Generational Computation Reduction in Informal Counterexample-Driven Genetic Programming

Thomas Helmuth[1[0000−0002−2330−6809]], Edward Pantridge[2[0000−0003−0535−5268]], James Gunder Frazier[1[0009−0008−9549−1485]], and Lee Spector[3,4[0000−0001−5299−4797]]

[1] Hamilton College, Clinton NY 13323, USA {thelmuth,jgfrazie}@hamilton.edu
[2] Real Chemistry, Boston MA 02111, USA ed@swoop.com
[3] Amherst College, Amherst MA 01002, USA lspector@amherst.edu
[4] University of Massachusetts, Amherst MA 01003, USA

Abstract. Counterexample-driven genetic programming (CDGP) uses specifications provided as formal constraints to generate the training cases used to evaluate evolving programs. It has also been extended to combine formal constraints and user-provided training data to solve symbolic regression problems. Here we show how the ideas underlying CDGP can also be applied using only user-provided training data, without formal specifications. We demonstrate the application of this method, called "informal CDGP," to software synthesis problems. Our results show that informal CDGP finds solutions faster (i.e. with fewer program executions) than standard GP. Additionally, we propose two new variants to informal CDGP, and find that one produces significantly more successful runs on about half of the tested problems. Finally, we study whether the addition of counterexample training cases to the training set is useful by comparing informal CDGP to using a static subsample of the training set, and find that the addition of counterexamples significantly improves performance.

Keywords: genetic programming · program synthesis · counterexamples · training data

1 Introduction

The bulk of the computational effort required for genetic programming (GP) is expended in the evaluation of programs in the evolving population. Typically, each program is evaluated on many inputs, which are generally referred to as "fitness cases" or "training cases." In most prior work, all available cases are used to evaluate each program.

Two recent developments in GP have offered new approaches to handling training cases that appear to provide significant advantages. One of these methods uses only a small, random sub-sample of the available cases each generation. This "down-sampling" saves significant computational effort per program evaluation, allowing one to run the evolutionary system for more generations with

the same computational budget, leading to significant improvements in problem-solving power [19,8,15].

A second method, counterexample-driven genetic programming (CDGP), generates training cases using formal specifications that must be provided for the problem to be solved [23,3,24,1,2,32]. In particular, it is able to generate training cases that are not correctly solved by the evolving programs, adding these cases to a growing training set. These "counterexamples" provide more focused guidance to the evolutionary process than do random test cases, and appear to direct evolution more specifically to master aspects of the target problem that are not properly handled by individuals in the current population. While CDGP has been applied to constrained problem domains where it is possible to check whether any given program satisfies the given formal specifications, it is impossible to check whether programs over a Turing-complete language satisfy given formal specifications [27,21]. Therefore, CDGP cannot be applied directly to general program synthesis problems, where GP evolves programs that may include looping or recursion and access to potentially unbounded storage.

In this paper, we describe a novel method that builds on ideas of down-sampling the training data and CDGP by extending the idea of counterexamples to not require formal specifications.[5] The approach that we describe, "informal CDGP" (iCDGP), evaluates individuals during evolution using only a small sub-sample of the user-provided training cases, like down-sampled GP, allowing more individuals to be assessed within the same computational budget. When training cases are added to the training set, they are not chosen randomly, but rather are chosen to be counterexamples for the best individuals in the current population. This allows iCDGP to direct evolution in much the same way as CDGP, but without requiring that the user provide formal specifications for solutions to the target problem.

We test iCDGP on a set of general program synthesis benchmark problems, which require evolving programs in a Turing-complete language [14]. Initial experiments found that many times GP was not able to find a program that passed all training cases, meaning no new cases were added. We develop two new variants of iCDGP, and find that one in particular outperforms standard GP; this variant ensures that new cases are added to the training set throughout evolution, whether or not a program is found that passes the current training set. The second variant limits the size of the training set, motivated by making better use of the computational budget; this variant does not show as much empirical promise.

2 Related Work

This work takes its motivation from and builds on counterexample-driven GP and down-sampled lexicase selection. We describe each of those techniques in detail, and then discuss other related work.

[5] This paper expands on a poster paper that we published in GECCO 2020 [18].

2.1 Counterexample-Driven GP

CDGP uses specifications provided as formal constraints in order to generate the training cases used to evaluate a population of evolving programs [23,3,24]. CDGP was extended to use both formal constraints and user-provided training data to solve symbolic regression problems [1,2]. Additionally, CDGP has been combined with synthesis through unification, which allows it to partially decompose parts of problems into subproblems to solve [32].

CDGP evaluates individuals in the population against both a set of automatically generated training cases and the provided formal constraints. The training set is the primary method of evaluating individuals for parent selection, and the formal constraints are used to generate new training cases when necessary. The CDGP algorithm proceeds as follows: The training set is empty at the start of evolution. Then, each generation, every individual is evaluated on each training case in the training set. If any individual passes all of the training cases[6], CDGP uses a Satisfiability Modulo Theories (SMT) solver to test the individual on the problem's formal constraints. If the program passes the formal constraints, evolution stops because it has found a solution. If the individual fails a constraint, the SMT solver returns a counterexample in the form of a new case that the program does not pass. CDGP adds this case to the training set for the next generation. This process continues until it either finds a solution or reaches a maximum number of generations.

In standard CDGP, a new counterexample case is added to the training set only when a program passes all current training cases. However, one extension of CDGP adds a fitness threshold $q \in [0, 1]$ that specifies the proportion of the training cases that an individual must pass before running the SMT solver on it to produce a new counterexample case [3]. This allows new cases to be added earlier, giving more search gradient for evolution to follow without having to find a program that passes all cases in the current training set. A value of $q = 1.0$ is equivalent to the standard CDGP, since it adds a new case only when an individual passes all current cases. On the basis of experimentation, the developers of this technique recommend a value for q in the range $[0.75, 1]$.

We are unaware of any previous work employing counterexamples without the use of formal specifications, as we do here.

2.2 Down-sampled lexicase selection

While down-sampling of training data has been occasionally used in GP, it has recently been studied in the program synthesis domain when using lexicase parent selection [19,8,15,16]. In this setting, the training set is down-sampled to include a random subset of the training cases each generation. Down-sampled lexicase selection reduces the cost to evaluate each individual, with the same motivation as iCDGP.

[6] Every individual in the first generation passes the training set, since it is initially empty.

Multiple studies have examined why and where down-sampled lexicase performs well [16,28,20]. In evolutionary robotics, down-sampled lexicase selection has been used to limit the costs of robotics simulations [25,26]. More recently, work has been done to make informed decisions when selecting the cases that appear in the subsampled training set [6,4,5].

2.3 Other Related Work

Guiding learning with counterexamples that modify a training set has been recently explored in machine learning [7,29]. Implementations of this technique require an error table to be constructed from the model's misclassified data points from training, which is then used as specifications for how to construct counterexamples to train the model.

Metamorphic testing has been applied to GP to extend the usefulness of each case in exposing undesirable behaviors in candidate solutions without needing to include more cases to train on [30]. To apply metamorphic testing to a problem however, a user must first identify a metamorphic relation a solution program's output must exhibit; these metamorphic relationships are related to but different from the formal specifications required by CDGP.

The core of iCDGP has been used to develop human-driven genetic programming for program synthesis, in which a user is responsible for providing the initial training set and for verifying whether or not generated cases are counterexamples to a potential solution. A prototype system has shown promise on some basic program synthesis problems [9].

3 Informal Counterexample-Driven GP

Informal counterexample-driven GP (iCDGP) borrows motivation from CDGP, but deviates in some significant and novel ways. Specifically, we aim to adapt the core concept of a small training set that grows with added counterexamples. Since we do not have formal specifications, we instead expect the problem to be defined by a full training set of input/output examples, typically numbering 100 or more, which we call T.

In iCDGP, we use an active training set, $T_A \subseteq T$, that GP uses to evaluate the individuals in the population. In all of our experiments, T_A initially contains 10 random training cases from T, although other sizes could be used. During evolution, if an individual is found that passes all of the cases in T_A, we test the individual on all of the cases in $T \setminus T_A$; if it also passes all of them, then it is a training set solution and GP terminates. Otherwise, we select a random case in T that the individual does not pass, add it to T_A, and continue evolution. Note that if multiple individuals in a generation pass all of the cases in T_A, each of them goes through this process, potentially adding multiple new cases to T_A for the next generation.

Given that we already have a set of training cases, why does iCDGP use a smaller, likely less-informative set of active training cases? As with other approaches based on the sub-sampling of training cases (such as down-sampled

lexicase and cohort lexicase selections [19,8]), a smaller active training set allows iCDGP to perform fewer program executions per generation, making each generation computationally cheaper than if using the full set of training cases. In our experiments, we compare methods based on the same maximum number of program executions, allowing iCDGP to run for more generations than standard GP while using the same total program executions. Additionally, the CDGP idea of adding a counterexample case to T_A that the best individual does not pass allows it to augment the training set in ways that specifically direct GP to solve difficult parts of the problem that have not yet been solved by the evolving population.

In our experiments, we test several variants of iCDGP to try to improve it. One such variant uses a fitness threshold q to determine when to add a new counterexample, a variant introduced in formal CDGP [3]. For iCDGP, this triggers the system to test the individual on all of T, and then add a case to T_A that the individual does not pass. It is possible that the individual passes all of the cases in T that are not already in T_A; in this case, T_A does not change.

We designed another variant of iCDGP to address an issue that we discovered when analyzing our results, as presented in Section 5. In particular, many times iCDGP cannot find a program that passes all (or even a sufficiently high percentage to exceed a fitness threshold) of the active training set without passing all training cases. It may still be beneficial to add new training cases to T_A to provide GP with more information to guide search. Thus we created a variant of iCDGP, called generation-based case additions, that adds a new training case to T_A every d generations after the last case was added (whether through this process or an individual passing all of T_A). In order to select a case that provides better information for the search, we evaluate the best individual in the population (i.e. the one that passes the most cases in T_A, with ties broken at random) on all cases in T, and choose a random case that it does not pass.

Finally, we test a variant that sets a maximum number of cases to add to T_A. This variant prevents T_A from getting too large, which may be undesirable since it reduces the number of generations that GP can evaluate before using up the program execution budget. When adding a case that would increase the size of T_A past the given limit, we first remove the case in T_A that is passed by the most individuals in the population. In this way, we can remove cases that provide less useful direction to search while adding cases not passed by the best individuals.

4 Experiment Design

In this study we focus on general program synthesis problems, which require the GP system to generate programs that have similar qualities to the types of programs we expect humans to write. For our experiments we use 12 problems with a range of difficulties selected from the PSB1 benchmark suite [14]. These problems use different data types as inputs and outputs, and many require iteration or recursion and conditional execution to solve.

Table 1. PushGP system parameters.

Parameter	Value
population size	1000
max generations for runs using full training set	300
parent selection	lexicase
genetic operator	UMAD
UMAD addition rate	0.09
initial size of T_A	10

Each program synthesis problem is defined by a training set T of input/output examples, as well as an unseen test set that is used to test for generalization of solutions to unseen data.[7] As discussed above, when a program is found that passes all of the cases in the active training set T_A, it is tested on all cases in T; if it passes those as well, GP terminates. We then automatically simplify the program using a process that shrinks program sizes without changing its behavior on T; this simplification has been shown to increase generalization on the benchmark problems used in this study [10]. The simplified program then undergoes generalization testing on the test set; if it passes all of the unseen test cases, we consider it a successful run. If a program does not pass the test set, or if a run terminates from reaching the execution limit, it is marked a failure. We use a chi-square test with a 0.05 significance level to test for significant differences in success rates.

Our experiments use PushGP, which evolves programs in the Push programming language [31]. Push, designed specifically for use in GP, uses a set of typed stacks to store data manipulated by a program. Push programs are hierarchical lists containing data literals, which are pushed onto stacks when encountered in programs, and instructions, which take their inputs from specifics stacks and return their results to the stacks. We use an implementation of PushGP written in Clojure in our experiments.[8]

The PushGP system parameters used in our experiments are given in Table 1. Individual genomes are stored in the Plushy representation, and translated into Push programs for execution. We use uniform mutation with additions and deletions (UMAD) as our only genetic operator, making all children through mutation only, since this mutation has produced the best known results when using PushGP on these benchmark problems [13]. We use the size-neutral version of UMAD, adding new instructions before or after 9% of instructions in the parent. Additionally, we use lexicase selection for parent selection [14,17].

Lexicase selection has one peculiar characteristic with respect to iCDGP: when an individual is found that passes all cases in T_A but not all those in T, we add a new case to T_A. However, the selection of parents for the next generation

[7] The datasets for these problems are available at https://github.com/thelmuth/program-synthesis-benchmark-datasets.

[8] https://github.com/lspector/Clojush

Table 2. Full training set size and program execution limit for each problem.

Problems	Training Size	Executions
Compare String Lengths, Double Letters, Mirror Image, Replace Space With Newline, Smallest, String Lengths Backwards, Syllables	100	30,000,000
Last Index of Zero, X-Word Lines	150	45,000,000
Negative to Zero, Scrabble Score	200	60,000,000
Vector Average	250	75,000,000

is based on T_A before the new case is added, since that is the set of cases that the population is evaluated on. In lexicase selection, if an individual passes all cases considered for selection and no other individual does, then it will be selected in every single parent selection event that generation. Thus we might expect substantial drops in population diversity each time we add a new case to T_A. We investigate the implications of this interaction between lexicase selection and iCDGP empirically in Section 5.4.

Since iCDGP executes fewer programs per generation, all of our PushGP runs are limited by the number of program executions they allow, equivalent to using a population size of 1000 and 300 maximum generations for a full training set. This ensures that all methods receive the same number amount of computation. Since iCDGP uses fewer training cases to evaluate each individual, it runs for more generations to make up the same number of program executions. The number of training cases in the full training set varies per problem, so the maximum program execution limits also vary per problem, and both are given in Table 2.

5 Results

We first present results comparing iCDGP to GP using the full training set. The first three columns of Table 3 give the number of successful GP runs out of 100 using a full training set, iCDGP, and iCDGP with a fitness threshold of $q = 0.8$. First, note that iCDGP performed a bit worse than using the full training set, including significantly worse on four problems while only significantly better on one. On the other hand, iCDGP using a fitness threshold performed significantly better than the full training set on three problems while only performing significantly worse on two, showing the benefits of adding cases before finding a solution on the training set.

Despite producing no notable improvement in performance on these benchmark problems over using the full training set, we did notice that the solutions that iCDGP found often occurred earlier in evolutionary time than with the full training set. For example, Figure 1 shows the cumulative number of successes over evolutionary time on the Vector Average problem; this plot is representative

Problem	Full	iCDGP	q = 0.8	d = 25	d = 50	d = 100
CSL	32	*13*	41	30	20	*18*
DL	19	24	26	**37**	32	29
LIoZ	62	*41*	56	63	62	65
MI	100	98	*89*	*93*	98	96
NTZ	80	76	80	79	80	81
RSWN	87	94	**96**	91	**96**	91
SS	13	15	**30**	**50**	**31**	**42**
Smallest	100	*93*	*93*	96	95	95
SLB	94	90	90	*83*	87	87
Syl	38	*24*	44	**69**	**62**	49
VA	88	87	89	**97**	**97**	**98**
XWL	61	**75**	**82**	**85**	**89**	**87**

Table 3. The number of successes out of 100 GP runs. All results are compared to **Full**; results that are significantly better are in **bold**, and results that are significantly worse are *underlined and italicized.* **Full** is GP using the full training set. **iCDGP** is the standard version of iCDGP. **q = 0.8** is the variant of iCDGP that adds a case any time an individual passes more cases than the fitness threshold of $q = 0.8$. The three columns labeled with values for **d** are the variant that adds a case to the training set every d generations.

of many of the other problems we observed. Both of the iCDGP methods (with and without a fitness threshold) find many solutions quite early in their runs, reaching 50 successes around 10 million program executions, at which point the full training set has produced only 14 solutions. However, the full training set catches up over evolutionary time, reaching about the same number of successes by the time it hits the maximum number of program executions. iCDGP's ability to find solutions earlier likely stems from it executing many fewer programs per generation, allowing it to produce more generations (and therefore explore more programs) within the same number of program executions. The rapid production of solutions provides one argument for using iCDGP.

5.1 Variant: Generation-Based Case Additions

With generation-based case additions, we add a new case to T_A every time d generations have passed without a new case being added otherwise. We tested three settings for d: 25, 50, and 100 generations. Note that failed iCDGP runs often finished after 1000 to 3000 generations, depending on how many cases are added to T_A.

The last three columns of Table 3 present the number of successful runs for different settings of d. iCDGP with generation-based additions performed similarly to or better than iCDGP without them on every problem for all three settings of d. While all three performed sometimes better and sometimes worse than each other, we will concentrate on $d = 50$ here, which was significantly bet-

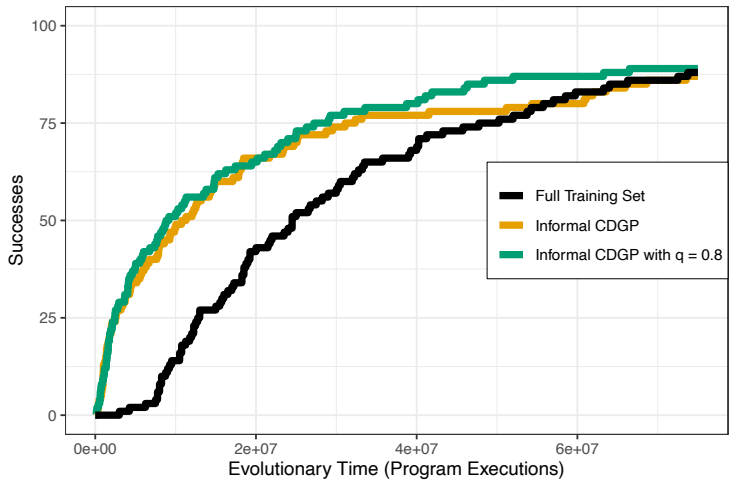

Fig. 1. Cumulative number of successful GP runs on the Vector Average problem over evolutionary time, as measured by program executions.

ter than the full training set on five problems while never performing significantly worse.

Generation-based case additions turns iCDGP's questionable benefits into clear ones. They also present an improvement over using a fitness threshold, likely because they allow for the addition of new cases without an individual having to reach the threshold. Future work would be needed to determine whether selecting a new case that is not passed by the best individual helps, or if adding any random case from $T \setminus T_A$ would be sufficient.

5.2 Variant: Maximum size of active training set

Since it appears that some of the benefits of iCDGP derive from the fact that it uses a small number of cases per program evaluation, we to hypothesize that its performance might improve if the number of cases were capped. For the experiments that produced the results shown in the columns **Max of 10** and **Max of 20** in Table 4, we began with the version of iCDGP using generation-based case additions every 50 generations. To this configuration we added a mechanism that removes a case each time a new case is added, once the number of cases has reached a pre-specified maximum. Specifically, whenever we add a case that would increase the size of T_A past the given limit, we first remove the case in T_A that is passed by the most individuals in the current population, with the intention to remove cases that provide less useful direction to search.

As can be seen in Table 4, limiting T_A to 10 cases degrades problem-solving significantly on the Scrabble Score and Syllables problems. Limiting T_A to 20 cases produces significantly worse results on Scrabble Score, but significantly

Problem	d = 50	Max of 10	Max of 20	Static	DSL
CSL	20	21	**53**	*0*	25
DL	32	46	37	*4*	**72**
LIoZ	62	66	58	*7*	68
MI	98	98	97	*13*	99
NTZ	80	77	80	*31*	84
RSWN	96	88	95	*57*	96
SS	31	*2*	*15*	*13*	*18*
Smallest	95	97	94	*40*	99
SLB	87	93	85	*35*	**96**
Syl	62	*36*	52	*9*	61
VA	97	95	95	*71*	100
XWL	89	91	94	*35*	95

Table 4. The number of successes out of 100 GP runs. All results are compared to **d=50**; results that are significantly better are in **bold**, and results that are significantly worse are *underlined and italicized.* **Max of 10** and **Max of 20** are the variant of iCDGP that caps the size of T_A at 10 or 20 respectively. **Static** uses a fixed training set, for the experiment in Section 5.3. **DSL** is down-sampled lexicase selection, as discussed in Section 5.6.

better on Compare String Lengths. Neither of these results suggests that limiting the size of T_A deserves recommendation.

5.3 Benefits of Counterexample Cases

One may wonder whether the benefits we have demonstrated with iCDGP come entirely from having a small active training set T_A on which we evaluate each individual, reducing the number of program executions per generation. In other words, it is possible that adding counterexample cases to T_A provides no benefits. To test this hypothesis, we conducted a set of runs that use a static active training set consisting of 10 random cases, the same number as the size of T_A at the start of our iCDGP runs. Note that the only functional difference in these methods happens when a program is found that passes all cases in T_A. When a run with a static training set finds a program that passes all cases in T_A, it is simply tested for generalization.

Table 4 compares the number of successes produced by GP with iCDGP adding a case every $d = 50$ generations to GP with a static training set. iCDGP is significantly better on every problem, often by huge margins. These differences highlight the importance of iCDGP's additions of counterexample cases to T_A.

5.4 Effects on Population Diversity

We are interested in the effects of iCDGP, especially with lexicase selection, on population diversity. In particular, as we discussed in Section 4, when an individual passes all cases in T_A and iCDGP adds another case, lexicase selects

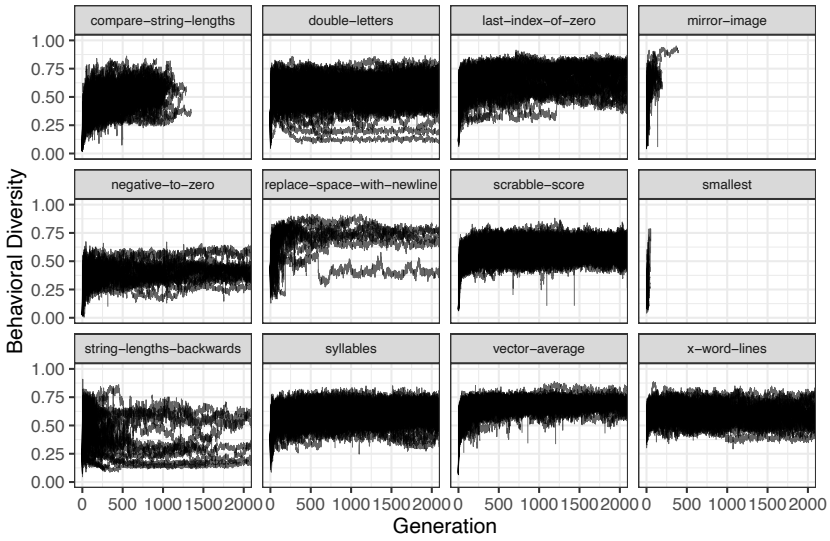

Fig. 2. Population behavioral diversity for standard iCDGP runs, cropped at 2000 generations. Each run is plotted separately. Note that all runs for Mirror Image and Smallest found solutions early in the runs, and Compare String Lengths runs ended earlier than others because they often added many training cases to T_A.

that individual as the parent of every child in the next generation. If more than one such individual is found in the same generation, then the selections will be randomly distributed among them.

These *hyperselection* events will mean that every individual created after such an event will be a descendant of the hyperselected individual. Previous work studied hyperselection events with lexicase selection, but specifically individuals that received 5-10% of the selections in a generation, not 100% of them [12]. This work found that hyperselection events had no noticeable effects on problem-solving performance, and only caused brief reductions in population diversity. Another study of lexicase selection found that it is able to quickly recover population diversity in situations when the population had low diversity [11]. Here we examine whether these hyperselection events have detrimental effects on population diversity when using iCDGP.

We measure population diversity in terms of *behavioral diversity*, or the proportion of distinct behavior vectors produced by a population [22]; a *behavior vector* is a list of outputs that the program produces when run on the cases in T_A. Figure 2 plots the behavioral diversity of every single iCDGP run on all 12 problems as a separate line. Looking closely, there are clearly instances

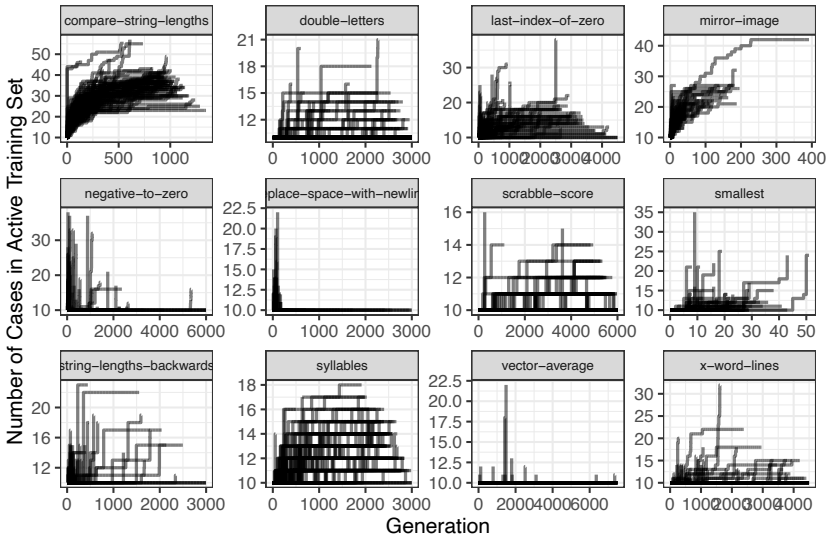

Fig. 3. The number of training cases in the active training set T_A for iCDGP runs. Each run is plotted separately. Note that no cases are ever removed, so each line can only increase. Also note different x-axis and y-axis scales per problem.

where population diversity drops drastically in one generation[9], with many in the Scrabble Score and Vector Average problems, and a few in most of the other problems. Most of these drops in diversity follow one of two patterns: a solution to the full training set T is found in the next generation, leading to a line that drops down and then ends; or a quick increase in diversity over a few generations back to levels seen before the drop. On the other hand, we see little evidence for sudden drops in diversity leading to extended stretches of low diversity.

So, while these hyperselection phenomena do occur when an individual passes all cases in T_A, there does not seem to be corresponding long-term detrimental effects on population diversity. Lexicase selection may be the cause and the cure, as its case-by-case effects provide boosts in diversity following hyperselection.

5.5 Number of Active Cases

In order to get a better idea of how often iCDGP adds cases to T_A, we plot the number of cases in T_A over evolutionary time for standard iCDGP in Figure 3 and for the version that adds a case every 50 generations in Figure 4.

For iCDGP, we see many different patterns of when and how many cases are added to T_A. For example, Compare String Lengths and Mirror Image are

[9] Note that the diversity does not go all the way to 0, since even if one parent created all of the children in the next generation, some of those children are likely to display different behaviors from the parent.

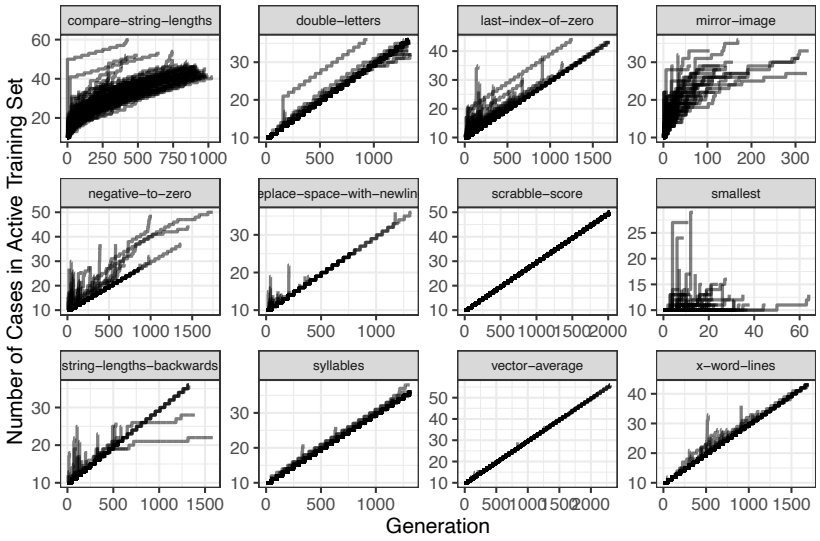

Fig. 4. The number of training cases in the active training set T_A for iCDGP with $d = 50$, adding a new case every 50 generations. Each run is plotted separately. Note that no cases are ever removed, so each line can only increase. Also note different x-axis and y-axis scales per problem.

the only problems in our benchmark set with Boolean-valued outputs, making them easier to pass all cases in T_A without passing all of T than problems with outputs coming from a wider domain, such as numbers or strings. Thus we find it unsurprising that these two problems consistently see the largest growth in T_A. Other problems add cases at different rates, with Replace Space with Newline, Vector Average, and Negative to Zero falling at the other extreme, where a few runs added quite a few cases early and were solved, while the rest never added any cases.

The stair-step pattern of sizes of T_A in Figure 4 reflects the cases that are added to T_A after 50 generations since a case was lasted added. Some problems, corresponding roughly with the problems in Figure 3 that add few cases, rarely if ever add a case besides every 50 generations. Other problems still seem to add quite a few cases for individuals that pass all of T_A. We find no correlation between these two types of problems and those at which this version of iCDGP performs better compared to the standard. The performance improvement seen when adding a case every 50 generations seems to benefit both kinds of problems.

5.6 Comparison with down-sampled lexicase Selection

We compare iCDGP to down-sampled lexicase selection, using results from [15]. To ensure fairness of the comparison, we only consider down-sampled lexicase with down-sample rates which result in 10 cases being evaluated each generation,

which is the size of iCDGP's active set T_A. For instances where no such down-sampling rate existed (Last Index of Zero, Vector Average, and X-Word Lines), we used the results from a down-sampling rate that resulted in just a few more than 10 cases being evaluated per generation.

We found that iCDGP (using $d = 50$) is competitive to down-sampled lexicase selection, with a comparison in Table 4. Of the 12 test problems, iCDGP performed significantly better on one, while significantly worse on only two. The results from down-sampled lexicase selection are among the best results achieved on these PSB1 problems, giving iCDGP a strong comparison to the state-of-the-art.

6 Conclusions and future work

We conclude that informal counterexample-driven genetic programming (iCDGP) advances the state of the art for software synthesis by GP. It builds on the recent advance provided by formal CDGP, but it is likely to be more widely applicable because it does not require a formal specification of solutions to the target problem. The same set of test inputs that would be used for traditional GP can be used for iCDGP, with the only difference being how they are used. Specifically, iCDGP begins with a small initial subset of the cases, and augments the subset with counterexamples whenever an individual passes all of the current cases. We introduce new variants of iCDGP that experimentally outperform the standard version. We recommend using the version that adds a new case to the active set T_A every d generations, ensuring that cases are added even if no program is found that passes all cases in T_A. This variant performed best for iCDGP, and future work could investigate its use in CDGP with formal constraints.

Although we explored several variants of iCDGP, we anticipate other variants to emerge which may outperform ones presented here. Future work should focus on conducting further analyses of the underlying evolutionary dynamics that are responsible for the success of the technique to guide us in developing improvements. We have presented here some preliminary data on behavioral diversity and numbers of cases in T_A over evolutionary time, but many other aspects of these runs can be investigated, and other variants of the technique tested to explore hypotheses about the reasons that it works. For example, with respect to generation-based additions, it would be useful to learn wither it is important to include new cases that are not passed by the best individual, or if the same benefit would result, more simply, from adding any random case from $T \setminus T_A$.

Acknowledgements

We thank the members of the PUSH lab for discussions that improved this work. This material is based upon work supported by the National Science Foundation under Grant No. 2117377. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Błądek, I., Krawiec, K.: Solving symbolic regression problems with formal constraints. In: GECCO '19: Proceedings of the Genetic and Evolutionary Computation Conference. pp. 977–984. ACM, Prague, Czech Republic (13-17 Jul 2019). https://doi.org/doi:10.1145/3321707.3321743
2. Błądek, I., Krawiec, K.: Counterexample-driven genetic programming for symbolic regression with formal constraints. IEEE Transactions on Evolutionary Computation **27**(5), 1327–1339 (2023). https://doi.org/10.1109/TEVC.2022.3205286
3. Błądek, I., Krawiec, K., Swan, J.: Counterexample-driven genetic programming: Heuristic program synthesis from formal specifications. Evolutionary Computation **26**(3), 441–469 (Fall 2018). https://doi.org/doi:10.1162/evco_a_00228
4. Boldi, R., Bao, A., Briesch, M., Helmuth, T., Sobania, D., Spector, L., Lalejini, A.: The problem solving benefits of down-sampling vary by selection scheme. In: Proceedings of the Companion Conference on Genetic and Evolutionary Computation. pp. 527–530 (2023)
5. Boldi, R., Briesch, M., Sobania, D., Lalejini, A., Helmuth, T., Rothlauf, F., Ofria, C., Spector, L.: Informed Down-Sampled Lexicase Selection: Identifying productive training cases for efficient problem solving (Jan 2023). https://doi.org/10.48550/arXiv.2301.01488, http://arxiv.org/abs/2301.01488, arXiv:2301.01488
6. Boldi, R., Lalejini, A., Helmuth, T., Spector, L.: A static analysis of informed down-samples. In: Proceedings of the Companion Conference on Genetic and Evolutionary Computation. p. 531–534. GECCO '23 Companion, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3583133.3590751, https://doi.org/10.1145/3583133.3590751
7. Dreossi, T., Ghosh, S., Yue, X., Keutzer, K., Sangiovanni-Vincentelli, A., Seshia, S.A.: Counterexample-guided data augmentation (2018)
8. Ferguson, A.J., Hernandez, J.G., Junghans, D., Dolson, E., Ofria, C.: Characterizing the effects of random subsampling on lexicase selection. In: Banzhaf, W., Goodman, E., Sheneman, L., Trujillo, L., Worzel, B. (eds.) Genetic Programming Theory and Practice XVII. East Lansing, MI, USA (16-19 May 2019)
9. Helmuth, T., Frazier, J.G., Shi, Y., Abdelrehim, A.F.: Human-driven genetic programming for program synthesis: A prototype. In: Proceedings of the Companion Conference on Genetic and Evolutionary Computation. p. 1981–1989. GECCO '23 Companion, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3583133.3596373, https://doi.org/10.1145/3583133.3596373
10. Helmuth, T., McPhee, N.F., Pantridge, E., Spector, L.: Improving generalization of evolved programs through automatic simplification. In: Proceedings of the Genetic and Evolutionary Computation Conference. pp. 937–944. GECCO '17, ACM, Berlin, Germany (15-19 Jul 2017). https://doi.org/10.1145/3071178.3071330, http://doi.acm.org/10.1145/3071178.3071330
11. Helmuth, T., McPhee, N.F., Spector, L.: Effects of lexicase and tournament selection on diversity recovery and maintenance. In: GECCO '16 Companion: Proceedings of the Companion Publication of the 2016 Annual Conference on Genetic and Evolutionary Computation. pp. 983–990. ACM, Denver, Colorado, USA (20-24 Jul 2016). https://doi.org/doi:10.1145/2908961.2931657, http://doi.acm.org/10.1145/2908961.2931657
12. Helmuth, T., McPhee, N.F., Spector, L.: The impact of hyperselection on lexicase selection. In: Friedrich, T. (ed.) GECCO '16: Proceedings of the 2016 Annual

Conference on Genetic and Evolutionary Computation. pp. 717–724. ACM, Denver, USA (20-24 Jul 2016). https://doi.org/10.1145/2908812.2908851

13. Helmuth, T., McPhee, N.F., Spector, L.: Program synthesis using uniform mutation by addition and deletion. In: Proceedings of the Genetic and Evolutionary Computation Conference. pp. 1127–1134. GECCO '18, ACM, Kyoto, Japan (15-19 Jul 2018). https://doi.org/10.1145/3205455.3205603, http://doi.acm.org/10.1145/3205455.3205603
14. Helmuth, T., Spector, L.: General program synthesis benchmark suite. In: GECCO '15: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation. pp. 1039–1046. ACM, Madrid, Spain (11-15 Jul 2015). https://doi.org/10.1145/2739480.2754769, http://doi.acm.org/10.1145/2739480.2754769
15. Helmuth, T., Spector, L.: Explaining and exploiting the advantages of down-sampled lexicase selection. In: Artificial Life Conference Proceedings. pp. 341–349. MIT Press (15-19 Jul 2020). https://doi.org/10.1162/isal_a_00334, https://www.mitpressjournals.org/doi/abs/10.1162/isal_a_00334
16. Helmuth, T., Spector, L.: Problem-solving benefits of down-sampled lexicase selection. Artificial Life **27**(3–4), 183–203 (2022). https://doi.org/10.1162/artl_a_00341
17. Helmuth, T., Spector, L., Matheson, J.: Solving uncompromising problems with lexicase selection. IEEE Transactions on Evolutionary Computation **19**(5), 630–643 (Oct 2015). https://doi.org/10.1109/TEVC.2014.2362729, http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6920034
18. Helmuth, T., Spector, L., Pantridge, E.: Counterexample-driven genetic programming without formal specifications. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion. pp. 239–240. GECCO '20, ACM (Jul 8-12 2020). https://doi.org/doi:10.1145/3377929.3389983, https://doi.org/10.1145/3377929.3389983
19. Hernandez, J.G., Lalejini, A., Dolson, E., Ofria, C.: Random subsampling improves performance in lexicase selection. In: GECCO '19: Proceedings of the Genetic and Evolutionary Computation Conference Companion. pp. 2028–2031. ACM, Prague, Czech Republic (13-17 Jul 2019). https://doi.org/10.1145/3319619.3326900
20. Hernandez, J.G., Lalejini, A., Ofria, C.: An exploration of exploration: Measuring the ability of lexicase selection to find obscure pathways to optimality. In: Banzhaf, W., Trujillo, L., Winkler, S., Worzel, B. (eds.) Genetic Programming Theory and Practice XVIII. pp. 83–107. Springer Nature Singapore, Singapore (2022). https://doi.org/10.1007/978-981-16-8113-4_5, https://doi.org/10.1007/978-981-16-8113-4_5
21. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (1979)
22. Jackson, D.: Promoting phenotypic diversity in genetic programming. In: Schaefer, R., Cotta, C., Kolodziej, J., Rudolph, G. (eds.) PPSN 2010 11th International Conference on Parallel Problem Solving From Nature. Lecture Notes in Computer Science, vol. 6239, pp. 472–481. Springer, Krakow, Poland (11-15 Sep 2010). https://doi.org/10.1007/978-3-642-15871-1_48
23. Krawiec, K., Błądek, I., Swan, J.: Counterexample-driven genetic programming. In: Proceedings of the Genetic and Evolutionary Computation Conference. pp. 953–960. GECCO '17, ACM, Berlin, Germany (15-19 Jul 2017). https://doi.org/doi:10.1145/3071178.3071224, http://doi.acm.org/10.1145/3071178.3071224, best paper

24. Krawiec, K., Błądek, I., Swan, J., Drake, J.H.: Counterexample-driven genetic programming: Stochastic synthesis of provably correct programs. In: Lang, J. (ed.) Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18). pp. 5304–5308. International Joint Conferences on Artificial Intelligence, Stockholm (13-19 Jul 2018). https://doi.org/doi:10.24963/ijcai.2018/742, https://www.ijcai.org/proceedings/2018/742
25. Moore, J.M., Stanton, A.: Lexicase selection outperforms previous strategies for incremental evolution of virtual creature controllers. Proceedings of the European Conference on Artificial Life pp. 290–297 (2017). https://doi.org/10.1162/ecal_a_0050_14, https://www.mitpressjournals.org/doi/abs/10.1162/ecal_a_0050_14
26. Moore, J.M., Stanton, A.: Tiebreaks and diversity: Isolating effects in lexicase selection. The 2018 Conference on Artificial Life pp. 590–597 (2018). https://doi.org/10.1162/isal_a_00109, https://www.mitpressjournals.org/doi/abs/10.1162/isal_a_00109
27. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society **74**(2), 358–366 (1953), http://www.jstor.org/stable/1990888
28. Schweim, D., Sobania, D., Rothlauf, F.: Effects of the training set size: A comparison of standard and down-sampled lexicase selection in program synthesis. In: 2022 IEEE Congress on Evolutionary Computation (CEC). pp. 1–8 (2022). https://doi.org/10.1109/CEC55065.2022.9870337
29. Sivaraman, A., Farnadi, G., Millstein, T., den Broeck, G.V.: Counterexample-guided learning of monotonic neural networks (2020)
30. Sobania, D., Briesch, M., Röchner, P., Rothlauf, F.: Mtgp: Combining metamorphic testing and genetic programming (2023)
31. Spector, L., Klein, J., Keijzer, M.: The push3 execution stack and the evolution of control. In: GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation. vol. 2, pp. 1689–1696. ACM Press, Washington DC, USA (25-29 Jun 2005). https://doi.org/doi:10.1145/1068009.1068292, http://www.cs.bham.ac.uk/~wbl/biblio/gecco2005/docs/p1689.pdf
32. Welsch, T., Kurlin, V.: Synthesis through unification genetic programming. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference. p. 1029–1036. GECCO '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3377930.3390208, https://doi.org/10.1145/3377930.3390208