Destructiveness of Evolutionary Operations in Linear and Cartesian Genetic Programming

Mark Kocherovsky[1◑], Marzieh Kianinejad[1◑], Elijah Smith[1◑], Illya Bakurov[1]

1 Computer Science and Engineering, Michigan State University, East Lansing, MI, USA

◑These authors contributed equally to this work.

Abstract

It is often reported that performing crossover in Cartesian Genetic Programming (CGP) tends to hinder rather than facilitate search. Despite the numerous attempts to making a working crossover operator, there is very few literature on why the traditional (one- and two-point) operators are ineffective in the first place. Linear Genetic Programming (LGP), on the other hand, requires crossover to search despite LGP and CGP's design similarities. We test various crossover methods with LGP and CGP and hypothesize that LGP's internal calculation registers and ability to change program size diminishes the likelyhood that high-fitness substructures are destroyed by crossover. We also test several selection and mutation operators to determine their role in CGP search, and find that crossover is by far the most likely point of failure.

Introduction

Genetic Programming was originally developed using tree structures to solve simple binary and symbolic regression problems [1, 2]. Linear and Cartesian Genetic Programming (LGP and CGP) were developed as an alternative to traditional Tree GP to address shortcomings such as bloat and usability [3–5]. In LGP, a program is represented as a sequence of instructions where instruction input and output are read from and written to a set of registers. In CGP, the program is designed as a Directed Acyclic Graph (DAG) where each node takes input directly from previous nodes. However, both are conceptually similar enough to be able to convert one into the other [6].

It is repeatedly observed that simple crossover methods — typically one-point — are actually detrimental to CGP search [7] but nothing similar been reported in LGP, where, in fact, crossover is *necessary*. This is strange since the two algorithms have such similar designs. The literature shows numerous attempts to create *new* crossover operators [7–12] but very few studies on why the traditional crossover methods are insufficient [13, 14].

In this work we try to answer the question of why one- and two-point crossover methods are harmful to CGP but helpful for LGP. We test different crossover methods — one- and two-point crossover for both CGP and LGP, subgraph crossover [10] for the former, and uniform crossover [15] the latter with seven symbolic regression problems, and hypothesize that there are two core reasons why crossover destroys CGP programs. First, the use of internal calculation registers allows high-fitness substructures to be "anchored", thus less likely to be destroyed because evolution is not directly altering connections between instructions like is done in CGP. Second, LGP (typically) allows programs to change in size, whereas CGP programs stay at a fixed size. Size-fixing likely does not prevent crossover from “amputating” high-fitness substructures since there is less flexibility in the choice of crossover point.

We are also curious to see if the selection or mutation method plays a role in this phenomenon. Specifically, we want to understand at which point different selection schemes and pressure can mitigate the destructive effects in CGP. In the case of selection, we compare the performance of CGP, CGP One-Point crossover, and CGP Two-Point crossover using three different selection operators. In the case of mutation, we compare our standard (basic) mutation method, adaptive mutation [16, 17], and swap mutation [18]. We find that alternating selection and mutation methods likely does not contribute to the CGP crossover phenomenon.

Previous Literature

When discussing CGP crossover, [7] is usually cited as a main report of the phenomenon. In the study, the authors designed a crossover operator using conversions to-and-from real-valued genes. The authors of [8] use elitism to preserve the best parents between generations to preserve high-fitness substructures. In 2018, Husa and Kalkreuth used substructure modularization [9], and in 2020, Kalkreuth used “Subgraph” crossover to ensure that the same nodes from each parent remain active [10]. Cai *et al.* and Cui *et al.* posit that the reason for the failure of traditional crossover is CGP reliance on positional dependency, i.e. that nodes are more likely to be active if they are closer to the input [13, 14], but Tree and Linear GP also have their own forms of positional dependency.

Overview of Concepts

LGP and CGP

An LGP model is based on of a set of registers R, where, in a basic univariate problem, there is one output node r_0, an input node r_1 (which is also read-only), a set of constants such as $[0, 9]$, and a set of intermediate calculation registers $r_i \in [r_2, r_n]$. The program itself is a set of instructions that contains a destination node $\{r_d \in R | d \neq 1\}$, an operator o drawn from an operator set, and (assuming binary operations) two operands $a_{i,1}$ and $a_{i,2}$ drawn from R.

Our LGP programs p in the population P takes an encoding of the form

$$p \in P = \begin{bmatrix} d_0 & o_0 & a_{0,1} & a_{0,2} \\ d_1 & o_1 & a_{1,1} & a_{1,2} \\ ... & ... & ... & ... \\ d_m & o_m & a_{m,1} & a_{m,2} \end{bmatrix} \qquad (1)$$

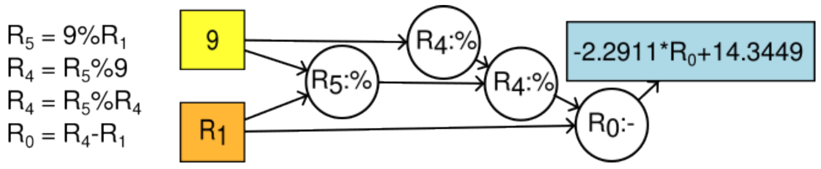
where $m \leq n$. Crossover can be done on a micro-scale, changing a single value, or on a macro-scale, adding or deleting a whole instruction with respect to the size limit n.

A small LGP model is shown in Fig 1. In this example, introns — instructions that do not contribute to the final output — are removed for clarity using the algorithm defined in [19]. However, introns themselves can be quite useful for searching [20] so they are kept during the actual evolutionary runs.

A CGP program is typically depicted as a one-dimensional string in the form

$$p \in P = o_0 a_{0,1} a_{0,2} o_1 a_{1,1} a_{1,2} ... o_n a_{n,1} a_{n,2} O_0 ... O_m, \qquad (2)$$

Fig 1. Example LGP Model. Directed graph depiction of an LGP individual where only effective (non-intronic) instructions are shown. The output R_0 is shown as a blue square, the input R_1 as an orange square, and constants in yellow. The operator % signifies the analytic quotient.

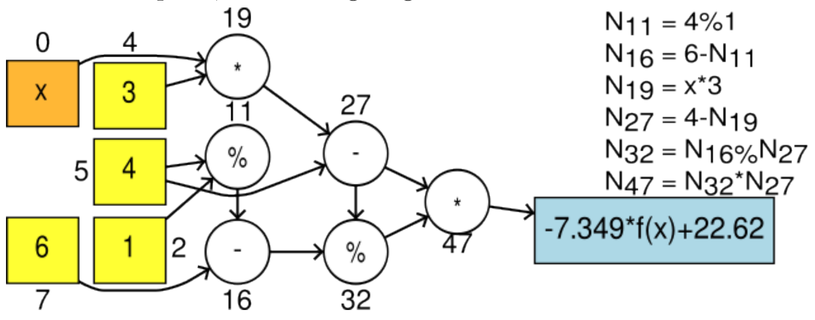


but we only use this form **during crossover**. For initialization, execution, and mutation, we use an encoding much closer to the LGP representation:

$$p = \begin{bmatrix} o_0 & a_{0,1} & a_{0,2} \\ o_1 & a_{1,1} & a_{1,2} \\ \cdots & \cdots & \cdots \\ o_n & a_{n,1} & a_{n,2} \end{bmatrix} \cup [O_0], \tag{3}$$

where each node, represented as a row, has an operator o, two connection genes ($a_{i,1}$, $a_{i,2}$) which are pointers to previous nodes, and in our univerate problems, a single output pointer O. Mutation in CGP is typically only done at the micro level because of the fixed size. Since crossover is traditionally not effective in CGP search, a $(1 + \lambda)$ strategy is commonly used. It is easier on resources to execute the program recursively from the output, and nodes which are not run are intronic, also referred to as **active nodes**. A small example is shown in Fig 2.

Fig 2. Example CGP program. Only active (non-intronic) nodes are shown. The input node I_0 is shown as an orange square, constants as yellow squares, the output node O_0 as a blue square, and scaling as green diamonds.



Crossover Operators

In both CGP and LGP, we test one- and two-point crossover, where as implied by the names, one or two points are chosen for each parent (which in CGP must be the same indices for both) and the relevant segments are exchanged. In LGP, we also test Uniform Crossover [15] where half of the instructions are swapped at random points, modified to fit our variable-size representations. In CGP we test Kalkreuth's Subgraph Crossover [10] which only exchanges active nodes and manages connections to make sure that nodes active in both parents remain active in the child.

Selection Operators

The primary selection operator used in this study is Elitist-Tournament selection with a tournament size of $n = 4$. In standard tournament selection [21], potential parents are separated into groups that are n-contestants large, and the best parent within each group is selected. Elitist-Tournament selection does not differ from this process except that the most fit individual from the population is selected once before any contests begin.

We also use Fitness-Proportionate selection [22] and ϵ-Lexicase selection [23] during the selection section of this study. In Fitness-Proportionate selection, parents are chosen randomly from the population with the percentage chance of being chosen rising as an individual's fitness improves.

In ϵ-Lexicase selection, parents are chosen because of an elite score on individual test cases rather than a high aggregate score when all test cases are combined. The basic process is as follows: First, the entire population is added to a candidate pool and all test cases are listed in a random order. Then, moving iteratively through the list of test cases, all candidates who are not within ϵ range of the highest observed score for the selected case are eliminated. The last surviving solution is returned as the selection winner. If multiple solutions survive all test cases, a winner is randomly chosen from among them. We use the difference between prediction and truth for individual data points from our baseline function as the test cases.

Mutation Operators

In the canonical CGP$(1 + \lambda)$ algorithm, mutation is the only operator for producing new individuals in the population. Thus, the performance of the algorithm is highly dependent on the proper choice of a mutation operator. Considering this, we want to investigate at which point the destructive effects of crossover in CGP can be mitigated and/or mediated by the usage of different mutation schemes. To analyze the effect of mutation operators on this phenomenon, we used three mutation operators:

- Basic mutation: In CGP, a single gene is changed; in LGP, either a single gene can be changed, a whole instruction can be removed, or a new instruction can be added to the program.
- Adaptive mutation: in this method, the mutation rate for each individual is adapted based on the average fitness for the population. Hence, if the fitness value for an individual is lower than the average fitness value for the population, the mutation rate for that individual increases, otherwise, if the fitness value for an individual is higher than the average fitness value for the population, the mutation rate for that individual decreases [16, 17].
- Swap mutation: In this method, two graph nodes are randomly selected and swapped in such a way that the structure of the graph is preserved [18].

Replicating the Phenomenon

We start by testing the previously-described crossover operators using the High Powered Computing Cluster at Michigan State University [24] with a crossover rate of 50% and a mutation rate of 2.5% since we use a (40+40) strategy (1/40 = 0.025), as summarized in Table 1. The maximum program size for LGP and the fixed program size for CGP was $n = 64$ instructions; in LGP we used four intermediate calculation registers. Our symbolic regression problems are taken from [25] and [26] to match those tested in [10], and are shown in Table 2.

Table 1. Description of evolutionary parameters to demonstrate the effects of crossover.

Notation	Xover	Mutation
CGP(1+4)	None	$4(\mu = 100\%)$
CGP(16+64)	None	$4(\mu = 100\%)$
CGP-1x(40+40)	One-Point (50%)	$\mu = 2.50\%$
CGP-2x(40+40)	Two-Point (50%)	$\mu = 2.50\%$
CGP-SGx(40+40)	Subgraph (50%)	$\mu = 2.50\%$
LGP-Ux(40+40)	Uniform (50%)	$\mu = 2.50\%$
LGP-1x(40+40)	One-Point (50%)	$\mu = 2.50\%$
LGP-2x(40+40)	Two-Point (50%)	$\mu = 2.50\%$

Table 2. Symbolic Regression Problems used in the experiment.

Problem	Function	Domain
Koza-1	$x^4 + x^3 + x^2 + x$	[-1, 1]
Koza-2	$x^5 - 2x^3 + x$	[-1, 1]
Koza-3	$x^6 - 2x^4 + x^2$	[-1, 1]
Nguyen-4	$x^6 + x^5 + x^4 + x^3 + x^2 + x$	[-1, 1]
Nguyen-5	$\sin(x^2)\cos(x) - 1$	[-1, 1]
Nguyen-6	$\sin(x) + \sin(x + x^2)$	[-1, 1]
Nguyen-7	$\ln(x + 1) + \ln(x^2 + 1)$	[0, 2]

Table Notes: Each trial took 20 points at random from the given domains.

For each problem, 50 trials were run with 10,000 generations each using Tournament Selection with Elitism by default. Our objective fitness function is based on the Pearson Correlation

$$f_i = 1 - r_i^2 \tag{4}$$

as suggested by Haut et al. because, unlike RMSE, it measures global rather than local error. For more details, see [27].

In Table 3 we can see the median fitness scores of the best models in each population. It is clear that CGP *without* crossover outperforms CGP with crossover, and CGP with crossover runs are routinely outperformed by their LGP counterparts. We also see, as an aside, that CGP-SGx, despite being evaluated in [10] as being more effective, performs worse than CGP (1+4) does in our paper, but that is likely because of different metrics and fitness functions between studies. In general, we see that **crossover is indeed detrimental to CGP search.**

Selection

Using conventional crossover methods in CGP applications appears to be detrimental. However, the choice of selection method may contribute to this effect. In the interest of eliminating alternative explanations for the crossover fitness gap, we compare the performance of CGP(1+4) with CGP-1x and CGP-2x using three different selection methods. These methods are elitist-tournament selection ($n = 1, 4, 8$), fitness-proportionate selection (roulette-wheel), and ϵ-lexicase selection ($\epsilon = 0.1$) [23]. Elitist-tournament selection is labelled as *etourn*. We test these configurations with 50 trials each on two baseline problems: Koza 1 and Koza 2, and display the average fitness progression as well as the distribution of each trial's most-fit solution.

The results for CGP1X and Koza 1 are shown in Fig 3. Of the selection methods, elite-tournament performed best while roulette wheel performed worst. However, none

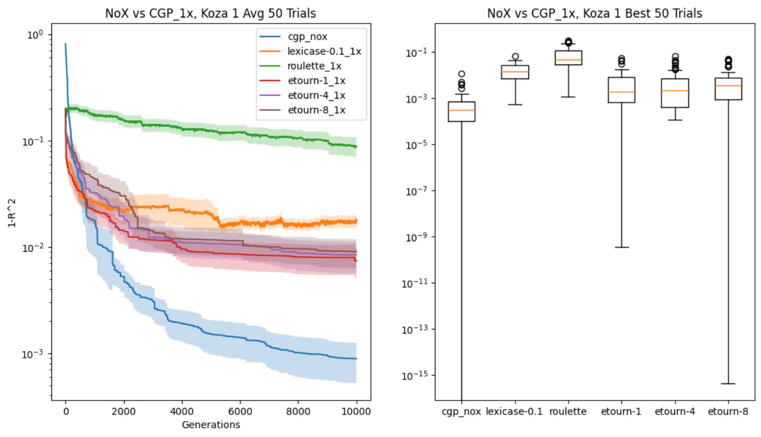
Table 3. Median correlation fitness for each problem and algorithm after 50 runs.

Algorithm	Koza-1	Koza-2	Koza-3	Nguyen-4	Nguyen-5	Nguyen-6	Nguyen-7
CGP(1+4)	0.00048	0.01080	0.01081	0.00040	0.00023	0.00044	0.00001
CGP(16+64)	**0.00022**	**0.00282**	**0.00512**	0.00030	**0.00006**	**0.00008**	$<$ **0.00001**
CGP-1x(40+40)	0.00252	0.04700	0.12883	0.00363	0.00205	0.00144	0.00012
LGP-1x(40+40)	**0.00100**	**0.01040**	**0.01220**	**0.00090**	0.00130	0.00075	$<$ **0.00001**
CGP-2x(40+40)	0.00215	0.04398	0.17211	0.00329	0.00198	0.00251	0.00017
LGP-2x(40+40)	**0.00085**	**0.00990**	**0.00810**	**0.00100**	**0.00010**	0.00070	$<$ **0.00001**
CGP-SGx(40+40)	0.00315	0.05150	0.15623	0.00438	0.00167	0.00153	0.00018
LGP-Ux(40+40)	0.00225	**0.02275**	**0.02775**	0.00200	0.00100	0.00110	**0.00010**

Table Notes: Smaller values are more fit. Algorithms that perform significantly better than their counterpart are shown in bold.

of the methods which employed one-point crossover outperformed CGP(1+4), which uses pure elitist selection and does not use crossover. These findings suggest that the fitness discrepancy observed in CGP with crossover is not the fault of poor selection methods and cannot be overcome by good selection methods. The remaining test configurations, seen in Figs 4-6 show similar results.

Fig 3. Fitness for Koza-1 using CGP-1x(40+40)



Mutation

In this set of trials, to test the effects of operator order, the traditional sequence of the evolutionary algorithm was used, i.e. first parent selection, then crossover to generate offspring, and finally mutation. We ran CGP, without crossover, CGP-1x and CGP-2x, on the functions in Table 2, for 50 trials using 10000 generations each, keeping all other parameters constant. The results in Figs 7-9 confirm the destructive effect of crossover on the CGP algorithm regardless of the employed mutation operator as CGP without crossover performed better in all cases.

Fig 4. Fitness for Koza-2 using CGP-1x(40+40)

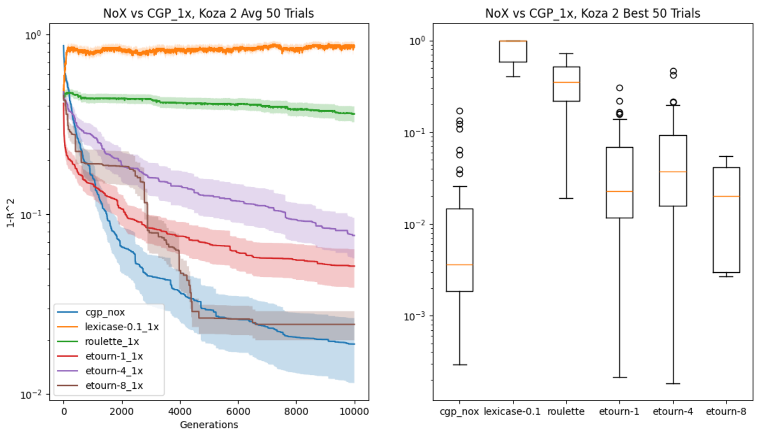
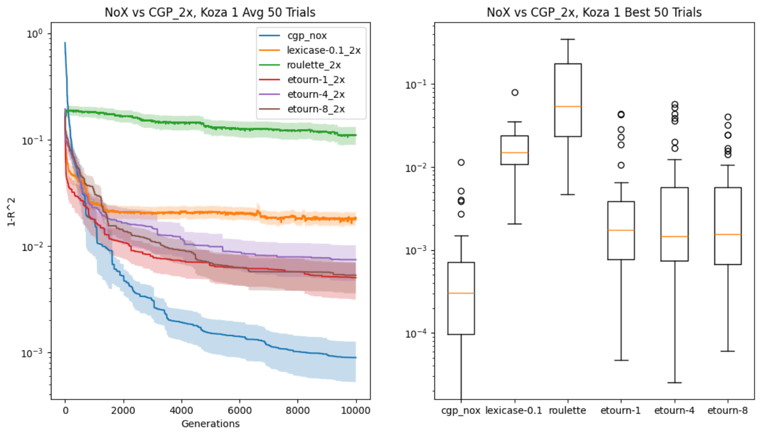


Fig 5. Fitness for Koza-1 using CGP-2x(40+40)



Explanation

Using an alignment-score-based method [28], we compared the similarity between the best-performing parent in a pair and their best child, resulting in Fig 10. It is rather obvious that children in LGP are far more similar to their parents than in CGP.

To explore further, we performed ablation testing on LGP-1x by testing how well it performs with different numbers of registers and whether or not it has the ability to perform better if program lengths are fixed. We thus tested with four, two, and no registers, and found that having an adequate number of registers is absolutely necessary for program stability, as is the ability to self-regulate program length, as shown in Fig 11. We hypothesize that the presence of registers allows high-fitness substructures to be "anchored" by the mediation of the registers. Since nodes in CGP are directly connected, changing a single connection gene can drastically alter, thus destroying, a parent program. We also posit that allowing the program to change size makes it less likely that cutting into a high-fitness substructure, or "amputation" occurs since there are more flexible choices of crossover points in *both* parents, thus avoiding a form of bloat [29], whereas in CGP having to use the same crossover point for both parents does

Fig 6. Fitness for Koza-2 using CGP-2x(40+40)

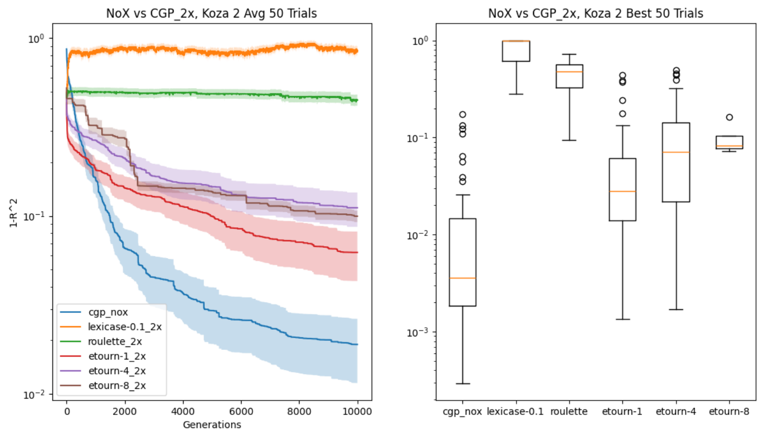
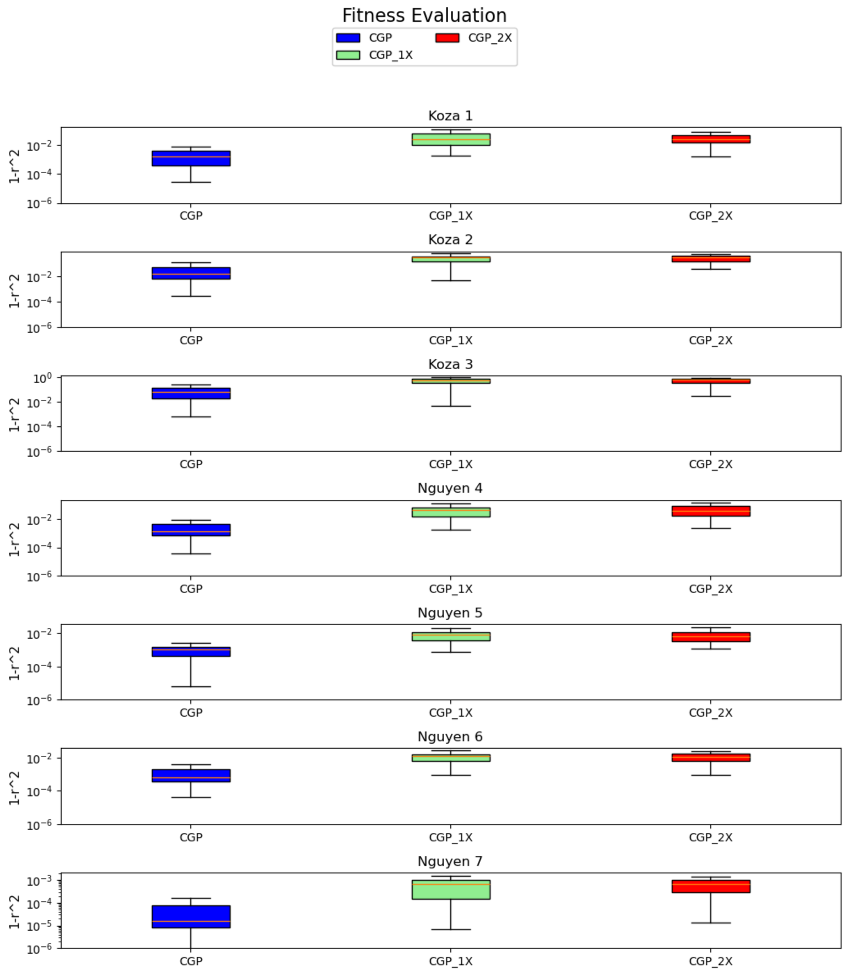


Fig 7. Fitness values on all functions in Basic Mutation



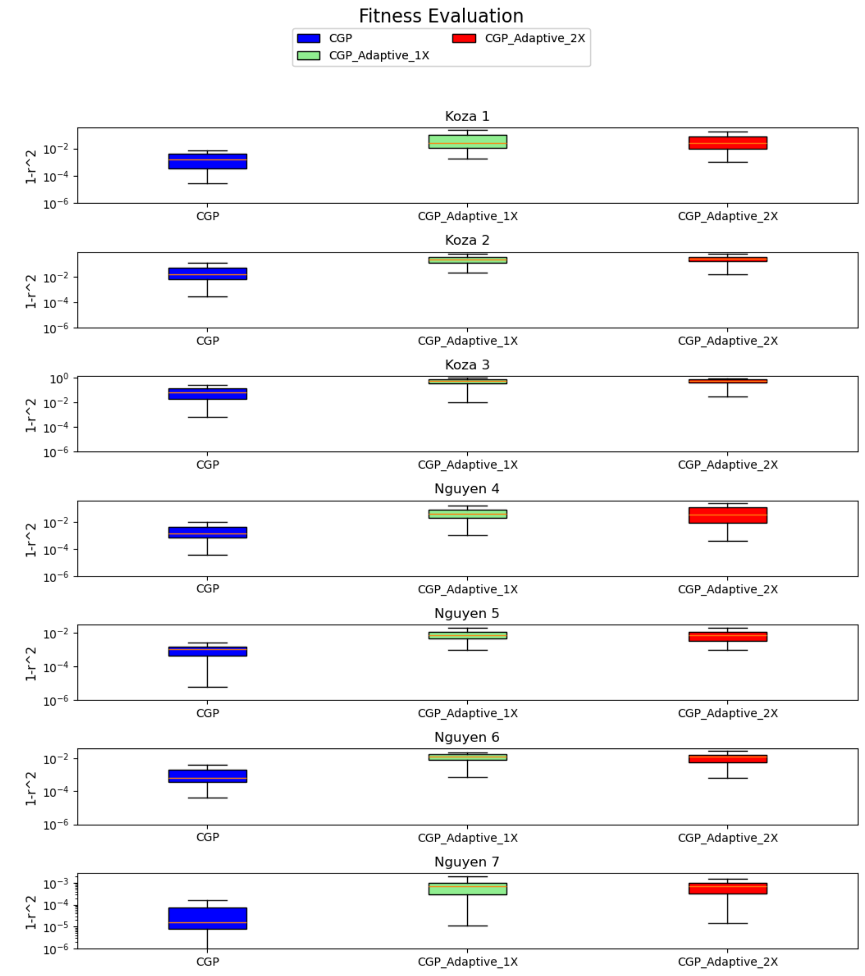


Fig 8. Fitness values on all functions in Adaptive Mutation

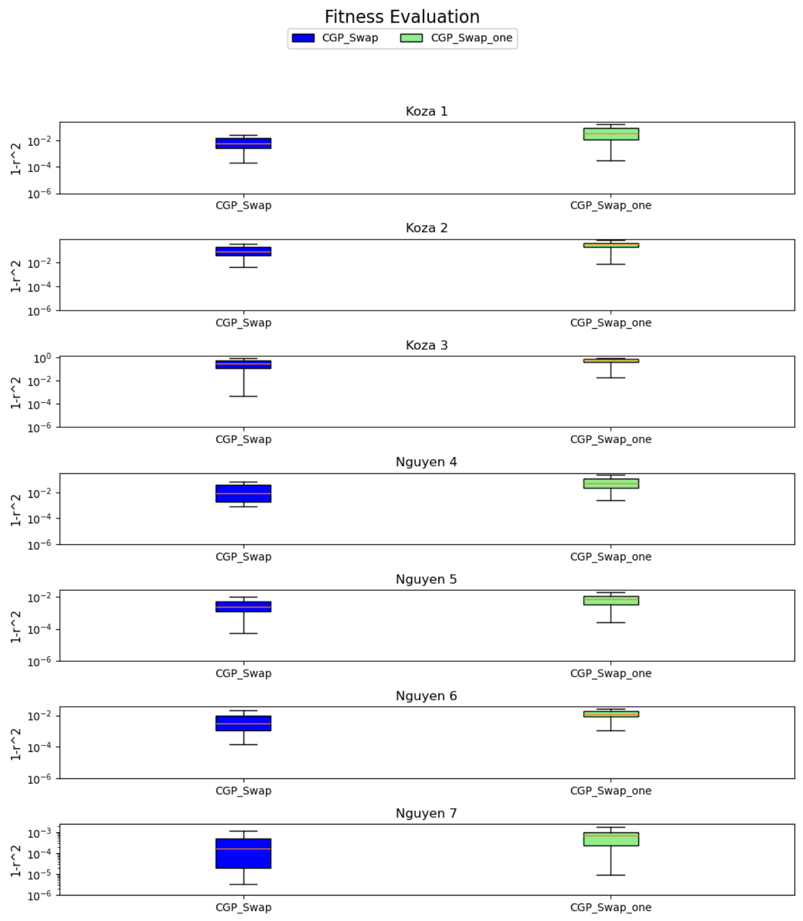
not decrease chances of amputation.

Discussion

In this work, we have seen that: CGP is hindered by crossover while LGP is not, that LGP produces children that tend to be more similar to their parents than CGP, and that LGP needs a sufficient amount of registers and the ability to regulate program size to search effectively. Additionally, our results suggest that this issue is not a consequence of selection or mutation operators. In the case of mutation operators, to support the claim with more evidence, we need to test our algorithms with different mutation operators and different mutation rates. Additionally, we can analyze different adaptive mutation operators in the literature. From these observations, we propose that intermediate calculation registers serve as anchors for LGP substructures, so that if one part of the program is destroyed by crossover, then other parts have more protection from these perturbations. We also hypothesize that LGP's ability to change program size provides more flexibility in recombining with high-fitness substructures intact, whereas in CGP it is easier to amputate a structure.

We thus have several opportunities to continue this line of research:

Fig 9. Fitness values on all function in Swap Mutation



- More rigorous testing of our hypotheses.
- Postulating *why* selection does not filter out poor-quality children.
- Attempts to hybridize LGP and CGP or otherwise engineer a new form of CGP crossover or mutation that takes these observations into account.
- Measuring the impact of similarity on search.

Acknowledgments

We would like to thank Drs. Christopher Adami and Wolfgang Banzhaf for their guidance and support.

Fig 10. Similarity between best parent and their best child. LGP methods produce children that are far more similar to their parents than CGP methods do.

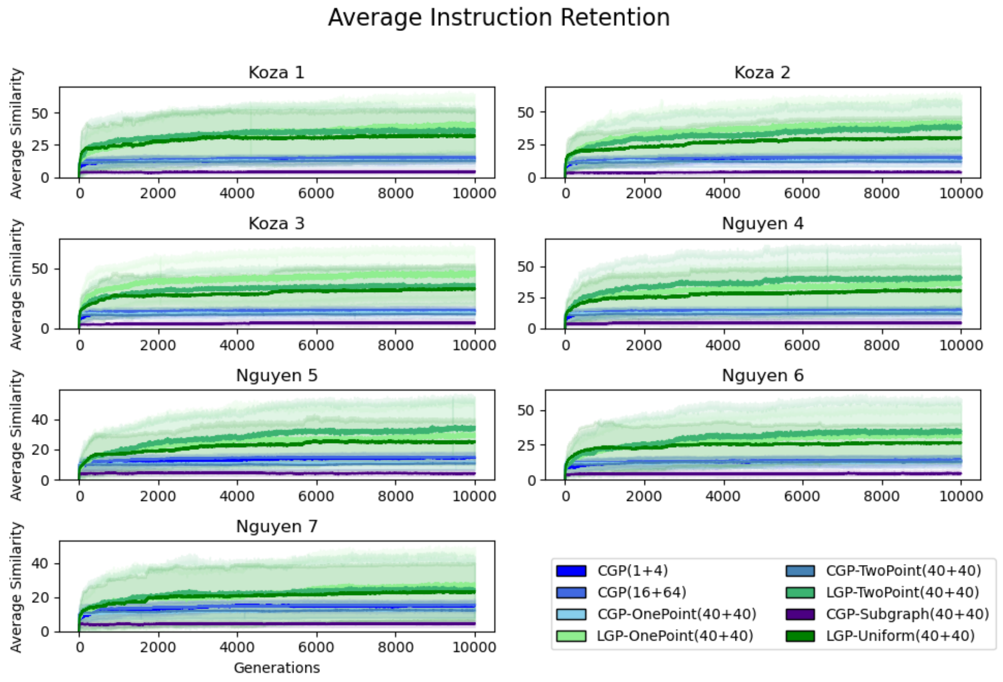
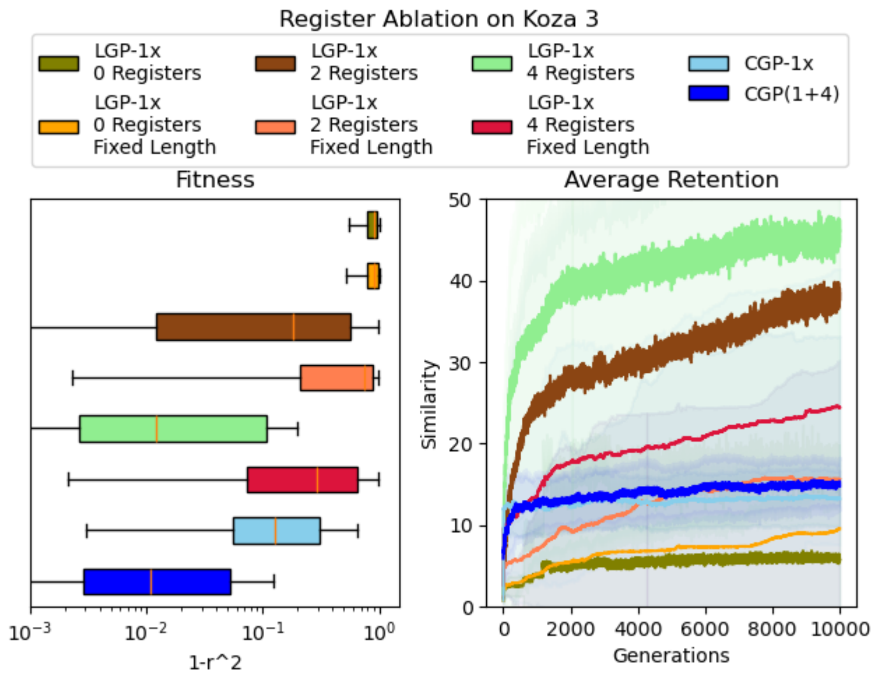


Fig 11. Ablation testing for Register count and Program Size Change. It is clear that having insufficient registers and/or fixing program length is poor for LGP.



References

1. Koza JR. Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press; 1992.
2. Koza JR. Genetic programming as a means for programming computers by natural selection. Statistics and Computing. 1994;4:87–112.
3. Miller JF, Harding SL. Cartesian Genetic Programming. In: Proceedings of the 10th Annual Conference Companion on Genetic and Evolutionary Computation. ACM Press; 2008. p. 2701–2726.
4. Miller JF, et al. 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. Morgan Kaufmann; 1999. p. 1135–1142.
5. Banzhaf W, Nordin P, Keller RE, Francone FD. Genetic programming: An Introduction — On the Automatic Evolution of Computer Programs and its Applications. Morgan Kaufmann Publishers Inc.; 1998.
6. Wilson G, Banzhaf W. A Comparison of Cartesian Genetic Programming and Linear Genetic Programming. In: Genetic Programming: 11th European Conference, EuroGP 2008, Naples, Italy, March 26-28, 2008. Proceedings 11. Springer; 2008. p. 182–193.
7. Clegg J, Walker JA, Miller JF. A New Crossover Technique for Cartesian Genetic Programming. In: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation. ACM Press; 2007. p. 1580–1587.
8. da Silva JE, Bernardino HS. Cartesian Genetic Programming with Crossover for Designing Combinational Logic Circuits. In: 2018 7th Brazilian Conference on Intelligent Systems (BRACIS). IEEE Press; 2018. p. 145–150.
9. Husa J, Kalkreuth R. A Comparative Study on Crossover in Cartesian Genetic Programming. In: Genetic Programming: 21st European Conference, EuroGP 2018, Parma, Italy, April 4-6, 2018, Proceedings 21. Springer; 2018. p. 203–219.
10. Kalkreuth R. A Comprehensive Study on Subgraph Crossover in Cartesian Genetic Programming. In: Proceedings of the 12th International Joint Conference on Computational Intelligence (IJCCI 2020); 2020. p. 59–70.
11. Kalkreuth R. Reconsideration and extension of Cartesian genetic programming. Technical University of Dortmund, Germany; 2021.
12. Torabi A, Sharifi A, Teshnehlab M. Using Cartesian Genetic Programming Approach with New Crossover Technique to Design Convolutional Neural Networks. Neural Processing Letters. 2023;55(5):5451–5471.
13. Cai X, Smith SL, Tyrrell AM. Positional Independence and Recombination in Cartesian Genetic Programming. In: European Conference on Genetic Programming. Springer; 2006. p. 351–360.
14. Cui H, Margraf A, Heider M, Hähner J. Towards Understanding Crossover for Cartesian Genetic Programming. In: Proceedings of the 15th International Joint Conference on Computational Intelligence (IJCCI 2023). SCITEPRESS; 2023. p. 308–314.

15. Oltean M, Groşan C, Oltean M. Encoding Multiple Solutions in a Linear Genetic Programming Chromosome. In: International Conference on Computational Science. Springer; 2004. p. 1281–1288.

16. Kalkreuth R, Rudolph G, Krone J. Improving Convergence in Cartesian Genetic Programming using Adaptive Crossover, Mutation and Selection. In: 2015 IEEE Symposium Series on Computational Intelligence. IEEE; 2015. p. 1415–1422.

17. Marsili Libelli S, Alba P. Adaptive Mutation in Genetic Algorithms. Soft Computing. 2000;4:76–80.

18. Alves SS, Oliveira SA, Neto ARR. A Novel Educational Timetabling Solution Through Recursive Genetic Algorithms. In: 2015 Latin America Congress on Computational Intelligence (LA-CCI). IEEE; 2015. p. 1–6.

19. Brameier M, Banzhaf W, Banzhaf W. In: Linear Genetic Programming. Springer; 2007. p. 36–37.

20. Sotto LFDP, Rothlauf F, de Melo VV, Basgalupp MP. An Analysis of the Influence of Noneffective Instructions in Linear Genetic Programming. Evolutionary Computation. 2022;30(1):51–74.

21. Miller BL, Goldberg DE. Genetic Algorithms, Tournament Selection, and the Effects of Noise. Complex Systems; 1995.

22. Hancock PJ. An Empirical Comparison of Selection Methods in Evolutionary Algorithms. AISB Workshop on Evolutionary Computing. 1994;.

23. La Cava LS William, Danai K. Epsilon-Lexicase Selection for Regression. Proceedings of the Genetic and Evolutionary Computation Conference. 2016;.

24. Hardware — Institute for Cyber-Enabled Research; 2024. `https://icer.msu.edu/hpcc/hardware`.

25. Koza JR. Genetic Programming II: Automatic Discovery of Reusable programs. MIT Press; 1994.

26. Uy NQ, Hoai NX, O'Neill M, McKay RI, Galván-López E. Semantically-Based Crossover in Genetic Programming: Application to Real-Valued Symbolic Regression. Genetic Programming and Evolvable Machines. 2011;12:91–119.

27. Haut N, Banzhaf W, Punch B. Correlation Versus RMSE Loss Functions in Symbolic Regression Tasks. In: Genetic Programming Theory and Practice XIX. Springer; 2023. p. 31–55.

28. Aygün E, Ecer D. python-alignment; 2017. `https://github.com/eseraygun/python-alignment`.

29. Banzhaf W, Bakurov I. On The Nature Of The Phenotype In Tree Genetic Programming. arXiv. 2024;2402.08011.