Accelerating Convergence in Cartesian Genetic Programming by Using a New Genetic Operator

Andreas Meier
Group Research
Volkswagen AG
38436 Wolfsburg, Germany
andreas.meier1@volkswagen.de

Mark Gonter
Group Research
Volkswagen AG
38436 Wolfsburg, Germany
mark.gonter@volkswagen.de

Rudolf Kruse
Faculty of Computer Science
University of Magdeburg
39114 Magdeburg, Germany
kruse@iws.cs.uni-magdeburg.de

ABSTRACT

Genetic programming algorithms seek to find interpretable and good solutions for problems which are difficult to solve analytically. For example, we plan to use this paradigm to develop a car accident severity prediction model for new occupant safety functions. This complex problem will suffer from the major disadvantage of genetic programming, which is its high demand for computational effort to find good solutions. A main reason for this demand is a low rate of convergence. In this paper, we introduce a new genetic operator called *forking* to accelerate the rate of convergence. Our idea is to interpret individuals dynamically as centers of local Gaussian distributions and allow a sampling process in these distributions when populations get too homogeneous. We demonstrate this operator by extending the *Cartesian Genetic Programming* algorithm and show that on our examples convergence is accelerated by over 50% on average. We finish this paper with giving hints about parameterization of the forking operator for other problems.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program synthesis*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Algorithms, Design, Performance

Keywords

Cartesian Genetic Programming, optimization, genetic operator

1. INTRODUCTION

In the field of evolutionary algorithms, genetic programming (GP) is a paradigm which enables automatic derivation of programs that solve problems. In contrast to other algorithms like artificial neural networks, GP algorithms learn interpretable models so that e.g. safety applications may be realized more easily. GP got significant attention after Koza has applied it to solve complex problems [8]. One of these problems is called *symbolic regression*, which seeks to find mathematical expressions describing the relation between input and output data. In contrast to parametrical regression, symbolic regression builds up complex formulae automatically by combining variables, basic operators (+,-,*,/) and functions (for example sin, cos or tan). Recently, Schmidt and Lipson used symbolic regression to discover natural laws describing kinematics of different oscillators [16].

Although GP is able to solve difficult problems, its major disadvantage is low efficiency. We define *efficiency* as the required computational effort for an evolutionary algorithm to achieve convergence. One measure for efficiency is the average number of generations it takes to find a solution. Another approach is to measure computation time until convergence is achieved. Schmidt and Lipson used a 32-core implementation which needs 30 to 40 hours to learn a solution for a specific oscillator problem [16]. Since they optimized their algorithm to require only 7 to 8 hours, we consider the original algorithm as being less efficient.

In contrast to accelerating execution by parallelization [9], we focus on increasing efficiency of GP algorithms by introducing the *forking* operator. This operator uses population statistics to interpret individuals dynamically as distributions in solution space. In that way, the forking operator samples new individuals when populations get too homogeneous. As a result, diversity is increased, which may lead to better convergence behavior. We demonstrate this improved convergence behavior by extending the *Cartesian Genetic Programming* (CGP) algorithm with the forking operator. As Miller describes, CGP represents programs as graphs in which nodes perform functions and the connection between the nodes controls the data flow [12]. Thus, the learning task consists of assigning functions to nodes and connecting the nodes with each other.

In this paper, we explain the background and relevant literature at first. Next, we give reasons for low efficiency of real-valued CGP and explain the forking operator. Afterwards, we present and discuss results for four symbolic regression problems used to measure efficiency of our extended CGP algorithm. We finish this paper with discussing conclusions and suggestions for future research questions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'13, July 6–10, 2013, Amsterdam, The Netherlands.
Copyright 2013 ACM 978-1-4503-1963-8/13/07 ...$15.00.

2. BACKGROUND

In this section, we explain Cartesian Genetic Programming, which is the GP algorithm that we extended with the forking operator. Furthermore, we give an overview of relevant literature.

2.1 Cartesian Genetic Programming

GP is a data-driven paradigm that seeks to evolve programs by applying genetic operators to individuals. Although first steps towards GP have already been taken in the 1970s [4, 15], this paradigm gained significant attention after Koza applied it to complex optimization and search problems [8]. Koza represented programs as LISP parse trees, which combine multiple simple functions to more complex functions. In 2000, Miller and Thomson introduced another algorithm for GP called *Cartesian Genetic Programming* (CGP) [14].

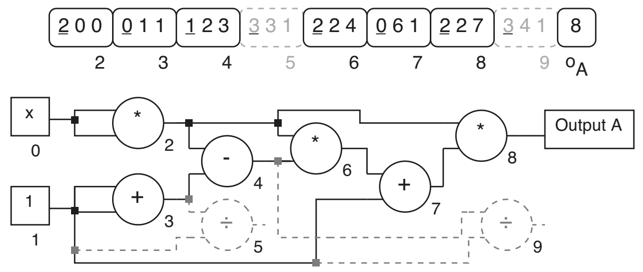


Figure 1: Genotype and phenotype for CGP individuals, source: Clegg et al.[2]

CGP represents programs as indexed, acyclic and directed graphs via genotype-phenotype-mapping. Originally, these graphs were structured as a rectangular grid, but later work focused on a graph with only one row [2]. In figure 1, we show an example of a genotype and its corresponding phenotype. The sequence of numbers at the top of the figure represents the genotype, in which each group maps to one node of the graph except for the last group. These numbers are indexes and either describe the function to perform (underlined number) or the index of the inputs to work on (non-underlined number). The index in the last group specifies the output of the program. The algorithm evaluates a program via recursive backward search through the directed graph. It starts from program output and processes all nodes directly or indirectly linked to it. In that way, only active nodes are processed whereas inactive nodes, which are not connected to the output, remain untouched. Thus, the given example specifies the function $x^6 - 2x^4 + x^2$.

2.2 Related work

The presented genotype-phenotype-mapping offers some properties, which were investigated in more detail. At first, we note that the genotype can encode functionality that is not relevant for the phenotype. In figure 1, the dashed groups and the corresponding dashed, inactive nodes do not have any direct or indirect connection to the output. Therefore, these genes can be arbitrarily set, the phenotype will not change. Miller and Thomson described this property as an example of *neutrality* and published work emphasizing its importance [14]. This genotype redundancy can improve efficiency when unused genes are altered and used in future individuals. Another important property is that integer-based CGP usually only uses mutation but not crossover operators. Koza underlined the importance of crossover operators for his tree-based program representation [8]. According to him, no run on his experiments, which used only a mutation operator and a fitness-proportionate reproduction, has ever produced a solution to any problem. In contrast, Clegg et al. showed that omitting a crossover operator and relying only on a mutation operator can produce a solution in a CGP algorithm [2]. Clegg et al. also evaluated different crossover operators, but reasoned that swapping integers as crossover operator influences efficiency negatively. Therefore, they changed gene encoding to real values from the interval $[0, 1]$ as shown in figure 2. For receiving indexes for the lookup tables, gene values are multiplied with the function or input count. This modification of gene encoding enables the use of a weighted average crossover operator. Without a crossover operator, integer-based and real-valued CGP achieve similar efficiency. In contrast, adding a weighted average crossover operator to real-valued CGP improves efficiency significantly.

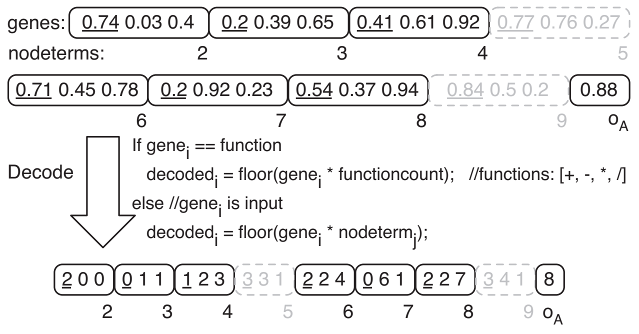


Figure 2: Decoding from real-valued to integer-based genotype, bases on: Clegg et al.[2]

Other approaches have also been investigated to improve efficiency of CGP. Miller analyzed the population size of a CGP algorithm, but also show node count influences efficiency [11]. With one exception out of 12 Boolean symbolic regression problems, lower population sizes tend to improve efficiency. It also improves when node count is increased. However, Miller stated that these findings may be limited to this type of problems. In a later publication, Miller and Smith investigated the correlation between efficiency and node count in more detail [13]. The authors presented a 2-bit multiplier problem and showed that with increasing node count, efficiency increases as well. According to them, the reason for this correlation is increased redundancy utilized by the mutation operator. Since a larger CGP individual can hold multiple inactive programs in parallel, the mutation operator may change the active program by deactivating or activating nodes. According to Miller, this behavior is a confirmation for the importance of neutrality [12].

Low efficiency is also related to *premature convergence*. Premature convergence is caused by decreasing population diversity and describes the effect that a population converges too early, thus getting stuck in local optima [3]. Despite increasing mutation rate [1], another solution is to continuously create new random individuals. Hornby introduced the *Age-Layered Population Structure* (ALPS) which tracks the age of each individual and allows genetic operations only to work on individuals of similar age [5]. Hornby showed that this paradigm helps to improve the fitness significantly for an antenna design problem. Later, Slanỳ combined the ALPS paradigm with CGP and demonstrated a better performance for the evolution of image operators [17]. However, for more complex problems, the performance gain appears to be less superior. Lehman and Stanley replaced the fitness function in a GP algorithm with a novelty function, which helps to maintain population diversity [10]. Although the novelty function leads to better and smaller solutions in the presented experiments, algorithms using a fitness function may find a solution faster.

Tu and Lu presented a genetic algorithm, which interprets individuals not only as points but also as stochastic regions in solution space [18]. They consider each individual as mean of a Gaussian distribution with an adaptable variance and sample over this distribution to produce and evaluate new individuals. In that way, each individual also covers slight modifications, which helps to improve efficiency. In their experiments, Tu and Lu showed a large increase in efficiency due to this sampling. Four years later, they have published a correction because of a mistake in their original implementation [19]. This mistake lead to much better results, but was originally not discovered due to a special behavior of the example problems. Although the corrected implementation does not achieve satisfactory efficiency, sampling likely good genes during evolution may be beneficial anyway. Thus, we use and modify this idea for our forking operator.

3. FORKING OPERATOR

In this section, we present our preliminary considerations that lead to the development of the new genetic operator called *forking*. Afterwards, we explain the operator and show how it integrates into the evolutionary process.

3.1 Preliminary considerations

In the previous section, we present approaches to improve efficiency of evolutionary algorithms, especially for CGP. Even for real-valued CGP, which we use as base for our operator, we often note low efficiency. At the beginning of evolution, the population converges fast towards good solutions whereas the rate of convergence decreases notably in the end. One way to avoid this low rate of convergence is to define an easily reachable convergence threshold. For most problems, we want to find very good or even perfect solutions so that this threshold must be kept very low. Therefore, other solutions have to be found to improve efficiency.

From our experience, the main reason for this convergence behavior is that individuals are getting more and more similar during evolution. Thus, the algorithm evaluates (nearly) identical individuals over and over again so that fewer regions of solution space are covered. This reduces the gene pool significantly so that crossover operators work less efficiently. From our perspective, real-valued CGP even intensifies this problem due to its genotype-phenotype-mapping.

The problem of genotype-phenotype-mapping is its large difference in complexity of genotype and phenotype spaces. Although all individuals in one population have most likely different genotypes, the probability of sharing the same phenotype is much higher. To underline this statement, we consider a example given in figure 3.

The example consists of one node with four possible functions, two inputs and one program output. Each gene is encoded by a 32-bit floating-point variable, which is a common size in modern programming languages like C [7]. According to IEEE 754 standard, a 32-bit floating-point variable encodes $2^{32} - 2^{25}$ different normal numbers [6]. Thus, the combination of four genes leads to $(2^{32} - 2^{25})^4$ =∼3,298e+38 possible genotypes. However, there are only 48 possible phenotypes. That means ∼3,298e+38 possible genotypes encode only 48 possible phenotypes, which prove the mapping as being very robust. Even if we use a more efficient encoding and pack all genes into one 32-bit floating-point variable, we still have $2^{32} - 2^{25} = 4,261,412,864$ possible genotypes. Naturally, this is a very simplified example to emphasize the disparity between phenotype and genotype spaces. Usually, CGP programs consist of more nodes, have more node functions and also more inputs. Furthermore, the gene encoding may be more efficient, but for real-valued CGP genotype and phenotype spaces will likely still maintain a large difference in complexity. However, CGP phenotype complexity prevents an exhaustive search for the best solution.

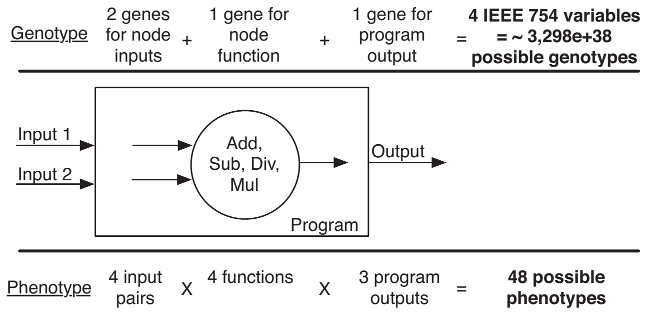
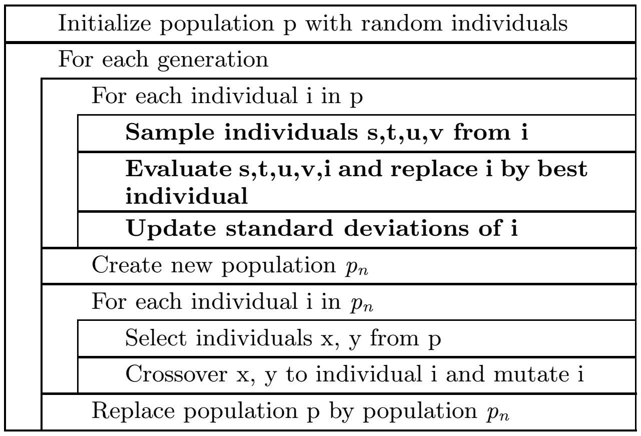


Figure 3: Example for difference in complexity of genotype and phenotype spaces

This large difference in complexity of genotype and phenotype spaces leads to different problems, which common genetic operators cannot easily address. Crossover operators usually work best when individuals differ notably. However, CGP individuals that differ in genotype may have the same phenotype with a reasonable probability. This property can render crossover operators useless if individuals focus on one region of genotype space and thus represent the same point in phenotype space. Mutation operators usually modify genotypes slightly by altering some genes. For real-valued CGP, mutation operators must perform larger modifications to genotypes. Otherwise changes in genotype will most likely not reflect to phenotype. Thus, the mutation rate may have to be increased or mutation operators must alter genes relevantly, which may slow convergence down when individuals are changed too much.

3.2 Introducing the forking operator

All these considerations lead to the development of a new genetic operator that we call *forking*. As base for forking, we use an idea similar to the approach published by Tu and Lu [18]. This idea states that an individual should not only represent a point but may also span a region in genotype space as a multivariate Gaussian distribution. For this distribution, the individual is the mean whereas specified standard deviations for each dimension define the region's size. In that way, new individuals can be created by drawing a sample from this distribution. In contrast to the work done by Tu and Lu [18], we apply this idea to GP. Furthermore, our algorithm decides dynamically whether it interprets an individual as a point or as a distribution.

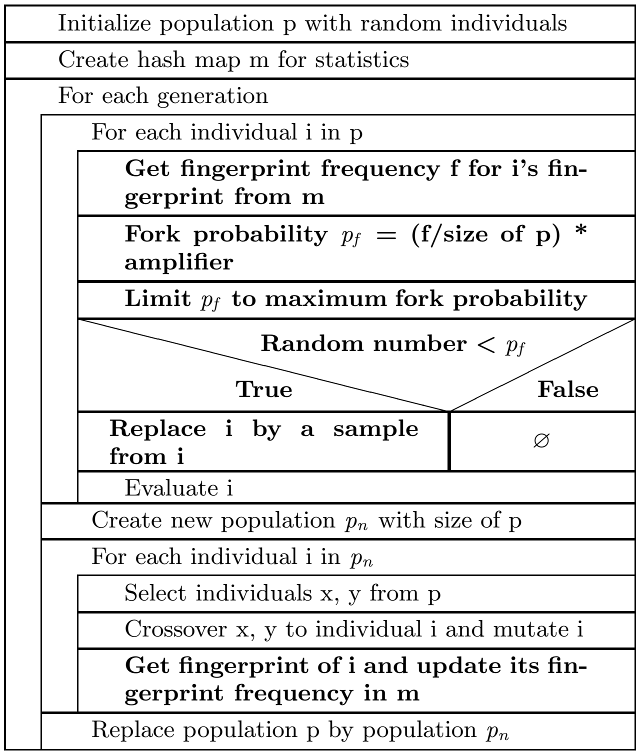


```
Initialize population p with random individuals
For each generation
    For each individual i in p
        Sample individuals s,t,u,v from i
        Evaluate s,t,u,v,i and replace i by best individual
        Update standard deviations of i
    Create new population $p_n$
    For each individual i in $p_n$
        Select individuals x, y from p
        Crossover x, y to individual i and mutate i
    Replace population p by population $p_n$
```

Figure 4: Structure chart of augmented algorithm

In our first implementation, we augmented real-valued CGP presented by Clegg et al.[2] with the stochastic algorithm described by Tu and Lu [18]. In figure 4 we show the augmented algorithm in which differences to the original algorithm are marked in bold. Since genotypes of real-valued CGP individuals consist only of floating-point numbers, the stochastic algorithm of Tu and Lu is applied without any modifications. As Tu and Lu described, we sample four individuals s, t, u and v in the distribution defined by each individual i and adapt the standard deviations of i as proposed. We compare this algorithm with a real-valued CGP algorithm similar to the one from Clegg et al., but grant the augmented algorithm to have only one fifth of the population size of the real-valued CGP algorithm. This adaptation is necessary for a fair comparison, because the augmented algorithm evaluates four additional individuals per each original individual so that its population size is virtually increased by factor five. In our experiments, real-valued CGP performs much better because it utilizes its larger real population size for covering more regions of genotype space in parallel. Even after reducing population size for the real-valued CGP algorithm and keeping the augmented algorithm untouched, real-valued CGP converges faster. Furthermore, we observed that when individuals are spread in genotype space, it is more efficient to interpret them as points. However, if they focus on few regions in genotype space, it is better to interpret them as distributions, because the sampling process leads to less redundant phenotypes.

Based on our first experiments, we conclude that it is better to combine both algorithms in a different way. In figure 5, we show real-valued CGP extended with the forking operator that realizes a dynamic combination of both algorithms. Again, modifications to real-valued CGP are marked in bold. Depending on current state of evolution, the forking operator decides for each individual whether it is interpreted as a point or as a distribution in genotype space. For this decision, forking uses population statistics. The central elements of these statistics are *fingerprints* stored in a hash map. As fingerprints, we use textual representations of phenotypes, e.g. a mathematical term like $x_0 + x_1 * x_0$, where x_0 and x_1 are inputs of the encoded program. The hash map stores the frequency of every fingerprint in the current population. Based on this frequency, the operator calculates a fork probability for an individual. The *amplifier* is a parameter and controls how a relative fingerprint frequency should be transformed into a fork probability. If the fork probability exceeds an adjustable *probability limit*, the fork probability will be set to this limit. Afterwards, the operator uses a uniform distribution to sample a random number and if this number is below the fork probability, the current individual will be interpreted as distribution. In this case, a new individual is sampled from current individual's distribution and it replaces the current individual. The next step is the evaluation of the individual. After all individuals have been evaluated, the population for the next generation is created. During this step, the hash map is updated with new fingerprint frequencies.



```
Initialize population p with random individuals
Create hash map m for statistics
For each generation
    For each individual i in p
        Get fingerprint frequency f for i's fingerprint from m
        Fork probability $p_f$ = (f/size of p) * amplifier
        Limit $p_f$ to maximum fork probability
        Random number < $p_f$
            True:  Replace i by a sample from i
            False: ∅
        Evaluate i
    Create new population $p_n$ with size of p
    For each individual i in $p_n$
        Select individuals x, y from p
        Crossover x, y to individual i and mutate i
        Get fingerprint of i and update its fingerprint frequency in m
    Replace population p by population $p_n$
```

Figure 5: Structure chart of extended algorithm

We realize sampling of new individuals from existing ones in a different way than Tu and Lu [18]. Similar to their work, we use the genotype of an individual as the mean of a multivariate Gaussian distribution. However, we do not use a separate standard deviation for each dimension. Instead, we perform sampling as shown in equation 1 and 2

$$G_{n,i} \sim \mathcal{N}\left(G_i, \sigma^2_{input}\right), \forall G_i \in \textit{input genes} \tag{1}$$

$$G_{n,f} \sim \mathcal{N}\left(G_f, \sigma^2_{function}\right), \forall G_f \in \textit{function genes} \tag{2}$$

, where $G_{n,i}$ represents a gene for an input, which is sampled from the Gaussian distribution defined by the current gene G_i as mean and σ_{input} as standard deviation for inputs. $G_{n,f}$ is a gene for a node function, which is sampled from the Gaussian distribution defined by the current gene G_f as mean and $\sigma_{function}$ as standard deviation for functions. By sampling in all genes G_i and G_f of an existing individual, the genotype of a new individual is created. Additionally to *amplifier* and *probability limit*, input deviation σ_{input} as well as function deviation $\sigma_{function}$ are parameters of the forking operator. In contrast to Tu and Lu, our two standard deviations are neither part of the genotype nor altered during evolution. Instead, standard deviations have to be defined carefully. As our example in figure 3 outlines, phenotype space is much smaller than genotype space. Since forking samples in genotype space, standard deviations have to be large enough so that genotypes may be created that map on different phenotypes. Setting standard deviations too large may lead to completely randomly initialized individuals so that any similarities to original individuals get lost. The choice of standard deviations depends also on program input and node functions count so that for many inputs and node functions, standard deviations should be smaller.

Population statistics are the key to decide dynamically between the two interpretations of individuals. When the current individual's phenotype and thus fingerprint is rare in the population, the forking operator likely interprets this individual as a point. This is usually the case in the beginning of evolution so that the algorithm behaves like real-valued CGP proposed by Clegg et al. [2]. With ongoing evolution, individuals focus on fewer regions so that the current individual's phenotype may be shared by other individuals as well. In this case, its fingerprint frequency increases and the forking operator will more likely interpret the current individual as a distribution. We also experimented with updating statistics, when a sampled individual replaces the original, but we did not see any significant differences in our experiments. The *amplifier* and *probability limit* influence the decision between both interpretations of an individual. It depends on the problem whether these parameters should be raised to increase likelihood of forking.

In addition to crossover and mutation, the forking operator is another genetic operator that modifies the genotype of individuals. Whereas mutation operators alter individuals due to a specified mutation rate, the forking operator always considers population statistics. These statistics base on phenotype and not on genotype, so that forking pays attention to the robust genotype-phenotype-mapping. Thus, forking is a dynamic genetic operator as it only reacts when phenotypes of individuals get too homogeneous. Furthermore, the forking operator performs a full sampling in all dimensions of the distribution and thus, in all genes. Although this sampling likely modifies all genes, it does not guarantee that a sampled individual will have another phenotype than distribution's mean. Instead, the probability of changes in individual's phenotype depends mostly on the standard deviations for node inputs and functions.

4. EVALUATION

In this section, we compare the efficiency of real-valued CGP and the algorithm using the forking operator. After describing our methodology, we present and discuss results for four symbolic regression experiments. Furthermore, we give hints for configuring the parameters of the forking operator and discuss the usage with other GP representations.

4.1 Methodology

For evaluating efficiency, we apply the real-valued CGP algorithm and the algorithm using the forking operator on the following symbolic regression problems.

$$\begin{aligned} f_1(x) &= x^6 - 2 * x^4 + x^2 \\ f_2(x) &= x^5 - 2 * x^3 + x \\ f_3(x,y) &= (x^2 * y^2)/(x+y) \\ f_4(x,y) &= x^5/y^3 \end{aligned}$$

Function f_1 and f_2 are also used by Clegg et al. [2]. We use the other functions to consider functions with multiple inputs as well as functions that also include the save division operator. As training data, we sample 50 random data points from the interval $[-1, 1]$. Like Clegg et al., we use the sum of absolute differences between expected and calculated outputs as cost function and require it to be smaller than 0.01. Since the implementations of real-valued CGP differ the results presented in this paper are not directly comparable to results published by Clegg et al.. Furthermore, we do not use variable crossover rates as proposed by Clegg et al., because their values were problem specific.

In our experiments, we use the same configuration for all algorithms, which is also similar to the configuration used by Clegg et al.. In table 1, we show this configuration.

Table 1: Basic configuration of all algorithms

Property	Value
Maximum node count	10
Function lookup table	+ (0), - (1), * (2), / (3)
Population size	50
Maximum Generations	20,000
Crossover operator	Weighted average
Crossover rate	0.75, not adaptive
Mutation operator	Reset gene to value $\in$ [0,1]
Mutation rate	0.20, not adaptive
Tournament selection size	20
Elitism size	2

The optimal configuration of parameters of the forking operator, *amplifier*, *probability limit*, *input deviation* and *function deviation*, depends on the problem to be solved so that we list it for each experiment separately. We also performed a meta-evolution on all four problems. This meta-evolution found a general set of parameters, which minimizes the number of generations to converge on all four problems. In table 2, we show this general set.

Table 2: General set of forking parameters

Parameter	Value
Amplifier	5
Probability limit	1.0
Input deviation	0.8
Function deviation	0.0

As measures for efficiency, we use the number of processed generations and also the required computation time until convergence. For every problem, we average these efficiency measures over 1,000 runs, which are performed with different seeds for the pseudo-random number generator. For each measure, e.g. 140 ± 316, the first number denotes the average value and the second the standard deviation. Our evaluation guarantees that all algorithms work on the same seeds. Furthermore, all algorithms process exactly the same input data. The algorithms are implemented in Java 6 (64-Bit) and not optimized for speed. They were run on a laptop equipped with an Intel Core i5-2520M CPU clocked at 2.50 GHz, 4 GB RAM and Microsoft Windows 7 Enterprise x64.

4.2 Results

Table 3: Efficiency for problem $f_1(x) = x^6 - 2 * x^4 + x^2$

Algorithm	Avg. number of generations	Avg. computation time in ms
Real-valued CGP	140 ± 316	837 ± 2,186
Best general	61 ± 91	330 ± 542
Best specific	60 ± 64	330 ± 391

In table 3, we show results for the first problem. *Real-valued CGP* is the equivalent to the algorithm proposed by Clegg et al.[2] and *Best general* represents the general set of forking parameters found by the meta-evolution. *Best specific* represents the optimal configuration of the forking operator for this problem and uses the following parameter values: amplifier = 5, probability limit = 1.0, input deviation = 0.4, function deviation = 0.1. Compared with real-valued CGP, this optimal configuration needs 57% fewer generations and 61% less computation time per run to converge on average. The general set of parameters is negligibly slower as it needs 56% fewer generations and also 61% less computation time per run.

In figure 6 and 7, we show plots summarizing convergence behaviors for the first problem. Figure 6 presents the average fitness of all 1,000 runs over the first 500 generations. Figure 7 shows the number of generations to converge for the sorted, slowest 500 runs. As visible, the forking operator enables notably faster convergence especially due to avoiding very slow runs. For the remaining problems, we will use the plot style as shown in figure 7. One main reason for this decision is that some of the presented problems use high polynomials, which lead to very high fitness values. Due to space limitations, it is not possible to plot these graphs as a whole in a good readable fashion. Furthermore, we are interested in finding perfect solutions so that the detailed development of convergence is not as important as the final number of generations to converge.

As visible in the tables 4-6 and their corresponding figures 8-10, the forking operator converges much faster for the other three problems as well. The configurations for the best specific configuration differ notably for amplifier, input and function deviation. Only the forking probability is always the same with 0.9 or 1.0.

Table 4: Efficiency for $f_2(x) = x^5 - 2 * x^3 + x$

Algorithm	Avg. number of generations	Avg. computation time in ms
Real-valued CGP	900 ± 1,882	5,720 ± 13,280
Best general	248 ± 294	1,431 ± 1,765
Best specific	244 ± 220	1,354 ± 1,231

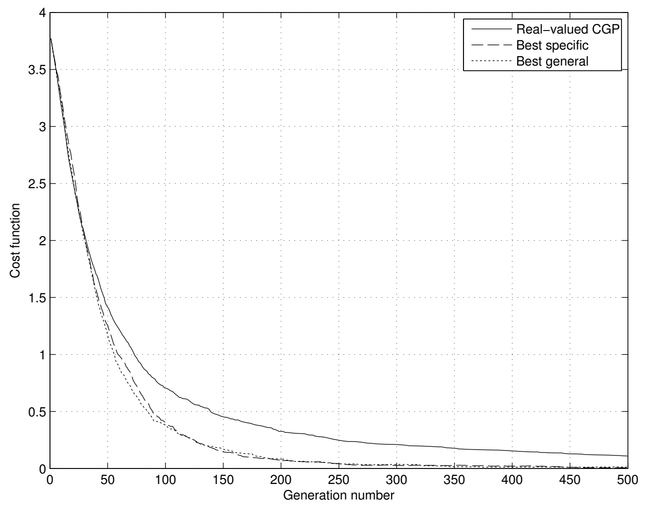


Figure 6: Average convergence for problem $f_1(x) = x^6 - 2 * x^4 + x^2$ covering multiple algorithms

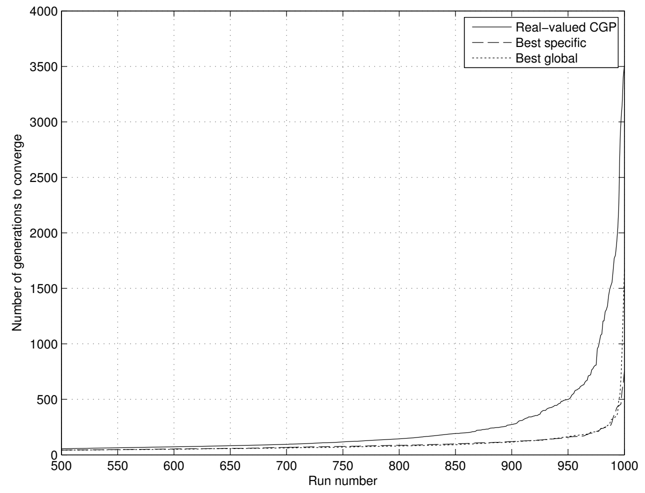


Figure 7: The number of generations to converge over the slowest 500 runs for $f_1(x) = x^6 - 2 * x^4 + x^2$

4.3 Discussion

Efficiency evaluations for the presented problems show that forking can accelerate convergence significantly. Although best specific parameter sets and best general set partly differ, their difference in efficiency is negligible. In comparison to the real-valued CGP algorithm, the forking operator configured with the best general parameter set reduces the number of necessary generations by 54% on average. Computation time per run is reduced by 61% on average. The reason for the difference between these speedups is that often not all individuals of the last generation are evaluated. Furthermore, the similar speedups for generation count and computation time in each experiment demonstrate that the additional operations of the forking operator do not affect computation time notably.

For number of generations to converge but also computation time, the standard deviations are significantly reduced in comparison to real-valued CGP. This fact is also underlined by the presented figures. As we have shown in the figures, the forking operator based algorithms do not accelerate all problem runs, but avoid very slow runs. Furthermore, all

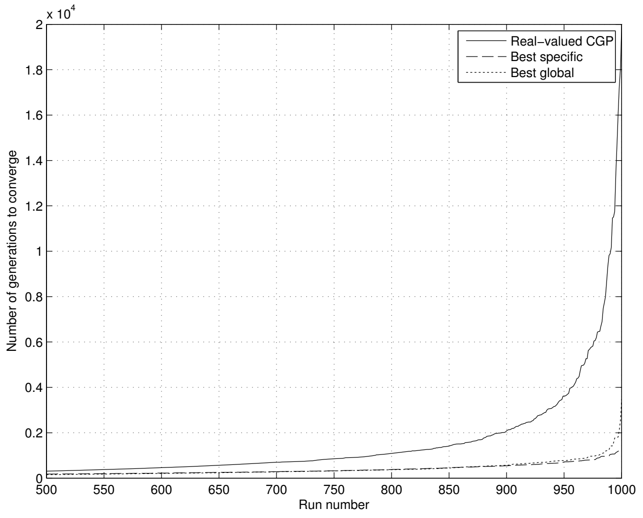


Figure 8: The number of generations to converge over the slowest 500 runs for $f_2(x) = x^5 - 2 * x^3 + x$

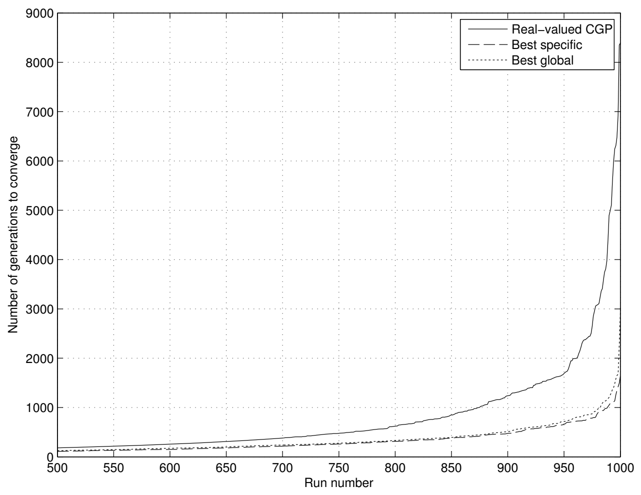


Figure 9: The number of generations to converge over the slowest 500 runs for $f_3(x, y) = (x^2 * y^2)/(x + y)$

Table 5: Efficiency for $f_3(x, y) = (x^2 * y^2)/(x + y)$

Algorithm	Avg. number of generations	Avg. computation time in ms
Real-valued CGP	466 ± 863	2,251 ± 4,366
Best general	216 ± 264	881 ± 1,131
Best specific	194 ± 224	741 ± 894

Table 6: Efficiency for $f_4(x, y) = x^5/y^3$

Algorithm	Avg. number of generations	Avg. computation time in ms
Real-valued CGP	411 ± 1,187	2,352 ± 7,563
Best general	260 ± 583	1,181 ± 2,705
Best specific	240 ± 471	1,128 ± 2,303

runs of the algorithm with the forking operator found the correct solution, so that premature convergence did not occur. However, it cannot be generalized that forking will not lead to premature convergence in some cases.

Although the forking operator demonstrated its ability to accelerate convergence, the introduction of new parameters without supplying a well-founded description of how to initialize them is a disadvantage. If we take the best general set of parameters for guidance, it seems that probability limit should always be set to the maximum value of 1.0 so that this parameter can be removed. Setting amplifier to 5 seems to be a good general choice, so that input and function deviations remain as most difficult parameters. According to their values in the best general set, the main problem is rather the correct connection between nodes, but not which function is associated to each node. However, we only used four possible node functions and ten nodes per CGP program. For other problems, which use more possible node functions, importance of the function deviation may increase. One solution for this problem could be the increase of available nodes per CGP program. As Miller and Smith outlined, increasing node count increases efficiency, too [13]. As a side effect, we assume that this change will likely cause every possible node function to be present in the graph. Thus, the problem could be reduced to finding the correct connection between nodes so that only the input deviation is of importance. However, both deviations should not be set close to 1.0 at the same time, because this may lead to a completely randomly initialized individual.

For using forking with other GP representations, effective methods for creating fingerprints and sampling new individuals from existing individuals have to be found. Basically, fingerprinting offers a way to detect *behavioral equivalence* in the population. Since every GP representation should offer a textual representation of a program, this is the simplest form to realize fingerprinting. However, especially for large programs, a fingerprint mechanism that finds semantical instead of only textual equivalence may be beneficial. Sampling new individuals can be realized easily when the genotype space is continuous. However, GP representations like Koza's parse trees have a discrete genotype space so that sampling using a Gaussian distribution requires a mapping from real-valued space to discrete genotype space. This mapping has to be considered, when standard deviations for each distribution are defined. Maybe, CGP's mapping from real-valued genotype to its discrete phenotype space can be modified to work with other GP representations as well.

5. CONCLUSIONS AND FUTURE WORK

Genetic programming is a powerful paradigm for solving difficult problems, but its algorithms often demand high computational effort to converge. In this paper, we introduced the forking operator, which samples new individuals from existing ones when populations get too homogeneous. Experimental results on four symbolic regression problems showed that forking can improve efficiency by over 50% on average. Despite evaluating the operator for other problems the choice of parameters for a specific problem in advance remains as an important research question. Furthermore, developing an improved fingerprint mechanism may be beneficial, so that syntactically different but semantically equivalent phenotypes could be recognized. Besides the forking operator, we think combining population statistics upon fingerprints with other genetic operators is an interesting research topic. For instance, the variable crossover rate pro-

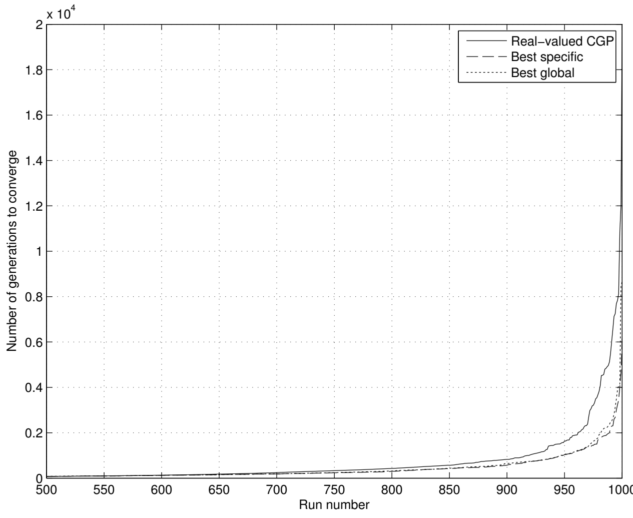


Figure 10: The number of generations to converge over the slowest 500 runs for $f_4(x,y) = x^5/y^3$

posed by Clegg et al. [2] may be realized with these statistics in a problem-independent way.

Another interesting research question could be extending forking to evolutionary algorithms. In contrast to GP applications, phenotype spaces for evolutionary algorithms are often continuous and not discrete. Thus, identifying individuals with the same phenotype is not as easy as for discrete spaces. A possible solution may be a metric that describes similarity of individuals in a problem-independent fashion.

For future research, our main focus will rely on further accelerating GP algorithms. Additionally, we will use these algorithms to realize a prediction model for car accident severity so that new intelligent occupant safety functions could be developed. By improving convergence behavior, we believe that we find better solutions for this particular problem as larger solution spaces could be covered.

References

[1] Wolfgang Banzhaf, Frank D Francone, and Peter Nordin. The Effect of Extensive Use of the Mutation Operator on Generalization in Genetic Programming Using Sparse Data Sets. In *Proceedings of the 4th International Conference on Parallel Problem Solving from Nature*, pages 300–309. Springer-Verlag, 1996.

[2] Janet Clegg, James Alfred Walker, and Julian Frances Miller. A New Crossover Technique for Cartesian Genetic Programming. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1580–1587. ACM, 2007.

[3] Lawrence Davis et al. *Handbook of Genetic Algorithms*, volume 115. Van Nostrand Reinhold New York, 1991.

[4] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* University of Michigan Press, April 1975.

[5] Gregory S Hornby. ALPS: The Age-Layered Population Structure for Reducing the Problem of Premature Convergence. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 815–822. ACM, 2006.

[6] IEEE Task P754. IEEE Standard for Floating-Point Arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society, August 2008.

[7] ISO. ISO C Standard 1999. Technical report, ISO, 1999. ISO/IEC 9899:1999 draft.

[8] John R. Koza. *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems.* Stanford University, Department of Computer Science, 1990.

[9] W. B. Langdon. A Many Threaded CUDA Interpreter for Genetic Programming. In *Proceedings of the 13th European Conference on Genetic Programming*, volume 6021 of *LNCS*, pages 146–158. Springer, 2010.

[10] Joel Lehman and Kenneth O. Stanley. Efficiently Evolving Programs Through the Search for Novelty. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, GECCO '10, pages 837–844. ACM, 2010.

[11] Julian F. Miller. An Empirical Study of the Efficiency of Learning Boolean Functions using a Cartesian Genetic Programming Approach. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1135–1142. Morgan Kaufmann, 1999.

[12] Julian F. Miller, editor. *Cartesian Genetic Programming.* Springer Berlin Heidelberg, 1st edition. edition, September 2011.

[13] Julian F. Miller and Stephen L. Smith. Redundancy and Computational Efficiency in Cartesian Genetic Programming. *Transactions on Evolutionary Computation*, 10(2):167–174, September 2006.

[14] Julian F. Miller and Peter Thomson. Cartesian Genetic Programming. In *Proceedings of the Third European Conference on Genetic Programming*, volume 1802, pages 121–132. Springer-Verlag, 2000.

[15] Ingo Rechenberg. *Evolutionsstrategie : Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Number 15 in Problemata. Frommann-Holzboog, Stuttgart-Bad Cannstatt, 1973.

[16] Michael Schmidt and Hod Lipson. Distilling Free-Form Natural Laws from Experimental Data. *Science*, 324 (5923):81–85, 2009.

[17] Karel Slanỳ. Comparison of CGP and Age-Layered CGP Performance in Image Operator Evolution. *Genetic Programming*, pages 351–361, 2009.

[18] Zhenguo Tu and Yong Lu. A Robust Stochastic Genetic Algorithm (StGA) for Global Numerical Optimization. *Transactions on Evolutionary Computation*, 8(5):456–470, October 2004.

[19] Zhenguo Tu and Yong Lu. Corrections to "A Robust Stochastic Genetic Algorithm (StGA) for Global Numerical Optimization". *Transactions on Evolutionary Computation*, 12(6):781–781, December 2008.