A Comparative Study on Subgraph Crossover in Cartesian Genetic Programming

Roman Kalkreuth

Department of Computer Science
TU Dortmund University, Germany
Email: roman.kalkreuth@tu-dortmund.de

1 Introduction

While tree-based Genetic Programming (GP) [1] is often used with crossover, Cartesian Genetic Programming (CGP) [2] is mostly used only with mutation as the sole genetic operator. In contrast to comprehensive and fundamental knowledge about crossover in tree-based GP, the state of knowledge in CGP appears to be still ambiguous and ambivalent. Two decades after CGP was officially introduced, the role of recombination in CGP is still considered to be an open and remaining question. The state of knowledge about crossover in CGP has been recently surveyed and the role of crossover is still considered to be an open and remaining question [3]. Even if some progress has been made in recent years, comprehensive and detailed knowledge about crossover in CGP is still missing [3]. A promising step forward was made by the introduction of the subgraph crossover [4] but this technique has not been comprehensively studied in the past. Therefore, this work follows up former work on the crosover question by investigating if the search performance of CGP algorithms that utilize the subgraph crossover can be more efficient as the commonly used mutation-only CGP on a set of well-known benchmark problems. This short paper provides an overview of the full paper version [5] of the presented work.

DOI: 10.58895/ksp/1000124139-16 erschienen in:
Proceedings – 30. Workshop Computational Intelligence: Berlin, 26. - 27. November 2020
DOI: 10.58895/ksp/1000124139 | https://www.ksp.kit.edu/site/books/m/10.58895/ksp/1000124139/

1.1 Cartesian Genetic Programming

In contrast to tree-based GP, CGP represents a genetic program via genotype-phenotype mapping as an indexed, acyclic, and directed graph. The CGP decoding procedure processes groups of genes and each group refers to a function node of the graph. The last genes of the genotype represent the outputs of the phenotype. Each node is represented by two types of genes which index the function number in the GP function set and the node inputs. These nodes are called *function nodes* and execute functions on the input values. The number of input genes depends on the maximum arity n_a of the function set. Given the number of outputs n_o, the n_o last genes in the genotype represent the indices of the nodes, which lead to the outputs. A backward search is used to decode the corresponding phenotype. An example of the backward search of the most popular one-row integer representation is shown in Figure 1. The backward search starts from the program output and processes all nodes which are linked in the genotype. In this way, only active nodes are processed during evaluation. The genotype in Figure 1 is grouped by the function nodes. The first (underlined) gene of each group refers to the function number in the corresponding function set in the figure. The integer-based representation of CGP phenotypes is mostly used with mutation only. The number of inputs n_i, outputs n_o, and the length of the genotype is fixed. Every candidate program is represented with $n_r * n_c * (n_a + 1) + n_o$ integers. CGP is traditionally used with a (1+λ) selection scheme of evolutionary algorithms. The new population in each generation consists of the best individual of the previous population and the λ created offspring. The breeding procedure is mostly done by a point mutation that swaps genes in the genotype of an individual in the valid range by chance.

2 The Subgraph Crossover Technique

The subgraph crossover technique for CGP is inspired by the subtree crossover found in tree-based GP. To recombine two directed acyclic graphs, the subgraph recombination is performed by respecting the CGP phenotype. The phenotype of each individual is represented by the active path of the graph

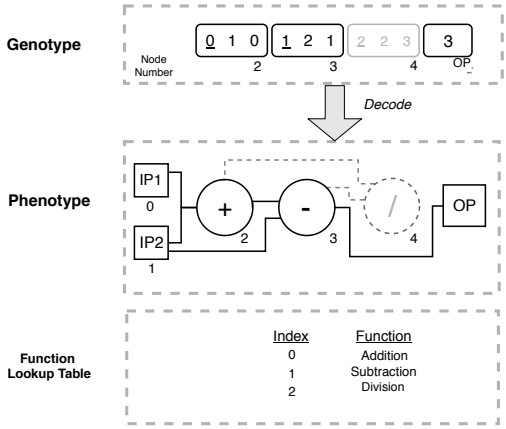

Figure 1: Example of the decoding procedure of a CGP genotype to its corresponding phenotype. The nodes are represented by two types of numbers which index the number in the function lookup table (underlined) and the inputs (non-underlined) for the node. Inactive function nodes are shown in gray color. The identifiers IP1 and IP2 stand for the two input nodes with node index 0 and 1. The identifier OP stands for the output node of the graph.

and is determined through the evaluation process. Furthermore, the active path of a graph leads to the semantic value of a certain individual in CGP. As a consequence, the subgraph crossover exclusively recombines the genetic material of the active paths. The idea of the subgraph crossover is that it should reduce the disruption which is caused by the genotypic single-point crossover in standard CGP and truly recombine subgraphs.

For the description of the subgraph crossover procedure, let n_i be the predefined number of input nodes and let n_f be the predefined number of function nodes. In CGP, the input nodes are indexed from n_i to $n_\mathrm{i}-1$ and the function nodes of each graph are indexed from 0 to $n_\mathrm{i}+n_\mathrm{f}-1$. The nodes which lie between the input and output nodes are denoted as function nodes. The crossover is done with two parents which are denoted as P_1 and P_2. For the crossover procedure, the node numbers of the active function nodes are necessary. The node numbers of the active nodes of P_1 and P_2 are stored in two arrays M_1 and M_2. The active nodes are determined by the backward search in the evaluation procedure.

To define one suitable crossover point, we define two possible crossover points $C_{\mathrm{P}1}$ and $C_{\mathrm{P}2}$ of the two parents. With information about the active nodes and the

length of the path, we can choose two possible crossover points. The possible crossover points C_{P1} and C_{P2} are chosen by chance in the range of the active function nodes which are stored in M_1 and M_2. The possible crossover points may not be input or output nodes. A general crossover point C_P is defined by choosing the smaller crossover point from C_{P1} and C_{P2}. The reason for this is that the subgraphs of the parents, which will be placed in front of or behind the crossover point of the offspring's genome should be balanced. The representation of CGP allows active paths of an individual, which can start in the middle or back of the graph. The subgraph which will be placed in front of the crossover point has to start at more leading active nodes. If C_P is defined as the possible point C_{P1}, the subgraph of P_1 in front of C_P will be placed in front of C_P in the offspring genome. The subgraph behind C_P of P_2 will be placed behind C_P in the offspring genome The crossover procedure produces a new genome that represents the offspring involving the phenotypes of both parents. In the case that two children should be produced, the crossover procedure is performed twice with two different general crossover points. Since the representation of CGP provides connections to any of the previous function nodes of the graph, performing only the *neighbourhood connect* could result in a monotone data flow of the resulting phenotype. An example of the crossover procedure is illustrated in Figure 2.

3 Experiments and Findings

We performed experiments in the problem domain of symbolic regression and Boolean function learning. To evaluate the search performance of the tested algorithms, we measured the number of fitness evaluations until the CGP algorithm terminated successfully (*fitness-evaluations-to-success*) and the best fitness value which was found after a predefined number of generations (*best-fitness-of-run*). We investigated a diverse set of popular GP benchmarks, including single and multiple output problems. The problems are listed in Table 2 and 3. We used a minimizing fitness function in all experiments. For the symbolic regression problems, the fitness of the individuals was represented by a cost function value. The cost function was defined by the sum of the absolute

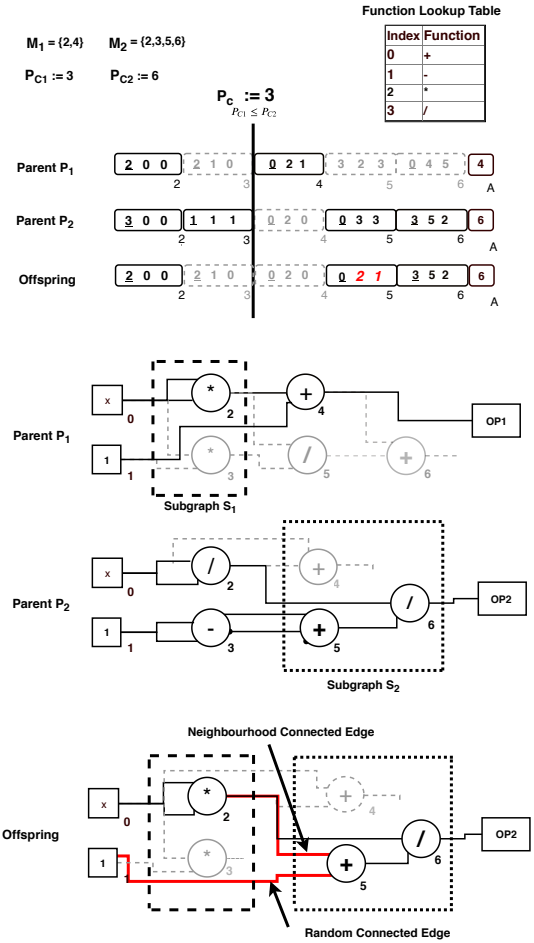

Figure 2: Example of the subgraph crossover technique. The subgraph crossover basically works similar to the single-point crossover except that the active nodes on both sides of the crossover point are preserved. The crossover point is chosen in a way that it is located between active function nodes. At the top of the figure, the arrays with the active nodes and crossover points are listed. Below this information, the genotypes and phenotypes of the parents and the offspring are shown, and the parts of the crossover are marked with dashed boxes.

difference between the real function values and the values of an evaluated individual. For the boolean parity even problems, the fitness was represented by the number of fitness cases for which the candidate solution failed to generate the correct value of the Even-Parity function. To evaluate the fitness on the multiple output problems, we defined the fitness value of an individual as the number of different bits to the corresponding truth table.
In addition to the mean values of the measurements, we calculated the standard deviation (SD) and the standard error of the mean (SEM). The algorithms which were used in our study are listed in Table 1. The best parameter configuration for each algorithm and problem has been determined with the help of meta-evolution. To classify the significance of our results, we used the Mann-Whitney-U-Test. The mean values are denoted $a^{\dagger}$ if the p-value is less than the significance level 0.05 and $a^{\ddagger}$ if the p-value is less than the significance level 0.01 compared to the $(1+4)$-CGP. Note that the mean values are **only** denoted with the significance level marker if the result of a certain algorithm is better than the result of the $(1+4)$-CGP. We performed 100 independent runs with different random seeds.
Table 4 and Table 5 show the results of the algorithm comparison in the Boolean domain. As visible, the Canonical-CGP and the $(\mu+\lambda)$-CGP perform better than the mutation-only CGP algorithms on various problems. The results of our experiments in the symbolic regression domain are shown in Table 6, Table 7 and Table 8. It is visible that the Canonical-CGP algorithm performs better than the mutation-only CGP algorithms on all tested problems.
The experiments demonstrate that the subgraph crossover can contribute to the search performance by using a canonical GA or $(\mu+\lambda)$-strategy. Moreover, the results of our experiments indicate that the predominance of the $(1+4)$-CGP and $(1+\lambda)$-CGP algorithms cannot be generalized in the Boolean domain. The experiments in the symbolic regression domain indicate that the use of the subgraph crossover is beneficial for the use of CGP and can contribute significantly to the search performance in this problem domain. Especially the search performance of the Canonical-CGP algorithm was superior to the $(1+4)$-CGP on all tested problems in the symbolic regression domain.

Table 1: List of the CGP algorithms

Identifier	Description
$(1+4)$-CGP	Traditional $(1+4)$-CGP algorithm
$(1+\lambda)$-CGP	Traditional $(1+\lambda)$-CGP algorithm
$(\mu+\lambda)$-CGP	$(\mu+\lambda)$-algorithm with subgraph crossover
Canonical-CGP	Canonical genetic algorithm (GA) with tournament selection and subgraph crossover

Table 2: Symbolic regression problems

Problem	Objective Function	Vars
Koza-1	$x^4+x^3+x^2+x$	1
Koza-2	x^5-2x^3+x	1
Koza-3	$x^6-2x^4+x^2$	1
Nguyen-4	$x^6+x^5+x^4+x^3+x^2+x$	1
Nguyen-5	$\sin(x^2)\cos(x)-1$	1
Nguyen-6	$\sin(x)+\sin(x+x^2)$	1
Nguyen-7	$\ln(x+1)+\ln(x^2+1)$	1
Keijzer-6	$\sum_i^x 1/i$	1
Pagie-1	$1/(1+x^{-4})+1/(1+y^{-4})$	2

Table 3: Boolean function problems

Problem	Number of Inputs	Number of Outputs
Parity-Even 3	3	1
Parity-Even 4	4	1
Parity-Even 5	5	1
Parity-Even 6	6	1
Parity-Even 7	7	1
Adder 1-Bit	3	2
Adder 2-Bit	5	3
Subtractor 2-Bit	4	3
Multiplier 2-Bit	4	4

Table 4: Results for the Boolean single-output problems evaluated by the number of fitness evaluations (FE) to termination

Problem	Algorithm	Mean FE	SD	SEM	1Q	Median	3Q
Parity-Even-3	$(1+4)$-CGP	3177	3417	±343	1246	2136	3760
	$(1+\lambda)$-CGP	2495	2919	±293	846	1534	2872
	Canonical-CGP	3107	3070	±307	1201	2104	3907
	$(\mu+\lambda)$-CGP	**1565**‡	**1517**	**±152**	**602**	**1168**	**1892**
Parity-Even-4	$(1+4)$-CGP	15420	14152	±1422	6292	10358	17726
	$(1+\lambda)$-CGP	16523	19168	±1926	6095	11276	18557
	Canonical-CGP	54967	47042	±4727	24813	40612	71851
	$(\mu+\lambda)$-CGP	**11135**‡	**8447**	**±845**	**5117**	**8527**	**14085**
Parity-Even-5	$(1+4)$-CGP	45542	33947	±3411	21524	36834	61222
	$(1+\lambda)$-CGP	**34375**‡	**28146**	**±2828**	**20685**	**27104**	**38941**
	Canonical-CGP	**28413**‡	**25538**	**±2566**	**23388**	**19640**	**34876**
	$(\mu+\lambda)$-CGP	43476	2055	±1022	23814	36188	57182
Parity-Even-6	$(1+4)$-CGP	199989	142915	±14291	107418	163234	242573
	$(1+\lambda)$-CGP	**118768**‡	**73682**	**±7368**	**65766**	**91577**	**156639**
	Canonical-CGP	242986	161762	±16257	134518	200196	309346
	$(\mu+\lambda)$-CGP	**110158**‡	**75163**	**±7516**	**63908**	**90676**	**135148**
Parity-Even-7	$(1+4)$-CGP	478055	301113	±30111	268210	393362	605372
	$(1+\lambda)$-CGP	441857	328539	±32853	226272	352254	545197
	Canonical-CGP	631568	548180	±54818	293613	453204	750792
	$(\mu+\lambda)$-CGP	**358420**‡	**246131**	**±24613**	**189278**	**303988**	**451667**

Table 5: Results for the Boolean multi-output problems evaluated by the number of fitness evaluations (FE) to termination

Problem	Algorithm	Mean FE	SD	SEM	1Q	Median	3Q
Adder-1Bit	$(1+4)$-CGP	1895	1856	$\pm$186	634	1252	2494
	$(1+\lambda)$-CGP	1415	1532	$\pm$154	508	1057	1640
	Canonical-CGP	2155	2018	$\pm$202	882	1521	2907
	$(\mu+\lambda)$-CGP	**1393**†	**1311**	**$\pm$132**	**496**	**954**	**1784**
Adder-2Bit	$(1+4)$-CGP	85667	84355	$\pm$8478	29506	58650	110794
	$(1+\lambda)$-CGP	73417	58589	$\pm$5888	33367	53654	94009
	Canonical-CGP	225652	200384	$\pm$20038	78247	158130	271717
	$(\mu+\lambda)$-CGP	68375	43229	$\pm$5361	32998	64006	98052
Multiplier-2Bit	$(1+4)$-CGP	11583	10469	$\pm$1046	5020	8524	14498
	$(1+\lambda)$-CGP	17664	21664	$\pm$2177	5400	9233	19262
	Canonical-CGP	30489	25700	$\pm$2582	13384	22696	22916
	$(\mu+\lambda)$-CGP	**11055**‡	**13281**	**$\pm$1334**	**3635**	**6693**	**13220**
Subtractor-2Bit	$(1+4)$-CGP	11029	13975	$\pm$13975	4642	6986	11878
	$(1+\lambda)$-CGP	**8377**‡	**9958**	**$\pm$1000**	**2989**	**6111**	**9579**
	Canonical-CGP	35829	41822	$\pm$4203	10346	20056	40698
	$(\mu+\lambda)$-CGP	15161	22388	$\pm$2250	5291	9671	16705

Table 6: Results for the algorithm comparison for the problems Koza 1, 2 & 3 evaluated by the number of fitness evaluations (FE) to termination

Problem	Algorithm	Mean FE	SD	SEM	1Q	Median	3Q
Koza-1	$(1+4)$-CGP	8675635	16681422	$\pm$1668142	441477	1814344	7045961
	$(1+\lambda)$-CGP	7370880‡	17384354	$\pm$1738435	204400	1050936	4294170
	Canonical-CGP	**663822**‡	**838546**	**$\pm$83854**	**135162**	**337950**	**710275**
	$(\mu+\lambda)$-CGP	7780751‡	15830735	$\pm$1583073	197284	1830312	6318740
Koza-2	$(1+4)$-CGP	8264426	19894512	$\pm$1989451	150140	888884	4378756
	$(1+\lambda)$-CGP	8191549	20275790	$\pm$2027579	94290	559028	4710848
	Canonical-CGP	**444118**‡	**95000**	**$\pm$286700**	**627550**	**29650**	**78800**
	$(\mu+\lambda)$-CGP	5729778	11021660	$\pm$1102166,	238156	1320880	5878696
Koza-3	$(1+4)$-CGP	600153	1214527	$\pm$121452	39076	177418	443038
	$(1+\lambda)$-CGP	753551	2535215	$\pm$253521	29528	120368	431318
	Canonical-CGP	**32870**‡	**57156**	**$\pm$10435**	**2488**	**6700**	**32713**
	$(\mu+\lambda)$-CGP	926857	3473467	$\pm$347347	28548	121040	362180

Table 7: Results of the algorithm comparison algorithm for the symbolic regression problems evaluated with the *best-fitness-of-run* method

Problem	Algorithm	Mean Best Fitness	SD	SEM	1Q	Median	3Q
Nguyen-4	$(1+4)$-CGP	0,68	0,55	±0,05	0,34	0,58	0,77
	$(1+\lambda)$-CGP	0,61	0,46	±0,04	0,35	0,54	0,74
	Canonical-CGP	**0,50**†	**0,28**	**±0,04**	**0,31**	**0,47**	**0,60**
	$(\mu+\lambda)$-CGP	**0,60**†	**0,40**	**±0,04**	**0,36**	**0,54**	**0,76**
Nguyen-5	$(1+4)$-CGP	0,45	0,42	±0,04	0,06	0,32	0,81
	$(1+\lambda)$-CGP	0,39	0,33	±0,03	0,08	0,27	0,63
	Canonical-CGP	**0,29**‡	**0,27**	**±0,03**	**0,05**	**0,20**	**0,40**
	$(\mu+\lambda)$-CGP	**0,28**‡	**0,25**	**±0,02**	**0,06**	**0,19**	**0,45**
Nguyen-6	$(1+4)$-CGP	0,54	0,66	±0,06	0,16	0,29	0,61
	$(1+\lambda)$-CGP	0,50	0,67	±0,06	0,15	0,22	0,50
	Canonical-CGP	**0,31**‡	**0,31**	**±0,03**	**0,15**	**0,24**	**0,40**
	$(\mu+\lambda)$-CGP	0,61	0,67	±0,06	0,16	0,35	0,67
Nguyen-7	$(1+4)$-CGP	0,79	0,48	±0,05	0,45	0,67	1,06
	$(1+\lambda)$-CGP	0,71	0,45	±0,04	0,44	0,67	0,76
	Canonical-CGP	**0,60**‡	**0,35**	**±0,03**	**0,36**	**0,60**	**0,68**
	$(\mu+\lambda)$-CGP	**0,62**‡	**0,40**	**±0,04**	**0,42**	**0,63**	**0,68**

Table 8: Results of the algorithm comparison algorithm for the symbolic regression problems evaluated with the *best-fitness-of-run* method

Problem	Algorithm	Mean Best Fitness	SD	SEM	1Q	Median	3Q
Keijzer-6	$(1+4)$-CGP	$3,78$	$2,61$	$\pm 0,26$	$2,16$	$3,24$	$4,59$
	$(1+\lambda)$-CGP	$3,38$	$2,52$	$\pm 0,25$	$2,41$	$3,03$	$3,158$
	Canonical-CGP	$\mathbf{2,81}^{\dagger}$	$\mathbf{1,13}$	$\mathbf{\pm 0,11}$	$\mathbf{1,78}$	$\mathbf{2,90}$	$\mathbf{3,75}$
	$(\mu+\lambda)$-CGP	$\mathbf{2,88}^{\dagger}$	$\mathbf{1,09}$	$\mathbf{\pm 0,1}$	$\mathbf{2,25}$	$\mathbf{3,14}$	$\mathbf{3,15}$
Pagie-1	$(1+4)$-CGP	$128,18$	$48,19$	$\pm 4,81$	$87,81$	$119,09$	$161,08$
	$(1+\lambda)$-CGP	$120,75$	$44,95$	$\pm 4,49$	$86,14$	$120,91$	$155,06$
	Canonical-CGP	$\mathbf{98,52}^{\ddagger}$	$\mathbf{50,57}$	$\mathbf{\pm 5,08}$	$\mathbf{59,04}$	$\mathbf{85,31}$	$\mathbf{130,04}$
	$(\mu+\lambda)$-CGP	$\mathbf{99,74}^{\ddagger}$	$\mathbf{41,246}$	$\mathbf{\pm 4,12}$	$\mathbf{65,32}$	$\mathbf{95,79}$	$\mathbf{131,76}$

References

[1] J. Koza. Genetic Programming: A paradigm for genetically breeding populations of computer programs to solve problems. Technical Report STAN-CS-90-1314, Dept. of Computer Science, Stanford University, June 1990.

[2] 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*, Orlando, Florida, USA, 1999.

[3] Julian Francis Miller. Cartesian genetic programming: its status and future. *Genetic Programming and Evolvable Machines*, 21(1):129–168, 2020.

[4] Roman Kalkreuth, Guenter Rudolph, and Andre Droschinsky. A new subgraph crossover for cartesian genetic programming. In *EuroGP 2017: Proceedings of the 20th European Conference on Genetic Programming*, volume 10196 of *LNCS*, pages 294–310, Amsterdam, 19-21 April 2017. Springer Verlag.

[5] Roman Kalkreuth. A comprehensive study on subgraph crossover in cartesian genetic programming. In *Proceedings of the 12th International Joint Conference on Computational Intelligence, IJCCI 2020, November 02-04, 2020*. ScitePress, 2020.