that, when an active node is hit, it is possible that the phenotype is completely changed. Thus, the list of active nodes may not stay the same during mutation and it is important to update the new active nodes before mutation is continued.

We speculate that *Multi-n* could lead to faster convergence due to several reasons. At first, more inactive genes are mutated per evolution step which leads to more genetic drift. The original authors of *Single* report an average of $\frac{t-a}{a+1} + 1$ expected mutations, where t is the number of total genes and a being the number of active parent genes. With *Multi-n*, we should expect an average of

$\prod_{i=1}^{n} \left(\frac{t-a_i}{a_i+1} + 1 \right)$ mutations with a_i being the number of active nodes after i active nodes have been hit. As is shown by Miller and Smith [15], typically there are more inactive genes than active ones. Considering *Multi-n*, this leads to even more mutated inactive genes per evolution cycle which should also introduce more genetic drift. Another reasoning in favor of *Multi-n* is a higher chance of a more impactful mutation in the phenotype, as more active nodes are mutated. A higher n leads to a higher probability of mutating a meaningful connection gene while also maintaining the chance to mutate function or parameter genes. With $n = 2$, the probability rises to 49% or even 74% for $n = 4$.

While *Multi-n* could lead to a faster convergence to possible solution spaces and local optima, it may also introduce a lot of changes per evolution step. It looses the ability to make small and incremental changes. *DMulti-n* introduces a slight modification to *Multi-n*. It can be imagined as *Multi-n*, but n decreases over time. Here, we employ a simple stepped decay rate:

$$n_{current} = n_{start} - \left\lfloor i_{current} \cdot \frac{n_{start}}{i_{total}} \right\rfloor \tag{1}$$

with $n_{current}$ being the current n value, n_{start} the initial n value, $i_{current}$ the current evaluation step and i_{total} the total evaluation steps to perform. We have, for example, the following two starting values: $i_{total} = 100$ and $n_{start} = 5$. This means that, in the beginning, we start mutating nodes until five active nodes are hit for each mutation step. After the 20th evaluation step, $n_{current}$ reduces to 4. Now the nodes are mutated until only four active ones are hit per mutation step. This repeats until a lower limit of $n_{current} = 1$ is reached and the training is finished with the equivalent of *Single* mutation.

5 Preliminaries

This section is now dedicated to the experiments conducted to explore the previously defined mutation concepts.

5.1 Experiment Description

As far as hyperparameters are concerned, default values found in the literature [19] are used. This means that $n_r = 1$ and $l = n_c$ are adopted. For the genotype, a length of $n_c = 100$ is employed. The number of maximum evaluations are $100,000$. If the parameter gene is needed, its value is randomly changed to a value in the range of $[-10, 10]$. Each experiment is repeated for a total of 15 times. The program inputs of the real-world datasets are standardized and standard k-fold cross validation with $k = 5$ is used when no train/test split is defined by the dataset.

For classification problems, the fitness function used in this work is subject to the Matthews Correlation Coefficient (MCC) [1]. Its score is in range $[-1, 1]$, with 1 and -1 indicating that every sample is correctly classified (but inverted,

in case of -1). A value of 0 indicates only falsely classified samples. Thus, the fitness value is defined as:

$$fitness = 1 - |MCC| \tag{2}$$

For regression problems, the mean squared error is used.

As optimization algorithm, a $(1 + \lambda)$ with $\lambda = 4$ evolutionary algorithm as mentioned in section 3.1 is utilized.

Furthermore as is discussed in the later section 5.2, two different kinds of datasets are used to evaluate the mutation operators. We employed real-world classification and regression datasets as well as symbolic regression benchmarks. Both utilize different function sets to accommodate to their respective problem. The function sets are specified in Table 1.

Table 1. Functions used in this work. The number of required inputs is given by arity.

Function Name	Arity	Description
Functions for Real-World Datasets		
INP, INPP, SKIP	0	Special input functions taken from [9]
OUTPUT	1	Special output functions taken from [9]
Add, Sub, Mul, Div	2	Standard mathematical functions
Addc, Subc, Mulc, Divc	1	Mathematical function, the operation takes one input i and the nodes' parameter as a constant c to calculate $i \circ c$
Sin, Cos, Tan, Tanh, Log, Log1p, Sqrt, Abs, Ceil, Floor	1	Standard mathematical functions
Max, Min	1	Compares the input with the nodes' parameter. Returns the bigger / lower value
Const	0	Returns the nodes' parameter
Negate	1	Returns the negated value
Functions for Symbolic regression datasets		
INP, INPP, SKIP	0	Special input functions taken from [9]
OUTPUT	1	Special output functions taken from [9]
Add, Sub, Mul, Div	2	Standard mathematical functions
Sin, Cos, Log	1	Standard mathematical functions

5.2 Datasets

We used a mixture of classification, regression and symbolic regression benchmarks to assess our mutation algorithms for multiple problem definitions. These datasets are chosen according to White et al. [21] as they surveyed and recommended multiple benchmarks for genetic programming. For classification and regression, we use Abalone, Breast Cancer, Credit, Forest Fire, Page Block and Spect, downloaded from the UCI repository [4]. We also employ symbolic regression benchmarks, namely Nguyen-7, Pagie-1 and Vlad-4 [21].

6 Experiments

In this work, several experiments are conducted to empirically test the different mutation strategies. We state the average fitness value as well as the average number of evaluations it takes until CGP converges, i.e. the number of evaluations until the best fitness result is achieved. Both values are then compared to their standard equivalent mutation strategy [4].

Furthermore, we utilize the Bayesian comparison of classifiers introduced by Benavoli et al. [2] to compare our models trained on classification datasets. The advantage here is that no null hypothesis is needed. Hence the results are presented as a triplet $(p_{default}, p_{equal}, p_{extend})$. These values indicate different probability values with $p_{default}$ stating the probability that the standard classifier is better; p_{equal} expresses the probability that the differences are within the region of practical equivalence and p_{extend} presents the probability that the modified mutation strategy is better. However, if the Bayesian comparison is not applicable since this only works for classification models, we utilize the Mann-Whitney-U-Test at $\alpha = 0.05$ between the default and the extended model. Again, we report our results as a triplet (U, Z, p) with U being the Mann-Whitney-U value, Z the z-statistic and p the p-value.

6.1 Impact of different probabilistic mutation strategies

To test the impact of splitting a single mutation rate p into two mutation rates p_a and p_i for the respective active and inactive nodes, we ran multiple experiments with different mutation rates and combinations thereof. For comparison, we utilize $p \in \{0.03, 0.1, 0.15, 0.2, 0.25\}$. As for utilizing two mutation rates, we ran experiments for every combination of the following probability values: $p_i \in \{0.1, 0.25, 0.5, 0.75, 1.0\}$ and $p_a \in \{0.03, 0.1, 0.15, 0.2, 0.25\}$. These mutation rates should cover a wide range of varieties while not being too finegrained or too fuzzy. For space-savings sake, the p and p_a values are averaged into p_{avg} or $p_{a,avg}$ respectively.

The averaged fitness values can be seen in Table 2. Here, even with averaged values there is a clear trend noticeable towards utilizing two different mutation rates instead of one as a split mutation rate almost always yields higher fitness values. Additionally, a higher mutation rate for inactive nodes oftentimes generate better results. This is in accordance to the same findings of other works, such as Kaufmann and Kalkreuth [12] who tested the influence of mutating every inactive node per mutation step. Still, it is not recommended to always set $p_i = 1.0$ as a more optimal p_i value is oftentimes lower than 1.0.

This finding is supported by the probabilistic evaluation in Table 3. Oftentimes, the comparison lies in favor of a split mutation rate or both methods being equal. However, only *Page Block* shows a trend towards the single mutation rate

[4] The code as well as its datasets preprocessing can be found in the following GitHub repository: https://github.com/CuiHen/Refining-Mutation-in-CGP.git

when a higher p_i value is used. Additionally, the p-values for *Forest Fire* are higher than 0.05, which means that those are not statistically significant.

Table 4 shows the convergence speed. Here, the results are averaged as is seen in Table 2. Interestingly, two different mutation rates oftentimes leads to more mutation steps needed for the model to converge. Moreover, the convergence speed for the best fitness results is oftentimes the lowest, too.

Table 2. The average fitness value achieved with a single and two mutation rates. The p and p_a values are averaged into p_{avg} or $p_{a,avg}$ respectively. Lower values are better. Bold symbols indicate the best values in the current row.

Dataset	p_{avg}	$p_{a,avg}$, $p_i = 0.1$	$p_{a,avg}$, $p_i = 0.25$	$p_{a,avg}$, $p_i = 0.50$	$p_{a,avg}$, $p_i = 0.75$	$p_{a,avg}$, $p_i = 1.0$
Abalone	6.59	5.96	5.84	5.99	5.84	**5.77**
Breast Cancer	0.074	0.054	0.049	0.042	0.049	**0.048**
Credit	0.229	0.230	0.241	**0.223**	**0.223**	0.229
Forest Fire	1.78	1.79	1.75	**1.72**	1.73	**1.72**
Heart Disease	0.763	0.743	0.744	0.736	**0.734**	0.742
Page Block	0.253	**0.212**	0.298	0.246	0.310	0.274
Spect	0.445	0.423	0.423	0.408	0.414	**0.406**
Nguyen-7	$1.42 \cdot 10^{-2}$	$5.44 \cdot 10^{-3}$	$6.16 \cdot 10^{-3}$	$5.94 \cdot 10^{-3}$	$\mathbf{5.30 \cdot 10^{-3}}$	$6.67 \cdot 10^{-3}$
Pagie-1	0.175	**0.093**	0.103	0.100	0.109	0.115
Vlad-4	0.037	**0.034**	**0.034**	**0.034**	0.035	0.035

6.2 Impact of *Multi-n* and DMulti-n

As for the other mutation strategy tested in this work, *Single* is compared to our *Multi-n* mutation strategy with $n \in \{2, 3, 4, 5\}$ at first.

Our results are shown in Table 5, 7 and 6. Here, we can see a clear trend towards higher n values, as they generally deliver better results. Interestingly, the highest n value of 5 seldom leads to the best results. It is possible that there are too many changes in the phenotype per mutation step so it becomes impossible to grasp more optimal results.

As the probability values in Table 6 suggest, the trend towards higher n values lead to better results, too. This reinforces the hypothesis that *Multi-n* can be better than *Single*. However, a caveat is that the datasets evaluated on the Mann-Whitney-U-Test oftentimes only significantly differ for $n > 3$.

Nonetheless, the convergence speed shows the same behavior as for the probabilistic approaches. This performance boost comes with a higher training time as most often CGP converges faster with a lower n value.

DMulti-n Afterwards, *DMulti-n* is utilized and n is decreased over time until $n = 1$. Interestingly, this does in fact not lead to better results because its

Table 3. The probability of one classifier being better than the other, evaluated on a single mutation rate compared to a split one. The p and p_a values are averaged into p_{avg} or $p_{a,avg}$ respectively. Datasets where the Bayesian comparison is used are marked with (B); the Mann-Whitney-U-Tests are marked with (MW). Every $(p_{a,avg}, p_i)$ value is compared to p_{avg}.

Dataset	$p_{a,avg}$, $p_i = 0.1$	$p_{a,avg}$, $p_i = 0.25$	$p_{a,avg}$, $p_i = 0.50$	$p_{a,avg}$, $p_i = 0.75$	$p_{a,avg}$, $p_i = 1.0$
Abalone (MW)	$(450, -2.6, 0.01)$	$(460, -2.9, 0.00)$	$(470, -3.0, 0.00)$	$(460, -2.9, 0.00)$	$(500, -3.6, 0.00)$
Breast Cancer (B)	$(0.00, 0.12, 0.88)$	$(0.00, 0.03, 0.97)$	$(0.00, 0.00, 1.0)$	$(0.00, 0.05, 0.95)$	$(0.00, 0.02, 0.98)$
Credit (B)	$(0.21, 0.63, 0.16)$	$(0.54, 0.26, 0.2)$	$(0.14, 0.49, 0.37)$	$(0.08, 0.61, 0.31)$	$(0.22, 0.58, 0.2)$
Forest Fire (MW)	$(310, -0.02, 0.98)$	$(370, -1.1, 0.25)$	$(370, -1.0, 0.30)$	$(360, -0.81, 0.42)$	$(390, -1.5, 0.14)$
Heart Disease (B)	$(0.00, 0.17, 0.83)$	$(0.00, 0.13, 0.87)$	$(0.00, 0.04, 0.96)$	$(0.00, 0.04, 0.96)$	$(0.00, 0.07, 0.93)$
Page Block (B)	$(0.08, 0.11, 0.81)$	$(0.87, 0.09, 0.04)$	$(0.32, 0.21, 0.47)$	$(0.82, 0.09, 0.09)$	$(0.59, 0.14, 0.27)$
Spect (B)	$(0.10, 0.21, 0.69)$	$(0.08, 0.22, 0.70)$	$(0.00, 0.02, 0.98)$	$(0.00, 0.05, 0.95)$	$(0.00, 0.03, 0.97)$
Nguyen-7 (MW)	$(490, -3.5, 0.00)$	$(470, -3.1, 0.00)$	$(480, -3.2, 0.00)$	$(470, -3.0, 0.00)$	$(460, -2.9, 0.00)$
Pagie-1 (MW)	$(510, -3.7, 0.00)$	$(490, -3.5, 0.00)$	$(510, -3.8, 0.00)$	$(480, -3.2, 0.00)$	$(510, -3.8, 0.00)$
Vlad-4 (MW)	$(520, -4.1, 0.00)$	$(500, -3.7, 0.00)$	$(500, -3.6, 0.00)$	$(510, -3.8, 0.00)$	$(500, -3.5, 0.00)$

Table 4. The average number of evaluations until convergence is achieved; given for a single and two mutation rates. The p and p_a values are averaged into p_{avg} or $p_{a,avg}$ respectively. Lower values are better. The number of evaluations for the best model (i.e. the best fitness, cf. Tab. 2) is underlined; the lowest number indicates the fastest convergence is in bold.

Dataset	p_{avg}	$p_{a,avg}$, $p_i = 0.1$	$p_{a,avg}$, $p_i = 0.25$	$p_{a,avg}$, $p_i = 0.50$	$p_{a,avg}$, $p_i = 0.75$	$p_{a,avg}$, $p_i = 1.0$
Abalone	**663**	2034	2531	2280	2352	2654
Breast Cancer	**962**	1406	1275	1412	1286	1724
Credit	**759**	1086	1108	1412	1653	847
Forest Fire	**866**	2077	1978	1826	2473	2490
Heart Disease	**759**	1440	1426	1702	2185	1364
Page Block	2142	2549	2210	2787	**1871**	2818
Spect	**834**	1244	1461	1398	1007	1689
Nguyen-7	276	316	282	300	468	**266**
Pagie-1	**700**	2330	2002	2021	1901	2433
Vlad-4	**819**	1553	1616	2213	2094	1959

Table 5. The average fitness value for *Single* and *Multi-n*. Lower values are better. Bold symbols indicate the best values in the current row.

Dataset	*Single*	$n = 2$	$n = 3$	$n = 4$	$n = 5$
Abalone	6.07	5.84	5.71	**5.60**	5.64
Breast Cancer	0.059	0.042	0.039	**0.035**	0.034
Credit	0.389	0.258	**0.222**	0.239	0.228
Forest Fire	1.75	1.72	1.71	**1.67**	1.69
Heart Disease	0.756	0.752	**0.736**	0.739	0.742
Page Block	0.273	0.282	0.255	**0.238**	0.239
Spect	0.489	0.433	0.445	**0.391**	0.408
Nguyen-7	$5.60 \cdot 10^{-3}$	$\mathbf{4.15 \cdot 10^{-3}}$	$5.19 \cdot 10^{-3}$	$6.15 \cdot 10^{-3}$	$7.04 \cdot 10^{-3}$
Pagie-1	0.075	0.056	0.044	0.042	**0.036**
Vlad-4	0.034	0.035	0.034	**0.031**	0.032

Table 6. The probability of one classifier being better than the other, evaluated on *Single* compared to *Multi-n*. Datasets where the Bayesian comparison is used are marked with (B); the Mann-Whitney-U-Tests are marked with (MW). Every n value is compared to *Single*.

Dataset	$n = 2$	$n = 3$	$n = 4$	$n = 5$
Abalone (MW)	$(130, -0.54, 0.59)$	$(150, -1.6, 0.11)$	$(170, -2.3, 0.02)$	$(170, -2.3, 0.02)$
Breast Cancer (B)	$(0.01, 0.25, 0.73)$	$(0.01, 0.20, 0.78)$	$(0.00, 0.08, 0.92)$	$(0.01, 0.11, 0.88)$
Credit (B)	$(0.04, 0.02, 0.94)$	$(0.01, 0.01, 0.99)$	$(0.02, 0.02, 0.96)$	$(0.01, 0.01, 0.98)$
Forest Fire (MW)	$(130, -0.58, 0.56)$	$(140, -1.2, 0.21)$	$(150, -1.6, 0.11)$	$(120, -0.41, 0.68)$
Heart Disease (B)	$(0.23, 0.41, 0.36)$	$(0.02, 0.22, 0.76)$	$(0.04, 0.29, 0.67)$	$(0.06, 0.35, 0.59)$
Page Block (B)	$(0.49, 0.19, 0.32)$	$(0.26, 0.17, 0.57)$	$(0.25, 0.10, 0.64)$	$(0.12, 0.14, 0.74)$
Spect (B)	$(0.01, 0.04, 0.95)$	$(0.02, 0.07, 0.91)$	$(0.00, 0.00, 1.00)$	$(0.00, 0.00, 1.00)$
Nguyen-7 (MW)	$(150, -1.50, 0.15)$	$(120, -0.37, 0.71)$	$(99, -0.54, 0.59)$	$(79, -1.4, 0.17)$
Pagie-1 (MW)	$(140, -1.10, 0.28)$	$(160, -1.80, 0.07)$	$(160, -1.9, 0.06)$	$(170, -2.4, 0.02)$
Vlad-4 (MW)	$(120, -0.25, 0.80)$	$(110, 0.00, 1.00)$	$(210, -3.9, 0.00)$	$(170, -2.4, 0.02)$

Table 7. The average number of evaluations until CGP converges for *Single* and *Multi-n*. The lower the better. The number of evaluations for the best model (i.e. the best fitness, cf. Tab. 5) is underlined; the lowest number indicates the fastest convergence is in bold.

Dataset	*Single*	$n = 2$	$n = 3$	$n = 4$	$n = 5$
Abalone	2084	**2043**	2244	2239	2230
Breast Cancer	1497	**1319**	1632	2085	1561
Credit	1553	2113	1734	2175	**1440**
Forest Fire	**1816**	1946	2437	2462	2264
Heart Disease	2174	**2042**	2802	3048	2805
Page Block	**846**	957	1044	1194	1041
Spect	**461**	891	724	1353	1508
Nguyen-7	108	**21**	64	38	20
Pagie-1	1866	2141	**1849**	2481	1997
Vlad-4	2149	**2052**	2494	3309	2527

fitness values and training times are very identical to the results in Table 5. This is why we did not list the results separately. Moreover, this implies that the later decreases of n do not improve the fitness values as all models converge before n is decreased. Hence it can be said that there is no need for smaller and more incremental changes in this setting. This leads to the assumption that there is no gain derived from decreasing n over time.

However, these findings are highly counter intuitive to the findings of *Multi-n* and its slight decrease in performance between $n = 4$ and $n = 5$ or $n = 3$ and $n = 5$. One may assume that, by decreasing n, CGP should be able to counter balance the negative effects of a starting value of $n = 5$. The reason is that there is a big portion of mutation steps utilizing lower n values. Thus, it should be able to balance the best results of *Multi-n* seen in Table 5. This may, in theory, come with the cost of having a lower convergence rate. Nevertheless, this is not the case here. Our theory is that CGP seems to get stuck in a local optimum after the first iterations. In later stages of training, decreasing the hyperparameter n apparently cannot significantly help to increase performance. Further research should be done in this direction to explore the reasoning behind this phenomenon.

7 Conclusion

In this work, we empirically evaluated the fitness values as well as the convergence speed for three new mutation algorithms. The mutation based on probability as well as *Single* was extended. At first, two different mutation rates for active and inactive nodes respectively were employed. On the utilized datasets, we found that it is favorable to utilize two mutation rates. However, while a higher rate for inactive genes is encouraged, the optimal probability differs between the problems. Albeit most of the times, it is better to utilize a mutation rate of 50% or higher.

Afterwards, we extended the *Single* Algorithm to *Multi-n* and *DMulti-n*. We found that, in these datasets, *Multi-n* always outperforms *Single* but its improvements decline for high values of n.

However, there is the caveat that all of these improvements in fitness value oftentimes comes with the downside of longer training time. Most of the time, a single mutation rate or *Single* need less evaluation steps until a solution was found.

As for future work, the introduced concepts could be merged and used interchangeably. As Goldman et al. [6] found, *Single* performs better than the probabilistic approach when the best mutation rate is unknown. Both of these mutation strategies could be changed depending on the current training status and or fitness value.

Another possibility is to keep two mutation rates for active and inactive genes. In addition, these mutation rates could change with their position in the grid. In the work of Goldman and Punch [5], they found a high positional bias in CGP as well as challenged the status quo in terms of neutral search, bloat

and the importance of genetic drift. By applying higher mutation rates for genes positioned in the back, a partial success could be achieved to reduce CGP's positional bias.

At last, it is unclear as to why *DMulti-n* does not show an advantage over *Multi-n* as well as why it does not improve when n is lowered. Further works might investigate into this phenomena, possibly leading into deeper understanding of CGP.

References

1. Baldi, P., Brunak, S., Chauvin, Y., Andersen, C.A.F., Nielsen, H.: Assessing the accuracy of prediction algorithms for classification: an overview . Bioinformatics **16**(5), 412–424 (05 2000). https://doi.org/10.1093/bioinformatics/16.5.412
2. Benavoli, A., Corani, G., Demšar, J., Zaffalon, M.: Time for a change: a tutorial for comparing multiple classifiers through bayesian analysis. The Journal of Machine Learning Research **18**(1), 2653–2688 (2017)
3. Bentley, P.J., Lim, S.L.: Fault tolerant fusion of office sensor data using cartesian genetic programming. In: 2017 IEEE Symposium Series on Computational Intelligence (SSCI). pp. 1–8. IEEE (2017)
4. Dua, D., Graff, C.: UCI machine learning repository (2017), http://archive.ics.uci.edu/ml
5. 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. p. 933–940. GECCO '13, Association for Computing Machinery, New York, NY, USA (2013)
6. 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 Berlin Heidelberg, Berlin, Heidelberg (2013)
7. Goldman, B.W., Punch, W.F.: Analysis of cartesian genetic programming's evolutionary mechanisms. IEEE Transactions on Evolutionary Computation **19**(3), 359–373 (2015)
8. Harding, S., Graziano, V., Leitner, J., Schmidhuber, J.: Mt-cgp: Mixed type cartesian genetic programming. In: Proceedings of the 14th annual conference on Genetic and evolutionary computation. pp. 751–758 (2012)
9. Harding, S.L., Miller, J.F., Banzhaf, W.: Self-modifying cartesian genetic programming. In: Cartesian Genetic Programming, pp. 101–124. Springer (2011)
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 International Publishing, Cham (2018)
11. 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 International Publishing, Cham (2017)
12. Kaufmann, P., Kalkreuth, R.: On the parameterization of cartesian genetic programming. In: 2020 IEEE Congress on Evolutionary Computation (CEC). pp. 1–8 (2020)

13. Leitner, J., Harding, S., Forster, A., Schmidhuber, J.: Mars terrain image classification using cartesian genetic programming. In: Proceedings of the 11th International Symposium on Artificial Intelligence, Robotics and Automation in Space, i-SAIRAS 2012. pp. 1–8. European Space Agency (ESA) (2012)
14. Margraf, A., Stein, A., Engstler, L., Geinitz, S., Hahner, J.: An evolutionary learning approach to self-configuring image pipelines in the context of carbon fiber fault detection. In: 2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA). pp. 147–154. IEEE (2017)
15. Miller, J., Smith, S.: Redundancy and computational efficiency in cartesian genetic programming. Evolutionary Computation, IEEE Transactions on **10**, 167 – 174 (05 2006)
16. Miller, J., Thomson, P.: Cartesian genetic programming. In: Proc. European Conference on Genetic Programming. vol. 1802, pp. 121–132. Springer (2000)
17. Miller, J., Thomson, P., Fogarty, T., Ntroduction, I.: Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study. Genetic Algorithms and Evolution Strategies in Engineering and Computer Science (10 1999)
18. 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 - Volume 2. p. 1135–1142. GECCO'99, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1999)
19. Miller, J.F.: Cartesian genetic programming: its status and future. Genetic Programming and Evolvable Machines **21**(1), 129–168 (2020)
20. Turner, A., Miller, J.: Neutral genetic drift: an investigation using cartesian genetic programming. Genetic Programming and Evolvable Machines **16** (05 2015)
21. White, D., Mcdermott, J., Castelli, M., Manzoni, L., Goldman, B., Kronberger, G., Jaśkowski, W., O'Reilly, U.M., Luke, S.: Better gp benchmarks: Community survey results and proposals. Genetic Programming and Evolvable Machines **14**, 3–29 (03 2013)
22. Wilson, D.G., Miller, J.F., Cussat-Blanc, S., Luga, H.: Positional cartesian genetic programming. arXiv preprint arXiv:1810.04119 (2018)
23. Yu, T., Miller, J.: Neutrality and the evolvability of boolean function landscape. In: European Conference on Genetic Programming. pp. 204–217. Springer (2001)