An Explicitly Neutral Mutation Operator in Cartesian Genetic Programming

Eduardo Pedroni

Dept. of Electronics, University of York, York, UK

Abstract. In genetic programming, neutrality occurs when genotypes of the same fitness contain different inactive node configurations. This has been shown to be very beneficial in many areas, such as digital circuit design with Cartesian Genetic Programming. In this paper we propose an explicitly neutral mutation operator which allows the user to directly control the mutation rate of inactive genes. We compare the proposed mutation operator with the standard Cartesian Genetic Programming implementation in three different problems using a range of mutation rates and topologies. The proposed mutation is found to be less sensitive to mutation rate and to reach perfect solutions more consistently than the standard mutation operator.

Keywords: Cartesian Genetic Programming, mutation operator, neutrality.

1 Introduction

In genetic programming, programs are evolved by means of genetic mutation. A program is defined in terms of interconnected primitive function nodes, and a mutation operator changes both the function type and connectivity of each node. Each set of nodes, or genotype, has its fitness calculated with respect to the specified perfect output. Selection is performed by discarding or carrying forward genotypes based on their fitness level so that the population becomes better adapted to the problem as generations go by.

The concept of neutrality stems from the fact that genotypes might contain inactive genes, in other words, genes that do not actively manifest themselves as phenotypical traits. Mutations of inactive genes are referred to as neutral mutations, since their effect is not directly visible. It is theorized that while neutral mutations may not actively contribute to an individual's fitness, future non-neutral mutations might result in beneficial phenotypical traits as a consequence of past neutral mutations [7]. In a sense, the inactive portion of a genotype represents that genotype's potential for change, implying that neutral mutations are in fact a way of exploring the search space. This neutral exploration of the search space is termed neutral drift [3, Sec. 2.7].

Neutrality and neutral mutations have been categorised into two groups: explicit and implicit [8]. Explicitly neutral mutations are those applied to inactive genes. They are said to be explicit as they operate directly on the neutral portion

of the genotype. Implicit neutrality, on the other hand, refers to mutations which affect active genes and therefore lead to a change in the phenotype, all without causing a change in fitness.

In this paper, a new mutation operator will be proposed as an alternative to the standard Cartesian Genetic Programming operator. The proposed mutation technique aims at harnessing the advantages presented by neutrality to improve on the performance offered by the current operator.

2 Cartesian Genetic Programming

Cartesian Genetic Programming (CGP) is a form of genetic programming first proposed by Miller in 1999 [2]. It developed from a GP implementation used to design digital circuits, though it is quite a generic technique and it can be used to solve a variety of problems. It is referred to as Cartesian due to the way in which it treats its nodes; rather than allowing nodes to be created, a fixed number of nodes divided into rows and columns is defined by the user prior to running the program. A CGP genotype, therefore, is simply a fixed-length string of integers which is mapped when fitness is to be calculated. The genotype can be broken down into each of its component nodes; each node, in turn, consists of a function gene and n connection genes for an n-arity function. The output connections are appended to the end of the genotype as output genes, and can be mutated like any other gene. Fig. 1 illustrates this concept.

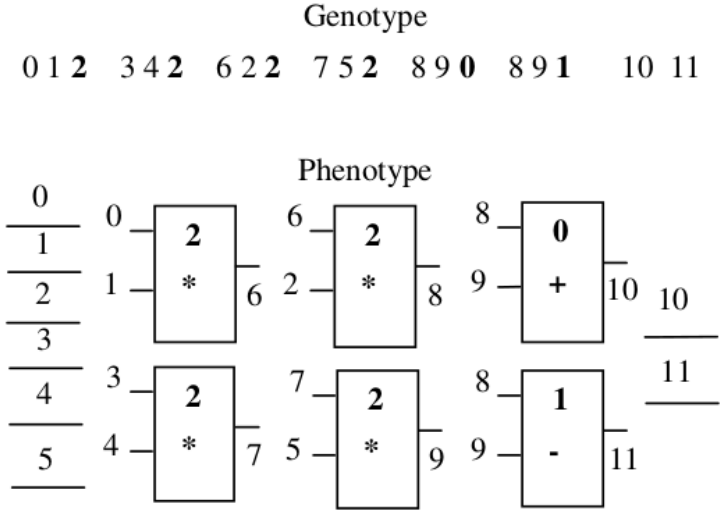

Fig. 1. Example of a CGP genotype and its associated phenotype, reproduced from [2]. Genes shown in bold font are functions genes, the others are connection genes. The numbering illustrates how the genotype is mapped into a circuit for evaluation.

Crossover, while popular with other techniques, has remained relatively unpopular in the CGP community; most applications focus purely on mutation operations [6, Sec. 7.2.3]. The evolutionary algorithm typically used is $1 + \lambda$, where λ is defined by the population size. Elitism is also employed to automatically promote the fittest genotype of a population, and a parameter called make_efficient allows genotypes to be optimized and reduced in size; this is done by adding the number of inactive nodes to the fitness value, thus encouraging smaller genotypes to be evolved.

CGP offers several advantages over other genetic programming techniques. While most GP methods suffer from bloat, CGP is inherently bloat-free as it specifies a fixed number of nodes and does now allow nodes to be created. Additionally, due to the fixed genotype size, CGP chromosomes have a tendency to display significant redundancy; out of 4000 nodes, for instance, it was found that only about 5% were active nodes [4]. Because of this redundancy, CGP is very suited for experimenting with neutrality. In fact, experiments have shown that neutrality is very important in obtaining good results [4,5,7,8]; Fig. 2 illustrates the effect of neutral drift. Because of this, the evolutionary algorithm in CGP specifically implements drift by selecting an offspring over the parent when their fitness is identical. Since most genes are inactive, with low mutation rates it is likely that only inactive genes are mutated, and therefore offspring should be selected over parents in such cases, if neutral drift is desired.

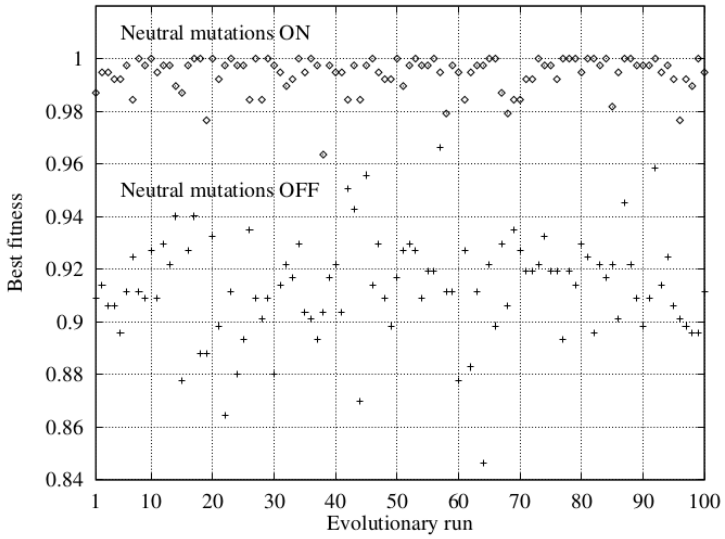

Fig. 2. Comparison of best fitness with and without neutral mutations for solutions of a 3-bit digital parallel multiplier, reproduced from [3, Sec. 2.7]. Neutral drift produces consistently superior results across all runs. In comparison, no perfect fitness genotypes were achieved with neutral drift suppressed.

In CGP, mutation operators have remained relatively untouched. The standard CGP implementation uses a point mutation operator, which works by randomly mutating a user-defined number of genes. The program calculates the number of genes to be mutated based on the specified mutation rate and selects genes at random until enough genes have been mutated. This is a very basic operation which does not offer much control over how mutation occurs; it mutates active and inactive genes alike. However, given the way neutrality works, a mutation operation which discriminates between active and inactive genes might offer advantages over point mutation.

3 Related Work

Goldman and Punch recently explored an alternative mutation operator, primarily as a means of reducing wasted evaluations [1]. This proposed operator, named *Single*, works by mutating genes at random until an active gene is mutated. Once that condition is fulfilled, the mutation process ends. It does not require a user-specified mutation rate, as the percentage of the genotype that is mutated varies. *Single* is said to have an effective mutation rate of "$\frac{n-a}{a+1} + 1$ expected mutations, where a is the number of active genes and n is the total number of genes." This is an interesting property, as it means that the number of mutated genes changes according to the number of active genes present. More specifically, less mutations occur on average as the number of active genes increases.

Single does not negate neutral drift as it mutates non-coding genes, but by mutating only one active gene per offspring, phenotype changes tend to be more incremental. Goldman and Punch used *Single* in benchmark problems of varying difficulty: parity, multiply, binary encode and binary decode. They concluded that *Single* was only appropriate "in situations where the mutation rate cannot be optimized", though a trend of better performance on harder problems was observed.

Miller and Yu also investigated the importance of neutrality by devising a way to measure it during the evolution process [8]. The hamming distance between genotypes is calculated, and used to completely allow or forbid neutral mutations to take place, or specify the degree of neutral drift allowed. This technique was used to investigate neutrality evolving the 3-bit even parity circuit with hamming distances of 0, 50, 150, 200, 250, 300, where 0 means neutrality is not permitted and 300 means any amount of drift may occur. It was found that hamming distances between 250 and 300 outperform lower values for any mutation rate, meaning drift effectively allows evolution to find better solutions.

4 Neutral Mutation Operator

We propose a mutation operator inspired by *Single* and the experiments performed in [8], which will be referred to as *Inactive*. This proposed operator takes the user-defined mutation rate to mutate only inactive genes; in other words, for

a mutation rate R, $R\%$ of all inactive genes will be randomly mutated. In addition, this operator mutates a single random active gene once the user-defined percentage of inactive genes has been mutated, in order to ensure phenotypical changes.

The idea behind *Inactive* is to provide some measure of control over neutral drift, as it was shown in previous experiments to have a positive effect on the success rate (see Section 2). A special case of *Inactive*, where the mutation rate is set to zero, will also be investigated. This special case exclusively mutates a single random active gene and zero inactive genes, and will be referred to as *Exclusive*.

5 Experimental Setup

The aim of the experiment is to compare the effectiveness of *Inactive* and *Exclusive* with the standard CGP mutation operation (this will henceforth be referred to as *Regular*). We make this comparison in two different digital circuit problems: 2-bit adder and 3-bit multiplier. The following mutation rates were investigated: 0.5, 1, 5, 10, 30 and 80 percent. A mutation rate of 2% was used in addition to these for the multiplier circuit for a more accurate comparison, since *Regular* is expected to work best with lower mutation rates. Note that these mutation rates do not apply to *Exclusive*, which exclusively mutates a single active gene regardless.

These mutation rate values were chosen to provide an overview of how *Inactive* performs in comparison to *Regular* where *Regular* is expected to excel (lower mutation rates [4]) but also explore higher, less commonly used mutation rates. Since *Inactive* effectively controls the amount of neutral drift, a high mutation rate does not necessarily lead to excessive movement around the search space as it might with the point mutation; instead, it allows for bigger jumps to potentially occur without inhibiting the step increases in fitness which allow the algorithm to climb to local maxima.

Three different topologies were used: 75, 150 and 300 columns, all with a single row and levels back set to the number of columns. We chose to vary the topology as well as the mutation rate due to the connection between the two; more genes are mutated if the genotype is longer. In a situation where the genotype is short, few genes are inactive and the mutation rate is low, *Inactive* might fail to mutate any genes at all since the total number of genes to be mutated might for instance be rounded to zero. We include larger topologies in the experiments to account for this risk.

The primitive gates allowed were: AND, OR with one inverted input and XOR for the multiplication problem; AND and XOR for the adder problem. Each circuit was allowed 1,000,000 generations with a population of 5 and an evolutionary algorithm $1 + 4$. In addition, 100 runs were performed for each set of parameters, in order to ensure the statistical validity of the results. In order to save time, runs were set to prematurely end whenever a perfect solution was found.

6 Results

Fig. 3 and Table 1 contain data regarding the experiments done to evolve the 2-bit adder. While Table 1 shows data for relevant experiments, Fig. 3 omits 80% mutation rate as *Regular* failed to find any successful genotypes with those settings. Figs. 4, 5 and 6 and Table 2 contain the relevant data gathered from the multiplier experiment. Mutation rates that do not apply to the graphs have once again been omitted, while Table 2 still contains the complete set of data.

As mentioned previously, one additional mutation rate was used for the multiplier. The original six mutation rates (0.5, 1, 5, 10, 30 and 80) were found to provide a good overview for the adder performance, but since *Regular* is found to perform drastically better with lower mutation rates, we saw fit to run one additional experiment to provide better resolution in that range.

7 Discussion

		Mutation Rate					
Topology	Operator	0.5	1	5	10	30	80
75	Regular	100	100	100	100	23	0
	Inactive	69	100	100	100	100	100
	Exclusive			62			
150	Regular	100	100	100	100	13	0
	Inactive	100	100	100	100	100	100
	Exclusive			87			
300	Regular	100	100	100	99	6	0
	Inactive	100	100	100	100	100	100
	Exclusive			92			

Table 1. Table showing the success rate for every experiment done with the 2-bit adder. Note that *Exclusive* does not depend on mutation rate.

		Mutation Rate						
Topology	Operator	0.5	1	2	5	10	30	80
75	Regular	73	86	85	10	0	0	0
	Inactive	5	89	94	99	97	87	9
	Exclusive				5			
150	Regular	92	99	88	10	0	0	0
	Inactive	98	96	99	100	42	45	45
	Exclusive				36			
300	Regular	100	98	93	3	0	0	0
	Inactive	99	99	100	100	65	69	61
	Exclusive				63			

Table 2. Table showing the success rate for every experiment done with the 3-bit multiplier. Note that *Exclusive* does not depend on mutation rate.

7.1 2-bit Adder

The 2-bit adder is a relatively easy problem to solve with the gates given. We will look at success rate to compare the operators at higher mutation rates, and average number of generations taken to find perfect solutions in order to compare successful mutation rates. Table 1 outlines the success rates for each experiment.

In 1,000,000 generations, *Regular* was able to consistently achieve perfect solutions in all topologies with mutation rates as high as 10%. With mutation rates above 30%, however, the highest number of perfect solutions achieved was 23/100, with a topology of 75 and 30% mutation. In addition, no perfect solutions were found for any topology with 80% mutation rate. This falls in line with what was discussed earlier; when used with extremely high mutation rates, *Regular* randomly explores the search space rather than climbing a local maximum, making it statistically unlikely to reach a perfect solution within the given number of generations.

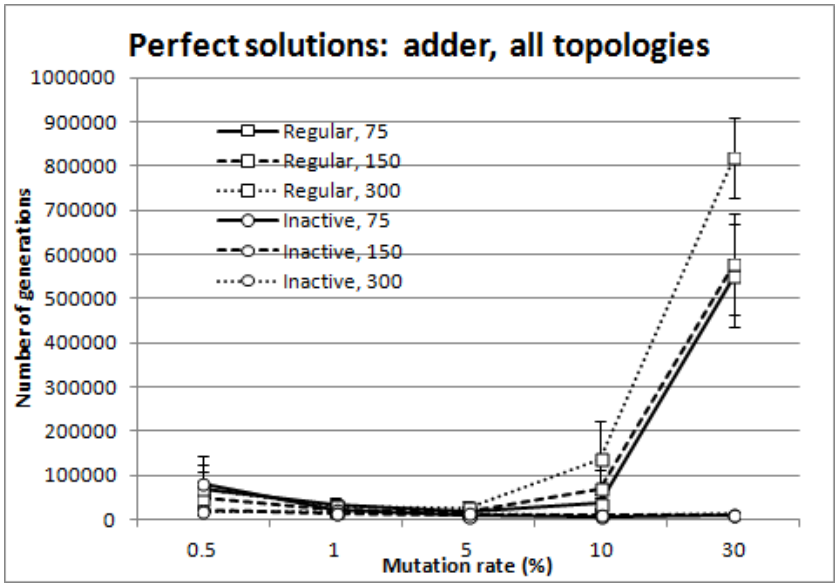

Fig. 3. Graph showing the number of generations required for each mutation operator to find a perfect 2-bit adder circuit with each mutation rate and topology. Note that the standard deviation for *Inactive* has been rendered as error bars, but is too small to be visible.

Conversely, *Inactive* found 100/100 perfect solutions with nearly all sets of parameters. In fact, as shown in Fig. 3, the number of generations taken to find a perfect solution does not significantly increase with mutation rate, as is the case with *Regular*. The low standard deviations in the *Inactive* results for higher

mutation rates actually suggest that it benefits from such parameters, rather than simply coping.

Exclusive displayed poor results in comparison with the other two operators. The highest success rate achieved was 92% with the largest topology, whereas only 62% of the runs with topology 75 ended with perfect scores. The number of generations taken to achieve perfect solutions displayed large standard deviations, often twice as high as the number of generations itself. This is most likely related to the lack of explicit drift. Since only active genes can be mutated, the operator fails to search a significant space.

Interestingly, *Inactive* performs anomalously bad with mutation rate 0.5% and topology 75, achieving only 69% success rate. In this case the number of generations taken for successful solutions is generally inconsistent and high on average. This is likely due to the size of the genotype not allowing significant neutrality, as discussed in Section 5. The fact that *Inactive* was able to perform well with 0.5% mutation rate given more nodes to work with further supports this hypothesis.

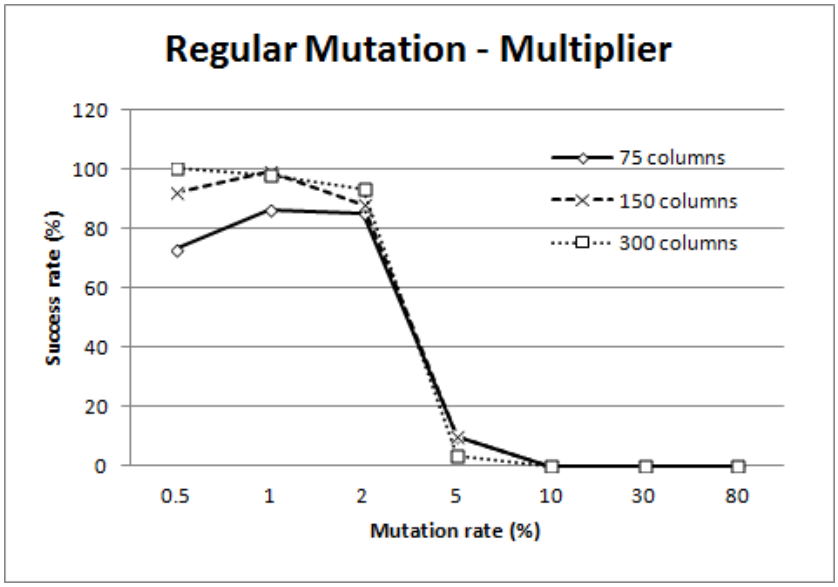

Fig. 4. This graph shows the success rate of *Regular* in finding perfect solution for the 3-bit multiplier with different topologies and mutation rates. Note that success rate here means number of perfect solutions out of 100 runs.

7.2 3-bit Multiplier

The 3-bit multiplier is a harder problem which generally requires more generations to solve than the 2-bit adder. To compare mutation operators with this

problem, we will look at the success rate for all mutation parameters, and average number of generations needed to achieve successful genotypes for the most efficient mutation rates. Table 2 outlines the success rate for each experiment.

Once again, *Regular* behaved as expected, producing its best results with lower mutation rates. It achieved 100% success rate with topology 300 and mutation rate 0.5%, and attained over 73% success for all topologies and mutation rates below 2%. As we can see in Fig. 4, very few perfect solutions were found with mutation rates higher than 5%; it is also apparent that a larger topology led to better results overall. The low success rates at high mutation settings are caused by too many active genes being mutated, leading the genotype to "leap" across the search space instead of climbing.

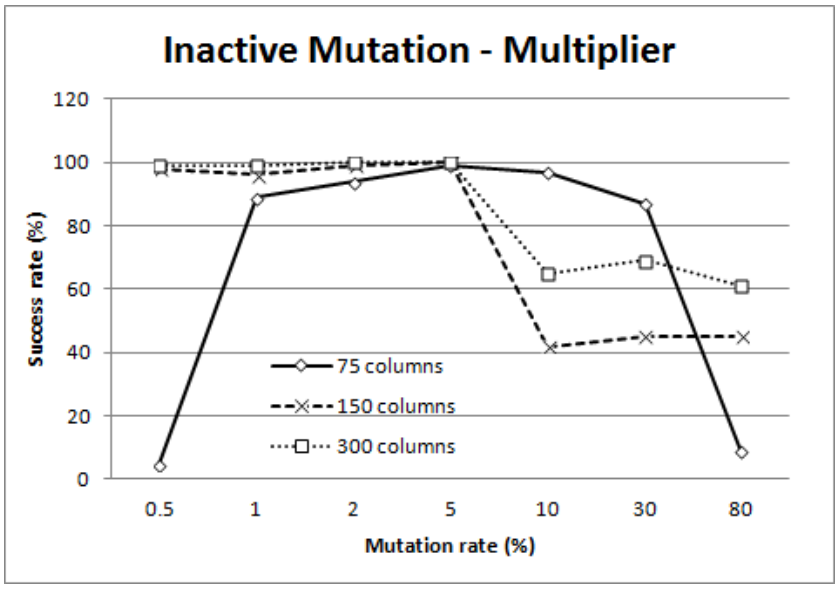

Fig. 5. This graph shows the success rate achieved by *Inactive* with each topology and mutation rate in trying to find a perfect 3-bit multiplier solution.

In comparison, *Inactive* has a higher overall success rate (see Fig. 5). It achieves comparable success rates at lower mutation rates and shows significant improvement over *Regular* for 5% mutation rate in particular. However, its success rate abruptly drops at 10% mutation, and yet it remains constant for further mutation rates. Moreover, this only happens with topologies 150 and 300; with topology 75, the cut-off only happens at a much higher mutation rate. While the cut-off at low mutation rates for topology 75 is expected (for the same reasons as mentioned in the previous section), the behaviour of *Inactive* in this problem with higher topologies and mutation rates is difficult to account for. One possible explanation is that a larger number of nodes is generally used for the multiplier

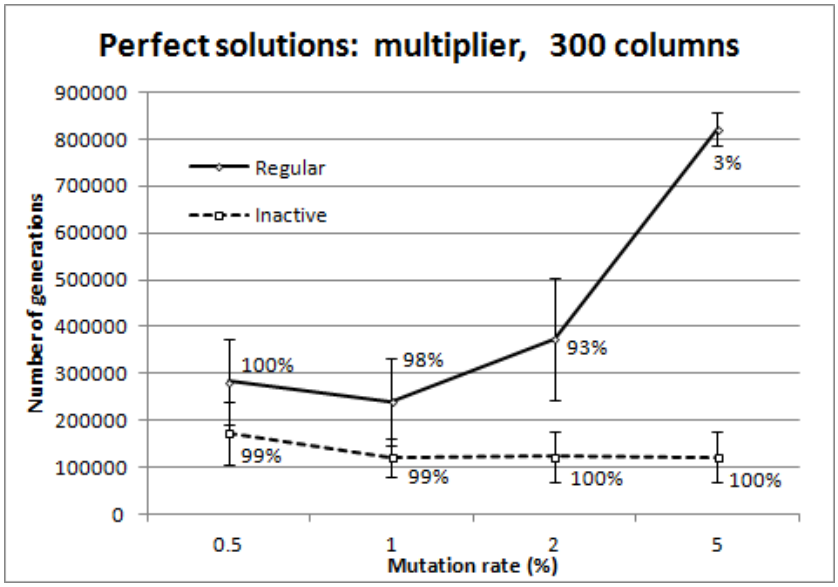

Fig. 6. This graph shows the average number of generations until a perfect solution is found *Regular* and *Inactive* with different mutation rates. Since both performed best with topology 300, topologies 75 and 150 are omitted for clarity.

circuit than for an adder; because of this, even higher topologies might need to be chosen to allow *Inactive* to realise its full drifting potential.

Inactive did once again outperform *Regular* in terms of number of generations needed to reach a perfect solution. As illustrated in Fig. 6, the two operations are largely equivalent until higher mutation rates are reached; above 5%, *Regular* has trouble climbing local maxima without leaping too far across the search space and therefore fails to find perfect solutions, whereas *Inactive* remains consistently quick at climbing up to perfect fitnesses. This suggests that enforcing neutral mutations does not prevent the genotype from stepping up the local maximum; this likely also relies on the single active gene being mutated at every generation.

Similarly to the adder circuit, *Exclusive* did not perform competitively in comparison with *Regular* and *Inactive*. Its success rate once again increased proportionally to the topology, however its highest value was only 63%. Once again, the number of generations taken to reach perfect solutions was very irregular, with high standard deviations preventing any statistically valid conclusions to be drawn. This is probably due to this operator's lack of explicitly neutral mutations.

8 Conclusion and Future Work

Neutrality is an important quality in genetic programming; neutral mutations have the ability to significantly improve the performance of a GP implemen-

tation, when integrated appropriately. Out of the two proposed mutation operators, *Exclusive* does not appear to be very promising. Its poor performance clearly reflects the lack of neutrality and provides further evidence to support neutral mutations. *Inactive*, on the other hand, performed at least as well as *Regular* where *Regular* excels. In addition, it outperformed *Regular* when applied with more unusual mutation rates by displaying both higher success rates and more consistent number of generations. In general, *Inactive* appears to be less sensitive to mutation rate; while this may not necessarily be a positive trait, it makes *Inactive* an interesting operator nonetheless, as it appears to explore the search space in a different way to *Regular*.

The efficacy of *Inactive* on different problems outside the realm of circuit design is worth investigating, as it may present an advantage in the way it performs its search, given different search space profiles. Miller and Smith suggest that the optimal topology for parity circuit design is 3000 [4]. It might therefore also be interesting to evolve circuits with topologies of such magnitude, where *Inactive* might deviate more decisively from *Regular*.

It is also worth mentioning that the sheer time taken for *Inactive* to perform mutations made it difficult to experiment with additional parameters. When mutating, *Inactive* must first acquire a list of active nodes, and then assert that each randomly picked gene is not active. This incurs relatively time-consuming computation, which, over many generations and runs, adds up to a significant time. If *Inactive* is to be investigated further, it would be ideal to find an alternative method to implement it, such as only allowing inactive genes to be picked, rather than picking completely at random and checking for activity individually.

References

1. B. W. Goldman and W. F. Punch, "Reducing wasted evaluations in cartesian genetic programming," 2013.
2. J. 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*, vol. 2, 1999, pp. 1135–1142.
3. J. F. Miller, Ed., *Cartesian Genetic Programming*, ser. Natural computing series. Springer, 2011.
4. J. F. Miller and S. L. Smith, "Redundancy and computational efficiency in cartesian genetic programming," *Evolutionary Computation, IEEE Transactions on*, vol. 10, no. 2, pp. 167–174, 2006.
5. J. F. Miller and P. Thomson, "Cartesian genetic programming," 2000.
6. R. Poli, W. B. Langdon, N. F. McPhee, and J. R. Koza, *A Field Guide to Genetic Programming*. Lulu, 2008.
7. V. K. Vassilev and J. F. Miller, "The advantages of landscape neutrality in digital circuit evolution," 2000.
8. T. Yu and J. Miller, "Neutrality and the evolvability of boolean function landscape," 2001.