On the Trade-Off between Population Size and Number of Generations in GP for Program Synthesis

Martin Briesch
Johannes Gutenberg University
Mainz, Germany
briesch@uni-mainz.de

Dominik Sobania
Johannes Gutenberg University
Mainz, Germany
dsobania@uni-mainz.de

Franz Rothlauf
Johannes Gutenberg University
Mainz, Germany
rothlauf@uni-mainz.de

ABSTRACT

When using genetic programming for program synthesis, we are usually constrained by a computational budget measured in program executions during evolution. The computational budget is influenced by the choice of population size and number of generations per run leading to a trade-off between both possibilities. To better understand this trade-off, we analyze the effects of different combinations of population sizes and number of generations on performance. Further, we analyze how the use of different variation operators affects this trade-off. We conduct experiments on a range of common program synthesis benchmarks and find that using larger population sizes lead to a better search performance. Additionally, we find that using high probabilities for crossover and mutation lead to higher success rates. Focusing on only crossover or using only mutation usually leads to lower search performance. In summary, we find that large populations combined with high mutation and crossover rates yield highest GP performance for program synthesis approaches.

CCS CONCEPTS

• **Software and its engineering → Genetic programming**.

KEYWORDS

Genetic Programming, Program Synthesis, Generations, Population Size, Crossover, Mutation

ACM Reference Format:
Martin Briesch, Dominik Sobania, and Franz Rothlauf. 2023. On the Trade-Off between Population Size and Number of Generations in GP for Program Synthesis. In *Genetic and Evolutionary Computation Conference Companion (GECCO '23 Companion), July 15–19, 2023, Lisbon, Portugal.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3583133.3590681

1 INTRODUCTION

Program synthesis deals with the automatic construction of source code for a programming task. Usually, program synthesis problems are specified by using either natural language or input/output pairs [18]. When working with input/output pairs, a well-known and successful method is genetic programming (GP) [12]. Common GP approaches used in program synthesis task are grammar-guided GP [5, 6, 21], stack-based GP [20], and linear GP [13].

GP runs for program synthesis problems are usually expensive and thus constrained by a computational budget. This budget is measured in program executions used for evolution; GP runs are stopped when a solution passes all training cases or the computational budget is exhausted. The computational effort depends on the size of the training set, the chosen population size, and the maximum number of generations. While the size of the training set is mostly given, population size and number of generations can be chosen. Given a fixed budget, we have a trade-off between population size and number of generations. While there is some work on GP in general [14, 15], there is only limited work on the trade-off between population size and number of generations for the program synthesis domain [4, 10]. This is especially important when allocating additional program executions gained from down-sampled lexicase selection techniques [1, 11].

Therefore, in this work we study how allocating the computational budget to different combinations of population size and number of generations affects the performance of grammar-guided GP for program synthesis problems. Furthermore, we investigate the influence of the variation operators crossover and mutation on the trade-off between population size and running time.

We perform experiments for six common benchmark problems from the program synthesis literature [7, 9] with different combinations of population size and number of generations while keeping program executions constant. We compare population sizes ranging from 250 to 4000 and corresponding numbers of generations ranging from 1200 to 75. Additionally, we investigate different combinations of variation operators. In particular the 1) classical approach with high crossover and low mutation probability, 2) pure mutation without crossover, and 3) high crossover probability combined with high mutation probability.

We find that for most problems, larger population sizes with fewer number of generations yield higher success rates. Furthermore, in most cases crossover has a positive influence on search; the influence of mutation depends on the problem. Combining crossover with a high mutation rate yields the best results on most of the studied problems.

Section 2 details on our experiments, followed by our results in Sect. 3. Section 4 wraps up the paper with a conclusion and future work.

2 METHODOLOGY

We present the benchmark problems used in the experiments, describe the experimental details, and explain the used metrics.

2.1 Benchmark Problems

We use six different program synthesis problems from common benchmark suites for our experiments [7, 9]. The problems are of varying difficulty and are well studied in literature [17, 18]. Additionally they consist of different input and output types and require different program paradigms to be solved. All problems are specified using input/output pairs as training and test cases. The training and test set for each problem consists of 200 training and 1,000 test cases.

We study the following problems: `Count Odds`, `Fizz Buzz`, `Fuel Cost`, `Grade`, `Scrabble Score`, and `Small or Large`.

2.2 Experimental Setup

Our experiments are conducted using a grammar-guided GP approach [5, 6, 16, 21], using context-free grammars in Backus-Naur form and a tree-based representation. For the implementation we use the PonyGE2 framework [3].

All populations are initialized with position-independent grow [2] with a maximum initial tree depth of 10. Selection is performed using lexicase [19]. We use subtree crossover and subtree mutation with one mutation event as variation operators. An elite size of 5 is used and the maximum tree depth is set to 17 [12].

To study the influence of different allocation of the computational budget, we set the budget to 60,000,000 program executions (with 200 training cases we obtain 300,000 evaluated individuals) as suggested in [7] and vary the population size N and number of generations G per run. We use the following five combinations of population size and number of generations: `[250,1200]`, `[500,600]`, `[1000,300]`, `[2000,150]`, `[4000,75]`.

To examine how different configurations of the variation operators affect the trade-off between population size and generations, we consider three different settings of crossover probability p_c and mutation probability p_m: High crossover probability in combination with high mutation probability (`[0.95,1]`), no crossover and only mutation as commonly used in other evolutionary systems (`[0,1]`) [8, 20], and the classical setting of high crossover and low mutation probability (`[0.95,0.05]`).

For each parameter configuration and program synthesis problem, we perform 100 independent runs. A run is terminated when an individual is found that passes all training cases.

2.3 Metrics

As is common in GP-based program synthesis, we use success rate as the main metric for evaluating GP performance. A GP run is stopped as soon as it finds a solution that solves all training cases. A run is considered to be successful, if this solution passes all (previously unseen) test cases. The success rate is defined as the number of successful runs out of the 100 runs we perform per configuration and problem (percentage of successful runs). Higher success rates are considered to be better.

3 RESULTS

We report the results of our experiments. First, we describe the observed effects of different combinations of population size and number of generations. Second, we study how different settings of variation operators influence these effects.

3.1 Population Size vs. Number of Generations

We analyze how the success rates depend on the trade-off between population size and number of generations. Figure 1 plots the number of successful runs over the combination of population size and number of generations for 100 runs. We use different colors for the variation operator settings and plot a linear regression line over the different popsize/generations combinations to better visualize the trend.

We find that, in general, the combination of large population and low number of generations leads to more successful runs. For example for the `Count Odds` problem, all configurations of search operators perform better if we allocate more computational budget towards larger populations. For the `Small or Large` problem and the high crossover/high mutation setting, we see an increase from 26 to 59 successful runs, more than doubling the success rate. Only for `Scrabble Score` (Fig. 1e) we observe a slight downward trend.

Table 1 presents the success rates (given 100 runs, the success rate is the number of successful runs) and the results of the statistical tests. The highest success rate per problem and variation operator setting (indicated by the probability p_c of crossover and mutation p_m) is displayed in **bold** font; the highest success rate per problem across all variation operator settings is indicated with an underline. We also performed statistical pairwise tests per problem between the combinations of population size N and number of generations G and different settings of variation operator probabilities p_c and p_m. All test where performed using a two-sided proportions z-test and corrected for multiple comparisons with a Bonferroni-Holm correction. A small letter in subscript indicates a significant difference to the corresponding combination of popsize/generations. A number in superscript marks a significant difference to the corresponding variation operator setting. All tests are performed at a significance level of $\alpha = 0.05$.

We find that in 5 out of 6 problems the highest success rate is achieved using larger populations and lower number of generations. Only for the `Scrabble Score` problem, we observe higher success rates with smaller populations.

We find significant differences between allocating the computational budget either towards larger populations or towards higher number of generations. In the `Count Odds`, `Fuel Cost`, `Grade`, and `Small or Large` problem, at least one variation operator setting is significantly better with large populations in contrast to higher number of generations. For larger number of generations, a significantly higher performance is only observed on the `Scrabble Score` problem.

In summary, the results (see Table 1) suggest that larger populations lead to significantly higher success rates. In addition, the results show difference between variation operator settings. Therefore, the next section takes a closer look at the role of the variation operators.

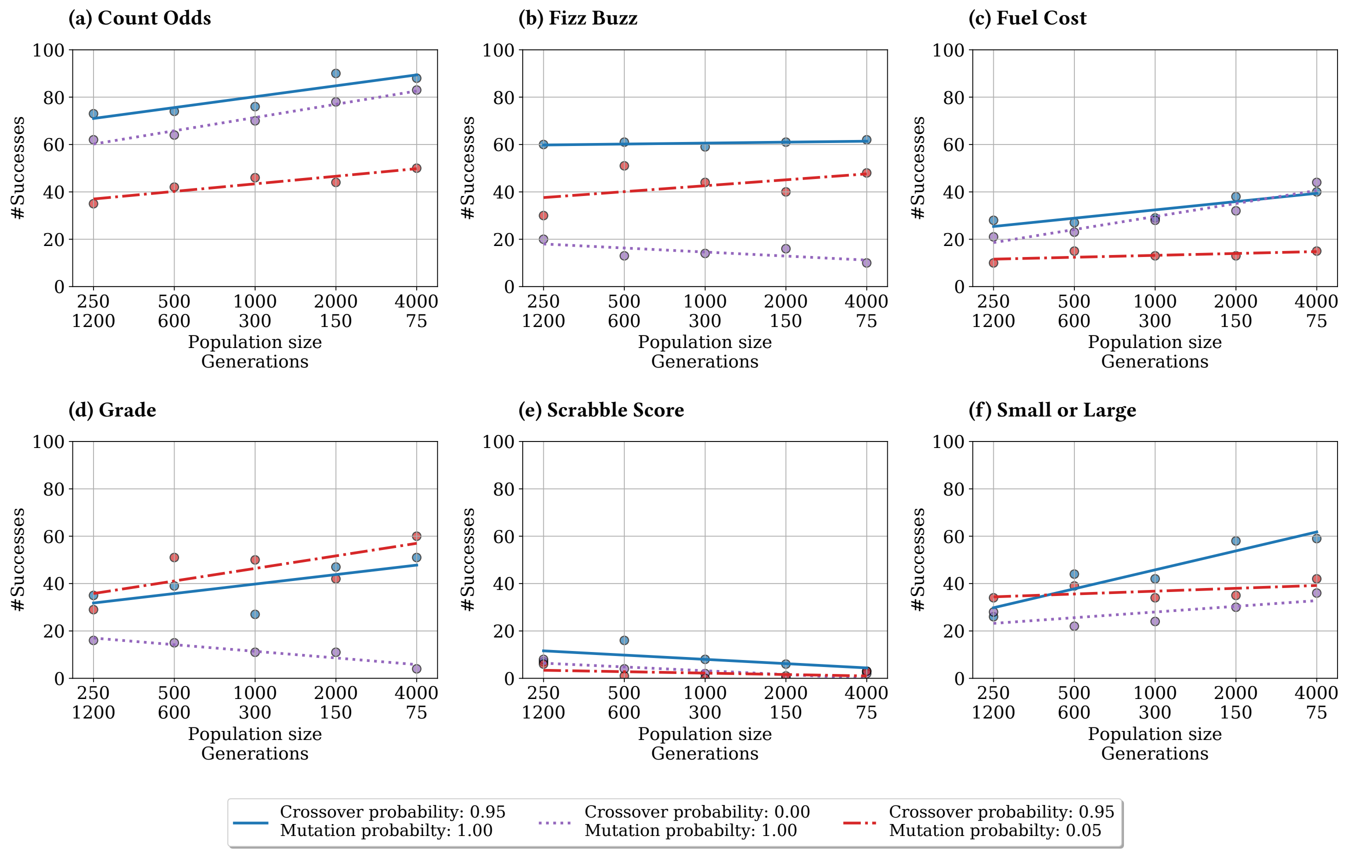

Figure 1: Number of successful runs over combinations of popsize/generations. Results are for different variation operator settings and test problems. The trend per variation operator setting is indicated by a linear regression line.

3.2 Influence of Crossover and Mutation

Figure 1 shows that the classical variation operator setting (high p_c, low p_m) outperforms pure mutation without crossover in three out of six problems (`Fizz Buzz`, `Grade`, and `Small or Large`). Pure mutation, on the other hand, performs better than the classical setting on the other problems (`Count Odds`, `Fuel Costs`, and `Scrabble Score`). However, both options are outperformed by a combination of high crossover and high mutation probability on five out of six problems. Only for the `Grade` problem, the classical setting performs slightly better. These differences are also confirmed by our statistical testing (see Table 1). Therefore, the combination of high p_c and high p_m yields best results.

4 CONCLUSIONS AND FUTURE WORK

In this work we studied the trade-off between population size and number of generations when working with a constrained budget of program executions in GP for program synthesis. Additionally, we analyzed the effect of different combinations of crossover and mutation. Further, we studied the relationship between using either recombination or mutation as main search operator.

We performed a set of experiments on standard and common program synthesis benchmark problems and found that overall, larger population sizes lead to a better search performance. Furthermore, we found that search performance is highest if both crossover and mutation probability is high. Using only either mutation or crossover usually leads to lower search performance.

Therefore, we encourage researchers that are faced with the decision where to allocate the computational budget, to use larger population sizes instead of longer runs with more generations. Furthermore, we recommend to combine crossover and high mutation rates in the program synthesis domain. We expect that even in pure mutation based systems, a crossover operator could significantly improve problem solving performance.

In future work we will investigate if these results can also be extended to other application areas of GP like symbolic regression.

ACKNOWLEDGMENTS

Parts of this research were conducted using the supercomputer Mogon and/or advisory services offered by Johannes Gutenberg University Mainz (hpc.uni-mainz.de), which is a member of the AHRP (Alliance for High Performance Computing in Rhineland Palatinate, www.ahrp.info) and the Gauss Alliance e.V.

Table 1: Success rates for the different problems. Highest success rates per problem and variation operator setting are displayed in bold font; highest success rate overall per problem with an underline. A number in superscript indicates a significant difference to the corresponding variation operator setting (p_c / p_m). A small letter in subscript indicates a significant difference to the corresponding combination of popsize N and number of generations G. All pairwise tests are performed using a two-sided proportions z-test and Bonferroni-Holm correction on a significance level of $\alpha = 0.05$.

	${}^{1}p_c$: **0.95** p_m: **1.00**	${}^{2}p_c$: **0.00** p_m: **1.00**	${}^{3}p_c$: **0.95** p_m: **0.05**
Count Odds			
$_a$ N=250 / G=1200	${}^{3}_{d}$ 73	${}^{3}_{e}$ 62	^{1,2} 35
$_b$ N=500 / G=600	${}^{3}_{d}$ 74	${}^{3}_{e}$ 64	^{1,2} 42
$_c$ N=1000 / G=300	^{3} 76	^{3} 70	^{1,2} 46
$_d$ N=2000 / G=150	${}^{3}_{a,b}$ $\underline{\mathbf{90}}$	^{3} 78	^{1,2} 44
$_e$ N=4000 / G=75	^{3} 88	${}^{3}_{a,b}$ **83**	^{1,2} **50**
Fizz Buzz			
$_a$ N=250 / G=1200	^{2,3} 60	^{1} **20**	${}^{1}_{b}$ 30
$_b$ N=500 / G=600	^{2} 61	^{1,3} 13	${}^{2}_{a}$ **51**
$_c$ N=1000 / G=300	^{2} 59	^{1,3} 14	^{2} 44
$_d$ N=2000 / G=150	^{2,3} 61	^{1,3} 16	^{1,2} 40
$_e$ N=4000 / G=75	^{2} $\underline{\mathbf{62}}$	^{1,3} 10	^{2} 48
Fuel Cost			
$_a$ N=250 / G=1200	^{3} 28	${}_{e}$ 21	^{1} 10
$_b$ N=500 / G=600	27	${}_{e}$ 23	**15**
$_c$ N=1000 / G=300	^{3} 29	^{3} 28	^{1,2} 13
$_d$ N=2000 / G=150	^{3} 38	^{3} 32	^{1,3} 13
$_e$ N=4000 / G=75	^{3} **40**	${}^{3}_{a,b}$ $\underline{\mathbf{44}}$	^{1,2} **15**
Grade			
$_a$ N=250 / G=1200	^{2} 35	${}^{1}_{e}$ **16**	${}_{b,c,e}$ 29
$_b$ N=500 / G=600	^{2} 39	^{1,3} 15	${}^{2}_{a}$ 51
$_c$ N=1000 / G=300	${}^{2,3}_{d,e}$ 27	^{1,3} 11	${}^{1,2}_{a}$ 50
$_d$ N=2000 / G=150	${}^{2}_{c}$ 47	^{1,3} 11	^{2} 42
$_e$ N=4000 / G=75	${}^{2}_{c}$ **51**	${}^{1,3}_{a}$ 4	${}^{2}_{a}$ $\underline{\mathbf{60}}$
Scrabble Score			
$_a$ N=250 / G=1200	7	${}_{d}$ **8**	**6**
$_b$ N=500 / G=600	${}^{2,3}_{e}$ $\underline{\mathbf{16}}$	^{1} 4	^{1} 1
$_c$ N=1000 / G=300	^{3} 8	2	^{1} 0
$_d$ N=2000 / G=150	^{2} 6	${}^{1}_{a}$ 0	1
$_e$ N=4000 / G=75	${}_{b}$ 3	2	3
Small or Large			
$_a$ N=250 / G=1200	${}_{d,e}$ 26	28	34
$_b$ N=500 / G=600	^{2} 44	^{1,3} 22	^{2} 39
$_c$ N=1000 / G=300	^{2} 42	^{1} 24	34
$_d$ N=2000 / G=150	${}^{2,3}_{a}$ 58	^{1} 30	^{1} 35
$_e$ N=4000 / G=75	${}^{2,3}_{a}$ $\underline{\mathbf{59}}$	^{1} **36**	^{1} **42**

REFERENCES

[1] Ryan Boldi, Martin Briesch, Dominik Sobania, Alexander Lalejini, Thomas Helmuth, Franz Rothlauf, Charles Ofria, and Lee Spector. 2023. Informed Down-Sampled Lexicase Selection: Identifying productive training cases for efficient problem solving. *arXiv preprint arXiv:2301.01488* (2023).

[2] David Fagan, Michael Fenton, and Michael O'Neill. 2016. Exploring position independent initialisation in grammatical evolution. In *2016 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 5060–5067.

[3] Michael Fenton, James McDermott, David Fagan, Stefan Forstenlechner, Erik Hemberg, and Michael O'Neill. 2017. Ponyge2: Grammatical evolution in python. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 1194–1201.

[4] Austin J Ferguson, Jose Guadalupe Hernandez, Daniel Junghans, Alexander Lalejini, Emily Dolson, and Charles Ofria. 2020. Characterizing the effects of random subsampling on lexicase selection. *Genetic Programming Theory and Practice XVII* (2020), 1–23.

[5] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2017. A grammar design pattern for arbitrary program synthesis problems in genetic programming. In *Genetic Programming: 20th European Conference, EuroGP 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings 20*. Springer, 262–277.

[6] Stefan Forstenlechner, Miguel Nicolau, David Fagan, and Michael O'Neill. 2016. Grammar design for derivation tree based genetic programming systems. In *Genetic Programming: 19th European Conference, EuroGP 2016, Porto, Portugal, March 30-April 1, 2016, Proceedings 19*. Springer, 199–214.

[7] Thomas Helmuth and Peter Kelly. 2021. PSB2: the second program synthesis benchmark suite. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 785–794.

[8] Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2018. Program synthesis using uniform mutation by addition and deletion. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1127–1134.

[9] Thomas Helmuth and Lee Spector. 2015. General program synthesis benchmark suite. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. 1039–1046.

[10] Thomas Helmuth and Lee Spector. 2022. Problem-Solving Benefits of Down-Sampled Lexicase Selection. *Artificial Life* 27, 3–4 (03 2022), 183–203.

[11] Jose Guadalupe Hernandez, Alexander Lalejini, Emily Dolson, and Charles Ofria. 2019. Random subsampling improves performance in lexicase selection. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 2028–2031.

[12] R John. 1992. Koza. Genetic Programming: On the Programming of Computers by Means of Natural Selection.

[13] Alexander Lalejini and Charles Ofria. 2019. Tag-accessed memory for genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 346–347.

[14] Sean Luke and Lee Spector. 1997. A comparison of crossover and mutation in genetic programming. *Genetic Programming* 97 (1997), 240–248.

[15] Sean Luke and Lee Spector. 1998. A revised comparison of crossover and mutation in genetic programming. *Genetic Programming* 98 (1998), 208–213.

[16] Conor Ryan, John James Collins, and Michael O Neill. 1998. Grammatical evolution: Evolving programs for an arbitrary language. In *Genetic Programming: First European Workshop, EuroGP'98 Paris, France, April 14–15, 1998 Proceedings 1*. Springer, 83–96.

[17] Dominik Sobania, Martin Briesch, and Franz Rothlauf. 2022. Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1019–1027.

[18] Dominik Sobania, Dirk Schweim, and Franz Rothlauf. 2022. A comprehensive survey on program synthesis with evolutionary algorithms. *IEEE Transactions on Evolutionary Computation* (2022).

[19] Lee Spector. 2012. Assessment of problem modality by differential performance of lexicase selection in genetic programming: a preliminary report. In *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*. 401–408.

[20] Lee Spector and Alan Robinson. 2002. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines* 3 (2002), 7–40.

[21] Peter A Whigham et al. 1995. Grammatically-based genetic programming. In *Proceedings of the workshop on genetic programming: from theory to real-world applications*, Vol. 16. Citeseer, 33–41.