Crossover Destructiveness in Cartesian versus Linear Genetic Programming

Mark Kocherovsky[1,2] and Wolfgang Banzhaf[1,3]
[1]Department of Computer Science and BEACON Center for the Study of Evolution in Action
Michigan State University, East Lansing, MI 48824, USA
[2]kocherov@msu.edu [3]banzhafw@msu.edu

Abstract

Cartesian Genetic Programming (CGP) literature repeatedly reports that crossover operators hinder CGP search compared to a $1 + \lambda$ strategy based on mutation only. Though there have been efforts in making CGP crossover operators work, the literature is relatively evasive on why the phenomenon is observed at all. This contrasts with what happens in Linear Genetic Programming (LGP), where we know that crossover works well. While both CGP and LGP individuals can be represented as directed acyclic graphs (DAGs), changing a single connection gene in a CGP individual can drastically alter the activeness of nodes in the entire graph, as opposed to LGP where crossover changes are much more beneficial. In this contribution, we demonstrate the phenomenon and show that LGP evolution produces children that are far more similar to their parents than in CGP. This lets us propose that the design of LGP, namely the inclusion of steady-state memory registers and program size regulation, serves to protect high-fitness substructures from perturbation in a way that is not provided for in CGP.

Introduction

Genetic Programming (GP) was first introduced by Koza to evolve programs to solve symbolic regression and Boolean problems by representing programs as trees (Koza, 1992, 1994a). Over the last three decades, we have seen the introduction of new paradigms and methods in the search for ever-better models. Two of these are Linear Genetic Programming (LGP), where programs are represented as a sequence of instructions (Banzhaf et al., 1998), and Cartesian Genetic Programming (CGP), where a program is represented as a Directed Acyclical Graph (DAG) (Miller et al., 1999; Miller and Harding, 2008).

Though both structures can be drawn as a DAG and are thus comparable in structure (Wilson and Banzhaf, 2008), the rule of thumb is that traditional crossover operators are detrimental to CGP exploration (Clegg et al., 2007), whereas it is mandatory for LGP. Although CGP crossover techniques beyond the standard one-point or two-point crossover have been proposed (Clegg et al., 2007; Kalkreuth, 2020), it still remains standard practice to run CGP using $(1 + \lambda)$ strategies with mutation only. Though the phenomenon has been repeatedly noted – and replicated in this paper – researchers are not quite sure *why* this performance discrepancy exists.

We posit that one- or two-point crossover in CGP is detrimental because introducing an entirely different set of connections in the CGP graph is likely to activate a set of introns (genetic material that did not contribute to the output until then) that may have been undergoing some form of changes, as well as de-activating high-fitness structures, which has the combined effect of drastically changing the graph trace. In LGP, on the other hand, the presence of registers allows the graph to “anchor” substructures so that if one substructure is perturbed by crossover or mutation, the other substructures are less likely to be damaged. LGP also benefits from not fixing the genotype size as it happens in CGP; this likely allows crossover to better protect substructures from amputation.

We first discuss related work in the field of CGP crossover before introducing details about CGP and LGP. We then explain our experimental method, discuss our results, and conclude with our main takeaways.

Related Work

Most work in CGP crossover focuses on developing working crossover operators. Clegg et al. developed a crossover technique for CGP using real-valued parameters after demonstrating that one-point and single-gene crossover are ineffective operators (Clegg et al., 2007). Da Silva and Bernardino used elitism to select the best individuals to serve as parents, thus potentially preserving good subgraphs (da Silva and Bernardino, 2018). Torabi et al. designed a crossover technique that aligns parents to encourage their similarity in order to decrease destructiveness (Torabi et al., 2023). Kalkreuth has published a number of contributions on CGP crossover, with several newly developed methods such as phenotype diversity measurement (Kalkreuth et al., 2015), swapping only active nodes in the form of subgraphs (Kalkreuth, 2020, 2021), and modularization (Husa and Kalkreuth, 2018).

There has been little discussion of *why* crossover is a hin-

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/isal2024/36/20/2461083/isal_a_00735.pdf by guest on 17 January 2025

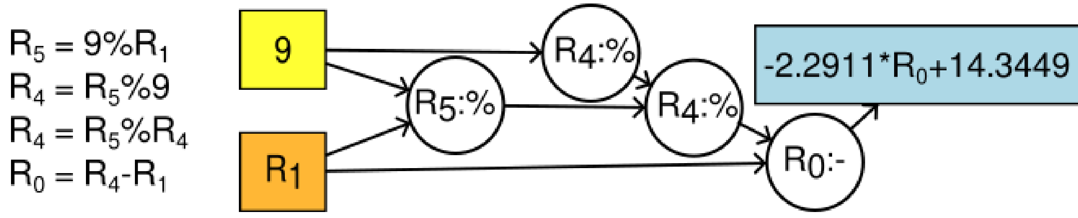

Figure 1: Directed graph depiction of an LGP individual where only effective (non-intronic) instructions are shown. The output R_0 is shown as a blue square, the input R_1 as an orange square, and constants in yellow. The operator $\%$ signifies the analytic quotient.

drance to CGP in the first place. Cai et al. (2006) proposed that one of the reasons for this failure is that the phenotype is dependent on the position of genes in the chromosome. Cui et al. (2023) build off Cai et al. (2006) by claiming that the directed requirement forces a non-uniform probability for connection mutation, which they call "positional bias". They argue that nodes closer to the input have a higher probability of being active causing new nodes closer to the output to connect to previously inactive nodes and making them overly destructive to the graph structure. However, LGP and Tree GP have similar ordering issues as they are step-by-step algorithmic procedures too. Wilson and Banzhaf have shown that the main point of divergence between LGP and CGP is the nature of input connection restrictions: while CGP forces acyclicity, LGP has to rely on registers to store memory Wilson and Banzhaf (2008).

LGP and CGP

LGP

In LGP, individual programs read from and write to a set of registers $r_n \in R$ where some are read-only (input or constant) values and the remaining registers are read-write, initialized to zero at the outset. In conventional implementations of LGP, the output is stored in r_0, multi-dimensional input is stored in $r_i \in [r_1, ..., r_j]$, and the results of intermediate calculations are stored in registers $r_c \in [r_{j+1}, r_k]$. An individual instruction (a gene) is thus composed of four items: A destination register $d \in r_0 \cup r_c$, an operator o, and some operands, which are drawn from $a \in R$. Hence, an LGP instruction directs the program to read values from any i registers, perform an operation, and store the result in any valid register. Note that memory cells we call registers can also hold more complex data structures like vectors or matrices (with corresponding operations applied in instructions), as Real et al. (2020) point out.

In this contribution, LGP encoding takes the form

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

where program p in the population P is a two-dimensional matrix with $m \leq n$ instructions. LGP conventionally allows programs to be of variable size, and unless explicitly mentioned, we follow suit in this paper.

Crossover and mutation are relatively simple operations. Standard one- and two-point crossover cuts between instructions, as does macro-mutation, while micro-mutation edits the internals of an instruction. In addition to those methods, we also test uniform crossover, where half of an individual's instructions are chosen at random for recombination (Oltean et al., 2004). Crossover in particular is conducted instruction-wise rather than point-wise as it would be with a one-dimensional vector.

The phenotype of a LGP individual can be seen as a directed graph using the method described in Brameier et al. (2007) to remove non-effective instructions. An example individual phenotype is shown in Figure 1 trained on the Nguyen-4 problem (Uy et al., 2011) which takes one constant value and the input value, and after four instructions returns a result, which is then linearly scaled because of our correlation-based fitness measurement.

LGP individuals often feature instructions in the genotype that contribute nothing to the phenotype; these are called **introns**. Introns generally make up a significant proportion of instructions in an individual. Though such instructions do not affect the calculation, they are stilluseful to provide new genetic material as they can be activated in future generations, and are considered generally beneficial to search (Sotto et al., 2022).

CGP

A CGP individual is an encoding of a directed acyclic graph (DAG). Each instruction is encoded as a "node", which usually contains two connections (which point to the output of previous nodes) and a single operator. There are also final output nodes which can reference any node or input. The acyclic character of the graph is enforced in evolutionary operations by preventing connections from pointing to nodes further forward than the current node. However, instead of reading to and writing from steady-state registers, nodes take the output of previous nodes as input.

Like LGP individuals, a CGP program can be represented with a linear vector of genes, so basic crossover and muta-

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/isal2024/36/20/2461083/isal_a_00735.pdf by guest on 17 January 2025

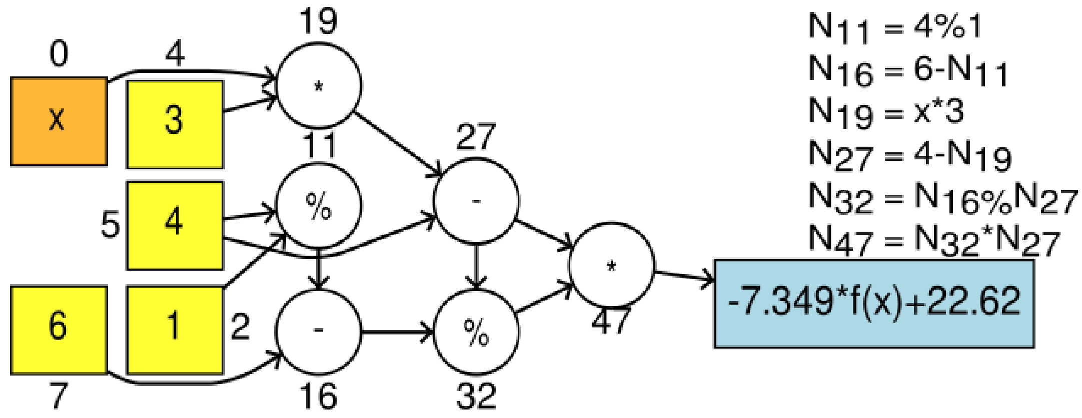

Figure 2: DAG depiction of an CGP individual where only active (non-intronic) nodes are shown. The input node I_0 is shown as an orange square, constants as yellow squares, the output node O_0 as a blue square, and scaling as green diamonds.

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/isal2024/36/20/2461083/isal_a_00735.pdf by guest on 17 January 2025

tion methods are easy to implement. However, due to the known crossover problems, the standard practice is a $(1+\lambda)$ approach, where normally $\lambda = 4$ (Clegg et al., 2007; Cui et al., 2023). Intronic nodes, which are not called in the graph trace, are labelled inactive nodes that can be activated (and active nodes can be disabled) by evolutionary operators.

An example CGP individual is shown in Figure 2. Since CGP cannot have cycles, each instruction is shown individually. Conventionally, a CGP individual with an arity of two is encoded as a vector

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

where program p is part of the population P and consists of n, nodes, each containing some operator o and two nodes to take as input a_{n1} and a_{n2}, ending with a set of output nodes O. While we use the one-dimensional vector for point crossover, we use a two-dimensional matrix for other operations:

$$p = \begin{bmatrix} o_0 & a_{0,1} & a_{0,2} \\ o_1 & a_{1,1} & a_{1,2} \\ ... & ... & ... \\ o_n & a_{n,1} & a_{n,2} \end{bmatrix} \cup [O_0, O_{m-1}], \quad (3)$$

which is analogous to our LGP representation and is simply easier to implement in Python 3.

Observation of the CGP Crossover Phenomenon

To confirm that crossover is indeed a hindrance to CGP, we tested five CGP methods and 3 LGP methods. Our CGP methods included two mutation-only methods, standard CGP(1+λ) and also CGP(16+64), plus three methods with crossover, CGP(40+40) with both one- and two-point crossover, and Kalkreuth's Subgraph Crossover method (Kalkreuth, 2020). Our LGP methods used one-point, two-point, and uniform crossover Oltean et al. (2004) with four calculation registers. In addition, we tested LGP(1+λ) without crossover, but the results were so poor that they are not included in our analysis. We fix CGP individuals to 64 instructions, and impose a limit of 64 instructions on LGP. If an individual ends up exceeding this limit, instructions are then deleted at random.

We used the set of four arithmetic operators, addition, subtraction, multiplication, and the analytic quotient (Ni et al., 2012). Each algorithm was run 50 times for each problem described below (350 trials per algorithm, or 2800 trials in total) for 10000 generations, and a tournament size of 4, a mutation rate of 2.5% and a crossover rate of 50% where applicable. Each trial was run on MSU's High-Powered Computing Cluster (ice, 2024). Table 1 summarizes our notation and operators used for these algorithms.

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

Instead of the conventional RMSE fitness function, we use the correlation fitness function described in Haut et al. (2023), where

$$f_i = 1 - r^2, \quad (4)$$

that is to say, the fitness f of individual i is dependent on the Pearson correlation coefficient r. We use correlation rather than RMSE because it has been shown to be more effective

Algorithm	Koza-1	Koza-2	Koza-3	Nguyen-4	Nguyen-5	Nguyen-6	Nguyen-7
CGP(1+4)	0.00054	0.01047	0.02448	0.00072	0.00044	0.00045	0.00001
CGP(16+64)	**0.00012**	**0.00297**	**0.00321**	**0.00016**	**0.00004**	**0.00006**	$<$ **0.00001**
CGP-1x(40+40)	0.00178	0.05296	0.09669	0.00245	0.00168	0.00136	0.00002
LGP-1x(40+40)	0.00105	**0.01445**	**0.01850**	**0.00110**	**0.00075**	0.00065	$<$ 0.00001
CGP-2x(40+40)	0.00121	0.02355	0.1105	0.00277	0.00152	0.00141	0.00005
LGP-2x(40+40)	**0.00040**	**0.00855**	**0.01325**	**0.00060**	**0.00015**	0.00070	$<$ **0.00001**
CGP-SGx(40+40)	0.00362	0.02157	0.11874	0.00315	0.00106	0.00115	0.00011
LGP-Ux(40+40)	0.00265	0.01995	**0.01795**	0.00275	0.00060	0.00090	0.00020

Table 2: Median correlation fitness for each problem and algorithm after 50 runs. Algorithms that perform significantly better than their counterpart are shown in bold.

in search since it takes into account the shape of the desired function, not just local error. Linear coefficients are then found to be able to scale predictions to match the range of the training function.

Our algorithms are applied on seven univariate objective functions, three from Koza (1994b) (labeled Koza-1/2/3) and four from Uy et al. (2011) (labeled Nguyen-4/5/6/7), shown in Table 3, to match the test functions in Kalkreuth (2020). These are relatively simple problems, and testing on more complex problems deserves a more in-depth investigation. Our training sets consist of twenty randomly-chosen values from the domain of [-1, 1], except for Nguyen-7, which has a domain of [0, 2]. Code is available at `https://github.com/MarkKocherovsky/cgp_crossover/tree/main`.

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 3: Symbolic Regression Problems used in the experiment. Each trial took 20 points at random from the given domains.

Because we are interested in the relative performance of our algorithms, we only consider results on trained data, not on unseen test data. Table 2 shows the median fitness for each algorithm applied to each problem. As previously mentioned, LGP$(1+4)$, with a median performance of 1.0 on each problem, is not shown here. Pairwise Mann-Whitney Test results can be found at `https://github.com/MarkKocherovsky/cgp_crossover/tree/main/output/significance`. The mutation-only CGP(16+64) is clearly the best performer in most problems. CGP without crossover significantly outperforms all crossover CGP methods, demonstrating that crossover is indeed harmful to CGP's search process, further substantiated by the full range of statistics, shown in Figure 3. We also see that, as a whole, CGP tests with different crossover operators tend to be significantly outperformed by their LGP counterparts. In summary, CGP without crossover significantly outperforms CGP with crossover. Therefore, **we have successfully replicated the CGP Crossover Phenomenon**.

Even CGP-SGx seems to be underperforming compared to CGP(1+4), which contradicts Kalkreuth (2020). We speculate that this is caused by our usage of correlation fitness rather than RMSE and Kalkreuth's measure of iterations to convergence rather than our method of checking fitness after 10,000 generations. This warrants further study.

Explanation of Phenomenon

We used an alignment scoring algorithm to measure how much the best parent out of two parents matched their best-performing child (Aygün and Ecer, 2017), where higher similarity scores indicate more similar individuals. In Figure 4, we can see that overall, LGP methods tend to produce children that are far more similar to their parents than CGP methods. At first, this seems intuitive: Changing only a single connection gene in CGP can completely change the graph trace, and in one- or two-point crossover recombination provides ample opportunity for changes. However, changing instructions in LGP can also disable or enable instructions, yet we do not see the same destructiveness in LGP. This is likely due to two factors, the primary factor being the presence of calculation registers, and the secondary being the variable length of LGP individuals.

To test the effect of registers on LGP, three new experiments were conducted using the Koza-3 problem, chosen because it seems to be the most difficult problem out of the set. We used three new pairs of LGP-1x configurations, where there were four, two, or no calculation registers, and either fixed-length or variable-length individuals.

We posit that an LGP program with no calculation registers is essentially a feedforward graph; the source registers

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/isal2024/36/20/2461083/isal_a_00735.pdf by guest on 17 January 2025

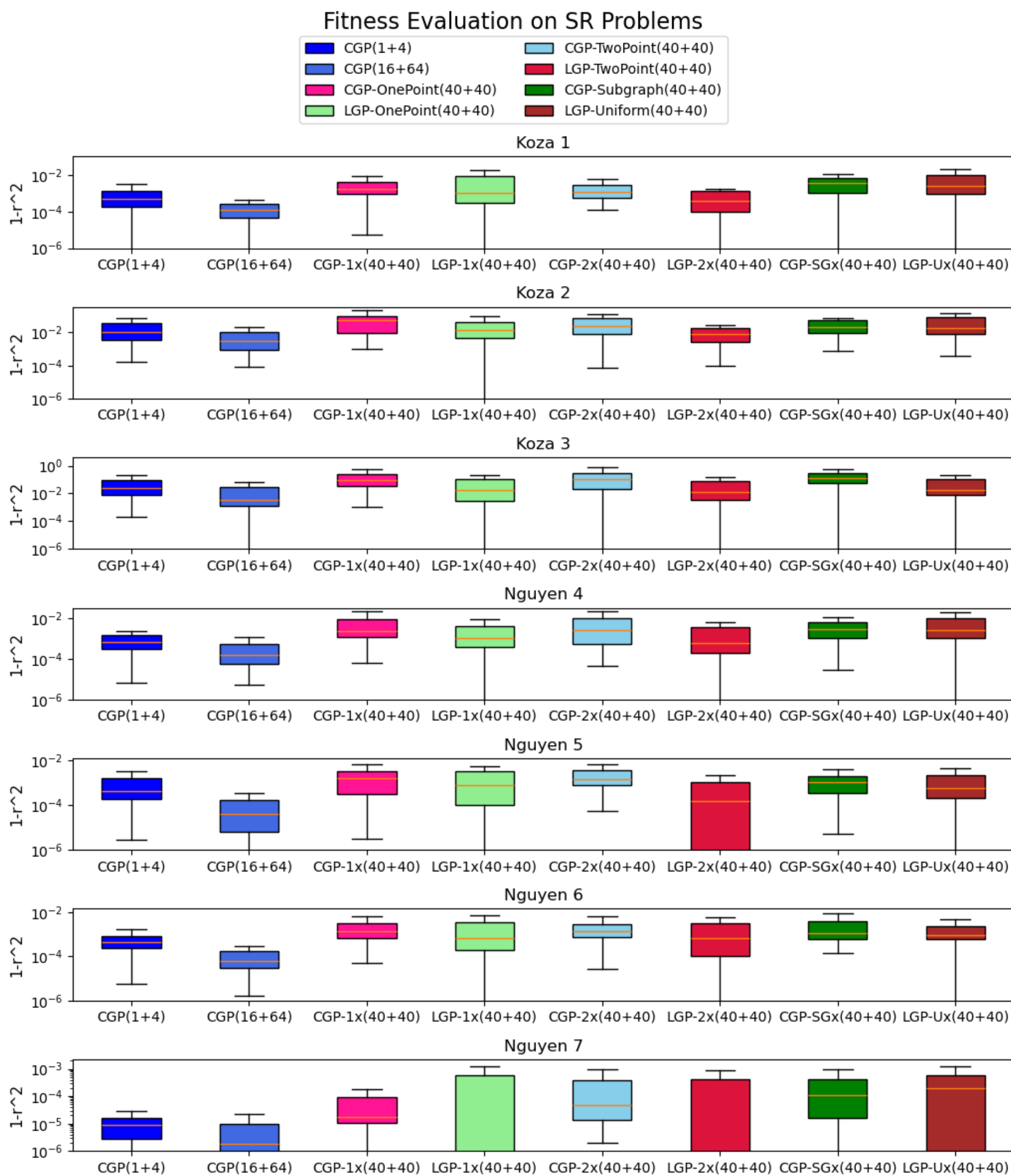

Figure 3: Fitness statistics of each algorithm. Data points outside of 150% of the interquartile range are not shown.

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/isal2024/36/20/2461083/isal_a_00735.pdf by guest on 17 January 2025

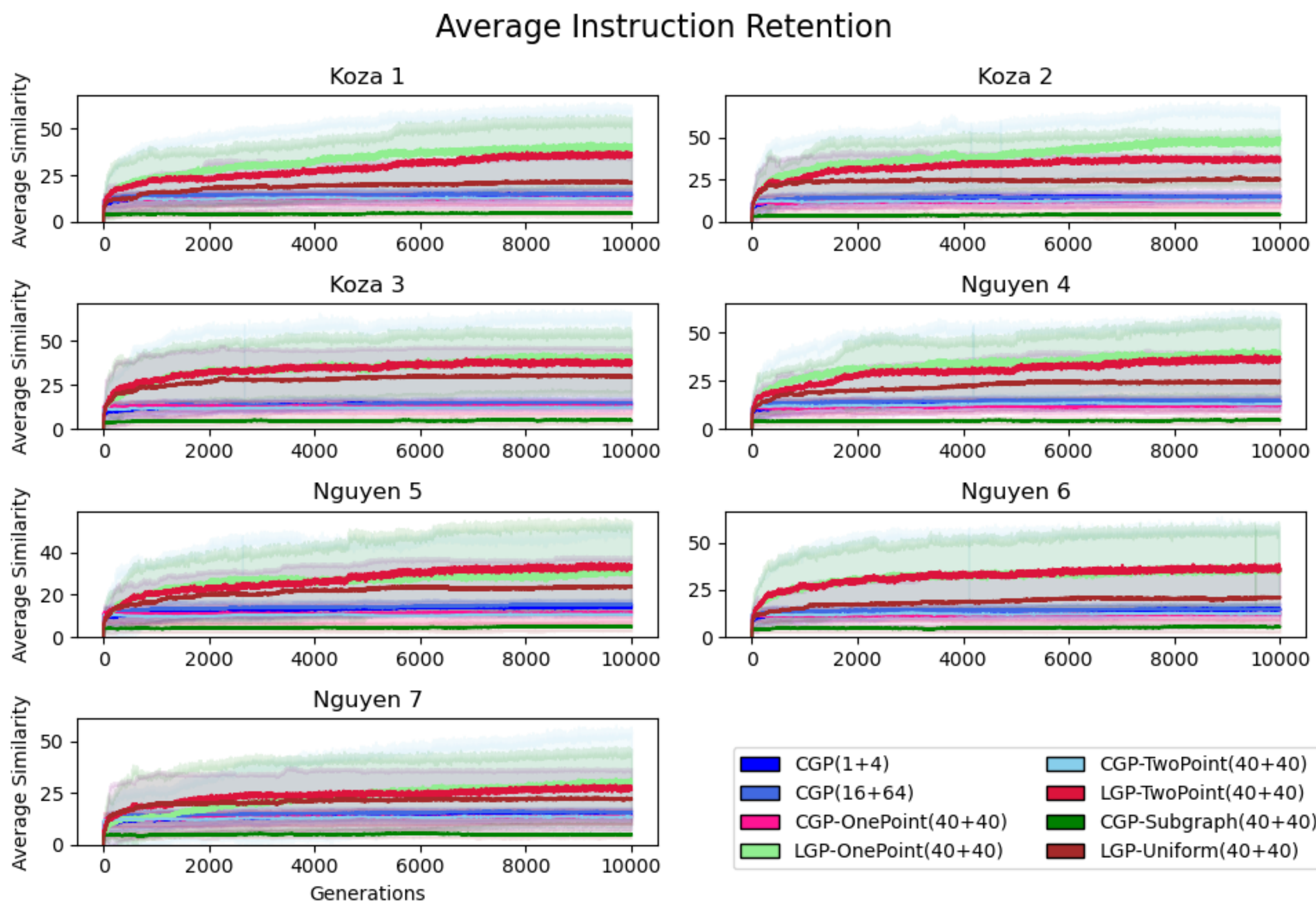

Figure 4: Similarity between best parents and best children within each pair of parents over time.

can only be R_0 (output), R_1 (input), or a constant, where R_0 can hold intermediate results as the program executes. A CGP program (also of fixed size) can also only call from an input node, constants, or an intermediate instruction node. However, the linear and graph natures of the two paradigms seems to play a distinct role in search, as we will see.

Figure 5 shows that CGP(1+4) and LGP-1x(40+40) with four registers significantly outperform the other tested methods. When comparing pairs of fixed and variable length, LGP-1x with two registers performs significantly better when the individuals are of fixed size, and LGP-1x without registers does not perform significantly worse when the length is variable. However, with our "standard" four registers and variable length, LGP-1x performs far better than if the four-register population's length is fixed. Having four registers is significantly more effective than having two or no registers of either individual length configuration. The population of two-register individuals with fixed length performed significantly better than either zero-register strategy, but the varied length counterpart did not perform significantly differently. It is also clear that children in fixed-length variants are much less similar than those in variable-length runs except where there are zero registers. Finally, we observe that having more calculation registers is vital for similarity.

These results clearly show that the number of registers matters in evolving solutions to symbolic regression problems. This supports our hypothesis that registers act as "anchors" to prevent the destruction of substructures during crossover. LGP is not designed to act as a feed-forward mechanism and thus needs to rely on the ability to store and recall intermediate results. This ability means that if one substructure is perturbed by an operation, other substructures are more likely to remain intact by virtue of lack of interactions with the registers relevant to the former substructure, even if their instructions are activated or deactivated.

It is also evident that even with an adequate number of registers, LGP individuals should not be fixed to a certain length. We conjecture, following Banzhaf and Bakurov (2024), that allowing programs to change size through evolutionary operators allows the population to **self-regulate** complexity, particularly by avoiding "amputation"

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/isal2024/36/20/2461083/isal_a_00735.pdf by guest on 17 January 2025

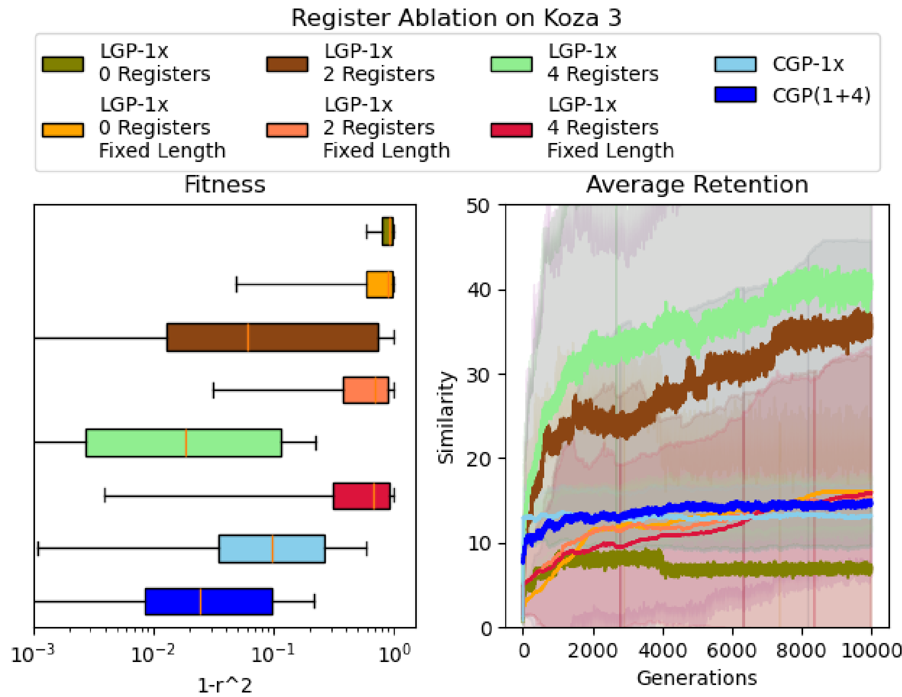

Figure 5: Statistics of runs on Koza-3 with register and length ablation testing.

of highly-fit substructures. A fixed-size representation, on the other hand, may be more likely to destroy good substructures due to a lack of flexibility in choosing the crossover point.

Discussion and Further Work

In this paper we have made several relevant observations:

- Conventional crossover methods are not effective for program search using Cartesian Genetic Programming, but are effective in Linear Genetic Programming.
- When comparing the best individuals from a pair of parents and their respective children, LGP methods produce children that tend to be more similar to their parents than CGP methods.
- LGP needs a sufficient number of calculation registers and the ability to regulate individual size to help search progress.

These observations lead us to propose that

- Registers in LGP act as anchor points for a program's overall structure, granting program substructures **robustness** against the alteration of other substructures. As registers are not present in conventional CGP, the graph's structure is too vulnerable to operator-caused destruction.
- The ability to regulate program size allows LGP to discard or add harmful or helpful instructions and/or substructures, respectively, even if all the affected nodes are intronic, thus allowing changes to be more cohesive and thus possibly more effective. CGP lacks this ability, and thus leaves itself more vulnerable to the transfer of incomplete modules.

These observations and conclusions allow us to make a third contribution: proposing new avenues for the study of a hitherto only weakly explored phenomenon:

- Studies pertaining to the actual properties of the graphs and their substructures to determine what encourages (or discourages) the usage of a particular substructure over others, possibly using insights and methods described in Hu and Banzhaf (2018).
- Graph studies to empirically ascertain registers' use as anchor points and to directly observe their effects on program similarity: measures and metrics that detect and track substructures through generations might give insight into relevant common properties.
- Studies on the effects of variable versus fixed-length individuals on LGP.
- Tests of different mutation and selection operators to see if they contribute to this phenomenon.

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/isal2024/36/20/2461083/isal_a_00735.pdf by guest on 17 January 2025

- Creation of new CGP variants and CGP-LGP hybrids which would allow us to study the direct effect of registers and length-variability on crossover.
- Studies of different encodings of CGP and LGP phenotypes and their treatment by evolutionary operators.
- Measurement of the effect of program similarity on search.
- Testing on more complex problems, such as the Ackley (1987) and Griewank (1981) functions.

Acknowledgements

This work was supported by a Michigan State University Doctoral Fellowship and used computational tools hosted by the MSU Institute for Cyber-Enabled Research. We would like to thank Dr. Illya Bakurov, Ms. Marzieh Kianinejad, and Mr. Elijah Smith for their guidance and advice.

References

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

Ackley, D. (1987). A Connectionist Machine for Genetic Hill-climbing. *The Kluwer International Series in Engineering and Computer Science*.

Aygün, E. and Ecer, D. (2017). python-alignment. `https://github.com/eseraygun/python-alignment`.

Banzhaf, W. and Bakurov, I. (2024). On The Nature Of The Phenotype In Tree Genetic Programming. *arXiv*, 2402.08011.

Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. (1998). *Genetic programming: An Introduction — On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann Publishers Inc.

Brameier, M., Banzhaf, W., and Banzhaf, W. (2007). *Linear Genetic Programming*, pages 36–37. Springer.

Cai, X., Smith, S. L., and Tyrrell, A. M. (2006). Positional Independence and Recombination in Cartesian Genetic Programming. In *European Conference on Genetic Programming*, pages 351–360. Springer.

Clegg, J., Walker, J. A., and Miller, J. F. (2007). A New Crossover Technique for Cartesian Genetic Programming. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, pages 1580–1587. ACM Press.

Cui, H., Margraf, A., Heider, M., and Hähner, J. (2023). Towards Understanding Crossover for Cartesian Genetic Programming. In *Proceedings of the 15th International Joint Conference on Computational Intelligence (IJCCI 2023)*, pages 308–314. SCITEPRESS.

da Silva, J. E. and Bernardino, H. S. (2018). Cartesian Genetic Programming with Crossover for Designing Combinational Logic Circuits. In *2018 7th Brazilian Conference on Intelligent Systems (BRACIS)*, pages 145–150. IEEE Press.

Griewank, A. O. (1981). Generalized descent for global optimization. *Journal of Optimization Theory and Applications*, 34:11–39.

Haut, N., Banzhaf, W., and Punch, B. (2023). Correlation Versus RMSE Loss Functions in Symbolic Regression Tasks. In *Genetic Programming Theory and Practice XIX*, pages 31–55. Springer.

Hu, T. and Banzhaf, W. (2018). Neutrality, Robustness, and Evolvability in Genetic Programming. In *Genetic Programming Theory and Practice XIV*, pages 101–117. Springer.

Husa, J. and Kalkreuth, R. (2018). 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*, pages 203–219. Springer.

Kalkreuth, R. (2020). A Comprehensive Study on Subgraph Crossover in Cartesian Genetic Programming. In *Proceedings of the 12th International Joint Conference on Computational Intelligence (IJCCI 2020)*, pages 59–70.

Kalkreuth, R. (2021). *Reconsideration and extension of Cartesian genetic programming*. PhD thesis, Technical University of Dortmund, Germany.

Kalkreuth, R., Rudolph, G., and Krone, J. (2015). Improving Convergence in Cartesian Genetic Programming using Adaptive Crossover, Mutation and Selection. In *2015 IEEE Symposium Series on Computational Intelligence*, pages 1415–1422. IEEE Press.

Koza, J. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.

Koza, J. R. (1994a). Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4:87–112.

Koza, J. R. (1994b). *Genetic Programming II: Automatic Discovery of Reusable programs*. MIT Press.

Miller, J. F. et al. (1999). 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*, pages 1135–1142. Morgan Kaufmann.

Miller, J. F. and Harding, S. L. (2008). Cartesian Genetic Programming. In *Proceedings of the 10th Annual Conference Companion on Genetic and Evolutionary Computation*, pages 2701–2726. ACM Press.

Ni, J., Drieberg, R. H., and Rockett, P. I. (2012). The Use of an Analytic Quotient Operator in Genetic Programming. *IEEE Transactions on Evolutionary Computation*, 17(1):146–152.

Oltean, M., Groşan, C., and Oltean, M. (2004). Encoding Multiple Solutions in a Linear Genetic Programming Chromosome. In *International Conference on Computational Science*, pages 1281–1288. Springer.

Real, E., Liang, C., So, D., and Le, Q. (2020). Automl-zero: Evolving machine learning algorithms from scratch. In *International Conference on Machine Learning (ICML-2020)*, pages 8007–8019. MLR Press.

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/isal2024/36/20/2461083/isal_a_00735.pdf by guest on 17 January 2025

Sotto, L. F. D. P., Rothlauf, F., de Melo, V. V., and Basgalupp, M. P. (2022). An Analysis of the Influence of Noneffective Instructions in Linear Genetic Programming. *Evolutionary Computation*, 30(1):51–74.

Torabi, A., Sharifi, A., and Teshnehlab, M. (2023). Using Cartesian Genetic Programming Approach with New Crossover Technique to Design Convolutional Neural Networks. *Neural Processing Letters*, 55(5):5451–5471.

Uy, N. Q., Hoai, N. X., O'Neill, M., McKay, R. I., and Galván-López, E. (2011). Semantically-Based Crossover in Genetic Programming: Application to Real-Valued Symbolic Regression. *Genetic Programming and Evolvable Machines*, 12:91–119.

Wilson, G. and Banzhaf, W. (2008). 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*, pages 182–193. Springer.

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/isal2024/36/20/2461083/isal_a_00735.pdf by guest on 17 January 2025