Towards Understanding and Refining the General Program Synthesis Benchmark Suite with Genetic Programming

Stefan Forstenlechner*, David Fagan†, Miguel Nicolau‡ and Michael O'Neill§
Natural Computing Research & Applications Group
School of Business
University College Dublin
Dublin, Ireland
Email: * stefan.forstenlechner@ucdconnect.ie, † david.fagan@ucd.ie, ‡ miguel.nicolau@ucd.ie, § m.oneill@ucd.ie

***Abstract*—Program synthesis is a complex problem domain tackled by many communities via different methods. In the last few years, a lot of progress has been made with Genetic Programming (GP) on solving a variety of general program synthesis problems for which a benchmark suite has been introduced. While Genetic Programming is capable of finding correct solutions for many problems contained in a general program synthesis problems benchmark suite, the actual success rate per problem is low in most cases. In this paper, we analyse certain aspects of the benchmark suite and the computational effort required to solve its problems. A subset of problems on which GP performs poorly is identified. This subset is analysed to find measures to increase success rates for similar problems. The paper concludes with suggestions to refine performance on program synthesis problems.**

I. Introduction

Program synthesis is the problem of automatically generating a program that fulfils a certain task given by a user [1]. Applications of program synthesis include helping people accomplish tasks without programming experience, supporting programmers in everyday tasks, and discovering new algorithms. Although program synthesis is not exclusively a Genetic Programming (GP) problem [1], [2], much progress has been made over the last couple of years [3]–[6]. Benefits of adopting a GP approach to program synthesis are that it is very flexible and was shown to be quite successful in tackling a benchmark suite presented at GECCO in 2015 [3]. This benchmark suite consists of 29 problems. At least one correct solution has been found with GP on 22 of these problems when the benchmark suite was originally presented. An additional problem has been solved in a later paper [5].

The benchmark suite consists of problems of varying difficulty and the success rate, i.e. the number of runs that find a correct solution out of all the runs executed, is still rather low in many cases. A solution is considered to be correct if it is able to solve all training and test cases. A solution/program that is nearly correct is of little use in the program synthesis domain. If the success rate for a problem is low, can it really be considered solved? Finding a single correct solution in many runs could be due to random factors and does not show that the algorithm is capable to solve it consistently. Even though the problems vary in difficulty the computational effort specified in the benchmark suite is nearly identical for all problems. This shows that the required effort for different problems is not well understood. Analysing how much computational effort is required to solve a problem would greatly improve the benchmark suite as it helps to perform meaningful comparisons between different approaches because even new approaches or operators will not be of any help if too little computational effort is used.

In this paper, a subset of the problems that either have a small success rate or have not been solved at all from the benchmark suite is used. In a first step, the computational effort is increased compared to the guidelines given by [3]. The purpose is to see if GP, in general, is not able to solve a problem (more often), or if there is an underlying problem that makes it necessary to change the function set available, change the dataset or use new/smarter operators. Further steps are then taken depending on how the success rates have changed.

The rest of the paper is structured in the following way. Section II gives some information about the benchmark suite used in the GP community and what methods have been used to tackle it. Section III explains some approaches to program synthesis, while it focuses on two approaches in detail that have been most successful on the benchmark suite. The description of the experimental setup is given in Section IV. The analysis of the result is in Section V and some hints on how to increase success rates for similar problems are given in Section VI, followed by conclusion in Section VII and future work in Section VIII.

II. General Program Synthesis Benchmark Suite Remarks

Thomas Helmuth and Lee Spector have presented a benchmark suite for general program synthesis [3]. The benchmark suite consists of 29 problems which have been selected from iJava [7] and IntroClass [8]. iJava is an interactive computer science textbook to learn Java. IntroClass is a benchmark suite for program repair of introductory course programs written by

students, although in case of the benchmark suite the purpose is to use these problems to evolve programs. The problems are of varying difficulty and require different data types (boolean, integer, float, string) and even containers like vectors/lists. A detailed documentation is available in the form of a technical report [9]. It contains a description of the training and test data used and how to generate it, the fitness function, the function sets used and suggestions for parameter settings, at least for PushGP [10]. This benchmark suite provides researchers with a well-documented set of problems and a way to compare results of different approaches and methods.

Source code for each problem of the benchmark suite is available and necessary for replication of the training and test data as a few details are missing in the technical report [9]. Also, detailed fitness functions, penalty values for missing results or too short/long vectors are not in the report. Since the benchmark suite has been introduced, the datasets of two problems, namely Checksum and Vector Average, have been changed, which shows that further improvement can be made.

Overall the general program synthesis benchmark suite is a great contribution to researchers working in this field. Changes being made to the benchmarks themselves are to be expected as the suite is rather new and even in other areas discussions about benchmarks are ongoing [11].

A. Solved Problems

When the benchmark suite was first presented PushGP was able to solve 22 of all problems at least one out of 100 times, although success rates vary depending on the problem. Since then, the datasets of two problems, Checksum and Vector Average have been adapted with additional data, which made it possible for PushGP to solve Checksum and increase the success rate of Vector Average [5].

Subsequently, the benchmark suite has been tested with other systems. A Grammar-Guided Genetic Programming (G3P) [4], which will be explained in more detail in Section III-A, whose results were compared to PushGP. Although PushGP was able to find at least one solution to more problems, in general, the success rate on problems that G3P found solutions for was higher on most problems.

A more exhaustive study of different inductive program synthesis methods was conducted by Pantridge et al. [6]. The five methods tested are again the two GP systems PushGP and G3P as well as Flash Fill [12], MagicHaskeller [13], [14] and TerpreT [15], which have not been tested on the benchmark suite before. TerpreT is the only one that has not been used on the benchmark suite, as no implementation of it is publicly available. As FlashFill and MagicHaskeller are deterministic, a single run is sufficient to check if a solution can be found. FlashFill was designed for string manipulation within spreadsheets, therefore it cannot be applied to all problems and subsequently fails to find solutions for many problems in the benchmark suite. MagicHaskeller was applied on all 29 problems and at least managed to find solutions for 6 of them. The results are not compared on success rates, but merely on the fact if a solution was found or not. Which seems reasonable for deterministic algorithms, but bears little meaning for stochastic algorithms. A stochastic algorithm like a GP system that is run up to 100 times on a specific problem and only comes up with a single solution is not reliable and such a problem can hardly be counted as solved. The problem with program synthesis is that contrary to e.g. symbolic regression, a solution close to the (global) optima is of little or no use.

In this paper, we strive to increase the success rate of GP on the problems of the benchmark suite with different means and give general advice when encountering similar difficulties in other problems in the program synthesis domain.

III. Previous Approaches to Program Synthesis

Program synthesis has been of interest even before GP and many approaches exist nowadays that address that problem, like Inductive Logic Programming [16] or SMT (Satisfiability Modulo Theories) Solvers. An overview of different approaches can be found by Kitzelmann [2] and Gulwani [1]. Also in the GP community, different GP systems have been created to tackle program synthesis, like Strongly Typed Genetic Programming [17], Grammatical Evolution [18] or Object Oriented Genetic Programming [19] some of which predate PushGP and G3P by Forstenlechner et al. [4]. The reason the focus lies on these last two systems is that they have been tested on an extensive set of benchmark problems and can handle a large variety of problems, while previous systems may have limited application to a certain problem domain within program synthesis.

A. Grammar-Guided Genetic Programming

Grammar-Guided Genetic Programming (G3P) by Forstenlechner et al. [4] is a grammar-based system that uses context-free grammars (CFGs) and operates on derivation trees like CFG-GP [20]. Unlike previous systems, it does not require a bespoken grammar for every problem, but it contains a set of grammars, one for every data type available in a programming language, that are automatically combined depending on the data types required by the problem tackled. Further, a so-called skeleton which contains the fitness function has to be defined, which can also be used to add additional libraries from a language and to write protected methods, to keep exceptions to a minimum. Grammars and skeletons have to be written for the programming language that code should be evolved for. Grammars and protected methods in a skeleton for the Python language are available, but can easily be created for most programming languages as described in [4]. As grammars and skeletons can be reused across the same programming language, only the fitness function has to be adapted to evolve code in Python for new problems.

The initial approach has undergone some refinement since it has been introduced. The G3P system uses multiple grammars that are combined depending on the datatypes required by the problem. In the original version, this has been done by hand, which is an error-prone process, and the grammars contained unit production rules that had only a single production. Since

then, an automated way of combining grammars has been established, which also removes unit productions as they add an extra node in the GP tree and change the probability where crossover and mutation will occur. Therefore, the success rates shown in this paper may differ from [4], as all the experiments have been run with the automatically combined grammar.

Due to the process of combining the grammars by hand in [4], an error occurred in the Vector Average grammar so that it was not possible to assign a value to the variable that was returned, which made it impossible to solve. After combining the grammars automatically, G3P was able to successfully find correct solutions. Another problem was found in generating the dataset for Super Anagrams. As the description in [9] is incomplete, the actual Clojure source code [21] has to be checked for an exact replication, which lead to a mistake that not all training and test cases were as complicated as they should have been. After fixing only the training was solved but not the test set.

At last, a few clarifications should be made as it seems that there have been misunderstandings in [6]. All problems except String Differences from the benchmark suite have been attempted with G3P in [4], but to save space problems that have not been solved with either G3P or PushGP have not been shown in that paper. Also, G3P can evolve programs in arbitrary languages and not only in Python if provided with grammars as described in that paper. The comparison is somewhat flawed as two of the datasets have been adapted and G3P was not run again on those datasets.

G3P has been used to tackle the benchmark suite and shown similar success to PushGP [4]. The results sorted by the number of successes are shown in Table I. G3P was run 100 times on every problem.

TABLE I
NUMBER OF SOLUTIONS FOUND THAT CORRECTLY SOLVE THE TEST DATA WITH 100 RUNS ON THE GENERAL PROGRAM SYNTHESIS BENCHMARK SUITE WITH G3P. THE TABLE ALSO SHOWS IF A PROBLEM IS USED IN THIS PAPER AND THE NUMBER OF TRAINING AND TEST CASES.

Problem	Successes	Used	Training	Test
NumberIO	94		25	1000
Smallest	94		100	1000
Vectors Summed	91		150	1500
Median	79		100	1000
String Lengths Backwards	68		100	1000
Negative To Zero	63		200	2000
Grade	31	X	200	2000
Last Index of Zero	22	X	150	1000
Super Anagrams	21	X	200	2000
Count Odds	12	X	200	2000
For Loop Index	8	X	100	1000
Small Or Large	7	X	100	1000
Vector Average	5	X	100	1000
Sum of Squares	3	X	50	50
Compare String Lengths	2	X	100	1000
Scrabble Score	2	X	200	1000
Even Square	1	X	100	1000
Checksum	0	X	100	1000
Collatz Number	0		200	2000
Digits	0		100	1000
Double Letters	0		100	1000
Mirror Image	0	X	100	1000
Pig Latin	0		200	1000
Replace Space with Newline	0		100	1000
Syllables	0		100	1000
Wallis Pi	0		150	50
Word Stat	0		100	1000
X-Word Lines	0		150	2000

B. PushGP

PushGP [10] is a GP system that evolves code in the programming language Push which has been created for use in evolutionary computation. PushGP is also the system that was used to tackle the benchmark suite when it was introduced [3], described in Section II. Its reference implementation is in Clojure, but many variants in other programming language are available. Push uses a stacked based model to store variables. One stack for every datatype exists. Data can only be taken from the top and put back on top, therefore additional operations exist to change the order of the elements stored in a stack. Even the operations that are going to be executed are stored in its own stack. In contrary to G3P, no variables have to be defined as data is taken from the stacks. But PushGP can only generate Push code, which is not used other than in the research community.

IV. EXPERIMENTAL SETUP

As the goal of this paper is to analyse and better understand the existing suite of benchmark problems, we first categorized the problems based on the ease with which they have been solved to date. Problems have been put into three categories. Problems with a high, medium and low success rate. Problems with more than 50 successful solutions in 100 runs have been categorized as high success rate, below or equal to 50 but more than 5 in 100 runs as medium and the rest as low success rate. These thresholds have been chosen without a statistical measure and are open to discussion, but seemed adequate when looking at the success rates achieved.

We now turn our focus to the subset of problems with low success rates to gain a deeper understanding of why success rates are low in those instances. The subset of problems selected for this study is shown in Table I. The problems marked with an X in the column Used have been tackled in this study. All problems categorized with a medium success rate as well as all that have been solved at least once from the low success rate problems are used in this study, as the idea is to see if and how the success rate increases. Additionally, Checksum has been added, because its dataset has been adapted in comparison to when the benchmark suite was introduced, as mentioned in Section II and Mirror Images due to experiments conducted outside of this study that have shown that it can be solved with G3P. Also, Super Anagrams can be found twice in the result section. The reason is that the description of the dataset in the paper that introduced the benchmark suite [3] was not very specific and the code to generate it has been adapted to be closer to the original. We kept the previous Super Anagrams dataset for comparison. All problems with a changed dataset in comparison with [4] have been marked with an asterisk (*) in the result section.

TABLE II
PARAMETER SETTINGS

Parameter	Increased effort / Default
Runs	100
Generations	**600** / 300
Population size	**2000** / 1000
Crossover probability	0.9
Mutation probability	0.05
Elite size	1
Node limit	250
Variables per type	3
Max execution time	1 second

A. Parameters and Computational Effort

The benchmark suite [3] was first used with PushGP as explained in Section II and therefore parameter settings for PushGP are available, which are not applicable to all other GP systems, except e.g. population size and number of generations. The population size was set to 1000 for all problems and the maximum number of generations was set to 300 except Number IO, Median and Smallest which used 200. When G3P was tested on these problems, the same settings were used to be able to do a comparison between the two approaches.

To analyse if GP is unable to solve the selected problems more often or if the computational effort it is given is just too limited, the population size and the number of generations were doubled which quadruples the search effort. All other parameter settings have been taken from [4] and no parameter tuning has taken place to be able to compare to previous results. A summary of the settings is shown in Table II. Lexicase selection [22] is used as it was shown to be superior to other selection operators in the program synthesis domain [3]. Runs are stopped as soon as one successful solution based on the training data is found, as there is no further improvement possible without looking at additional data.

B. Larger Training Set

After the experiments, we noticed that some of the problems have a high success rate on training, but fail some of the test cases. Therefore an additional experiment with an increased training set size was carried out with problems which showed such a characteristic after increasing the computational effort. Section V-D explains and discusses that experiment and its results.

V. RESULTS

This section discusses the results of the experiments run with increased effort, where the population size and number of generations were doubled. First, the overall success rate and performance of the two parameter settings are compared. Afterwards, certain problems of the datasets and overfitting are analysed, which are addressed in a subsequent experiment.

A. Success Rates

The number of successful results on all the problems tackled as well as the average test fitness of the best training individual per runs are shown in Table III. The number of successful runs is of importance in program synthesis as a program that not always gives the correct answer might be of little use unless it can be repaired after the run. Nevertheless, the test fitness of the best training solution gives a good indication of how close to the optima a solution is. A Wilcoxon rank-sum test is used to compare the test fitness of the best training solutions. The p-values are also shown in Table III.

Statistically significant different values are indicated in bold. As expected when increasing the computational effort, many problems show a significant difference, 8 out of 15. In some cases, like Compare String Lengths, Grade and Super Anagrams, it is not surprising that no difference is found. Even though the number of successful solutions could be increased, the number of solutions that pass all training cases is already rather high, which indicates that the runs have finished. Therefore the 8 significantly different problems with increased effort are definitely an improvement over the default setting. In many cases, even the number of successfully found solutions has drastically improved. In some cases, this number was nearly doubled or, in the extreme case of Sum of Squares, increased by more than eight times what was achieved with the default setting.

There are two cases in which the default settings have more successful solutions than with increased effort, Small Or Large and Super Anagrams. Small Or Large has less successful solutions with increased effort settings, but at the same time, it achieves a better performance on the average best fitness. When also comparing how many runs successfully managed to find a solution that fits the training data, the increased effort is able to find over 30 more solution, even though they do not generalize. A problem that is discussed in Section V-C. In case of Super Anagrams, the problem is that the training solutions do not generalize to test, which happens with default and increased effort settings. In both cases, the number of successful solutions found with default settings is only slightly bigger.

B. Accumulated Successful Solutions Over Generations

A more fine-grained comparison between the default settings and the increased effort one can be made with Figure 1, which shows the accumulated successful solutions that have been found for every problem over generations. If all the runs of a problem stop before reaching the maximum number of generations due to successful solutions on the training data, the line in the plot is also stopped to indicate how many generations the experiments ran at maximum. This is only the case for Compare String Length and Super Anagram with the increased effort parameter setting.

When comparing the number of successful solutions at generation 300, the results of only doubling the population size can be compared, which only doubles the computational effort. All problems that have more successful solutions, in the end, have already more successful solutions at generation 300 with increased effort except Even Square and Vector Average*, which are both just 1 off. This indicates that an increase of computational effort by a factor of four might not be required.

TABLE III

RESULTS ON BENCHMARK PROBLEMS RUNNING G3P 100 TIMES ON EACH PROBLEM WITH INCREASED EFFORT. THE TABLE CONTAINS THE NUMBER OF SUCCESSFUL RUNS ON TEST AND TRAINING DATA, THE AVERAGE TEST FITNESS AND THE AVERAGE PERCENTAGE OF SOLVED TRAINING AND TEST CASES OF THE BEST SOLUTION FOUND DURING TRAINING WITH THE IMPROVEMENT OVER THE DEFAULT SETTINGS AND THE P-VALUE FROM WILCOXON RANK-SUM TEST ON THE AVERAGE TEST FITNESS. THE RESULT IS COMPARED TO THE DEFAULT SETTING. THE DIFFERENCES ARE SHOWN IN BRACKETS.

Problem Name	Test	Training	Avg Fitness (% Improv.)	Avg Solved Training Cases	Avg Solved Test Cases	p-value
Checksum*	0 (+0)	0 (+0)	31065.12 (+13.74%)	53.37% (+0.21)	30.69% (+0.18)	**2.20E-07**
Compare String Lengths	3 (+1)	100 (+3)	108.32 (+7.56%)	100.00% (+0.03)	89.17% (+0.89)	0.5039
Count Odds	22 (+10)	28 (+16)	3881.11 (+24.25%)	59.58% (+15.92)	44.80% (+15.15)	**0.0029**
Even Squares	2 (+1)	4 (+3)	1947381.31 (+9.24%)	6.60% (+3.83)	5.37% (+3.32)	**2.17E-06**
For Loop Index	21 (+13)	35 (+15)	2591911.15 (+46.33%)	44.85% (+16.63)	44.14% (+16.47)	**1.27E-05**
Grade	34 (+3)	97 (+16)	70.19 (+71.51%)	99.90% (+2.65)	98.60% (+3.20)	0.2906
Last Index of Zero	26 (+4)	64 (+10)	2707.18 (+5.47%)	89.83% (+5.27)	71.87% (+3.55)	0.6149
Mirror Image	1 (+1)	94 (+43)	299.93 (+11.07%)	99.92% (+0.99)	70.01% (+3.74)	**0.0280**
Scrabble Score	7 (+5)	12 (+7)	5425.83 (+14.10%)	39.47% (+16.63)	23.85% (+11.76)	**0.0066**
Small Or Large	4 (–3)	87 (+36)	510.37 (+21.60%)	99.82% (+3.54)	89.76% (+3.30)	0.0981
Sum of Squares	26 (+23)	32 (+29)	58560.96 (+77.65%)	46.46% (+35.66)	43.58% (+34.72)	**6.75E-12**
Super Anagrams	19 (–2)	100 (+1)	22.49 (+1.01%)	100.00% (+0.00)	98.88% (+0.01)	0.5179
Super Anagrams*	0 (+0)	98 (+54)	278.48 (–4.40%)	99.99% (+0.00)	86.08% (–0.01)	**0.0248**
Vector Average	5 (+0)	6 (–1)	93068.60 (+5.99%)	7.58% (+0.86)	6.45% (+0.93)	0.2091
Vector Average*	18 (+2)	19 (+2)	237307.42 (–4.59%)	37.11% (+0.02)	36.05% (+0.03)	0.4693

Only a few problems, like Count Odds, Sum of Squares, For Loop Index and Scrabble Score, seem to take advantage of the increased number of generations. Especially Sum of Squares and Scrabble Score are able to double the number of successful solutions after generation 300.

Another aspect that should be mentioned is that although the increased effort parameter setting was given a total budget of four times the computational effort, this is only the worst-case scenario where no solution is found. As shown in Figure 1, the increase of the population on its own provided better results in most cases and many runs stop before reaching the maximum number of generations. On average the last generation reached over all problems is generation 362 with double the population. This is less than 2.5 times the total budget of computation effort compared to the default setting.

C. Problems with the Training Data

The increased effort further boosts the problem of having solutions that solve the training but do not generalize to the test set similar to the default parameter setting. This phenomenon has already been observed before with program synthesis problems [3], [4]. This boost is expected, as increasing population and generations does not counter this problem, but shows that it is an even bigger concern. Figure 2 depicts the number of successful solutions on training and test. As can be seen on nearly half of the problems the training data is solved by almost all runs, but only a few or no solutions generalize to the test set. This may be due to overfitting or because the training data does not represent the problem well. All runs for the problems Compare String Lengths and Super Anagram stop before reaching the maximum number of generations, generation 146 and 110 respectively, due to all runs finding solutions that solve the training cases. Even without increased effort 97 solutions that solve the training dataset have been found for Compare String Lengths and 99 for Super Anagram.

As mentioned in Section II, the datasets of two problems, Checksum and Vector Average have been adapted before, which lead to finding better solutions with PushGP [5]. This can be confirmed for Vector Average with G3P. This shows that the dataset has a tremendous influence on the success of the search. Adapting the training set seems to be a logical conclusion to represent the problem more accurately and counter overfitting.

D. Larger Training Set

An additional experiment was carried out on all problems that have been solved more than twice as often on the training than on the test data. Compare String Lengths, Grade, Last Index of Zero, Mirror Image, Small Or Large and Super Anagrams*. Each problem gets an additional 100 randomly generated training cases, which relates to a 50-100% increase in training data, depending on the problem. The experiment is run with increased effort as before.

The results are presented in Table IV and compared to the increased effort setting. Figure 3 illustrates these results. Three of the six cases, Compare String Lengths, Last Index of Zero and Small Or Large, show an increase of successful test solutions of up to four times. Only Grade and Mirror Image decrease slightly. Another positive side effect is that the number of solutions that solve the training but do not generalize to the test set decreases in almost all cases. For Mirror Image and Super Anagrams* this number has decreased to less than half than before. The decrease shows the advantage of using a bigger training set, as it indicates that the previous training set might not have represented the problem space accurately. As a solution that solves training but not test is of little use and only stops the search prematurely, no solution found that solves training might be better than one that solves training but not test. One should also be aware that just because a solution solves every case in the test set does not automatically mean that it is correct, as not every possible combination of inputs can be tested.

This experiment shows that with a bigger training dataset that better represents the problem overfitting can be countered

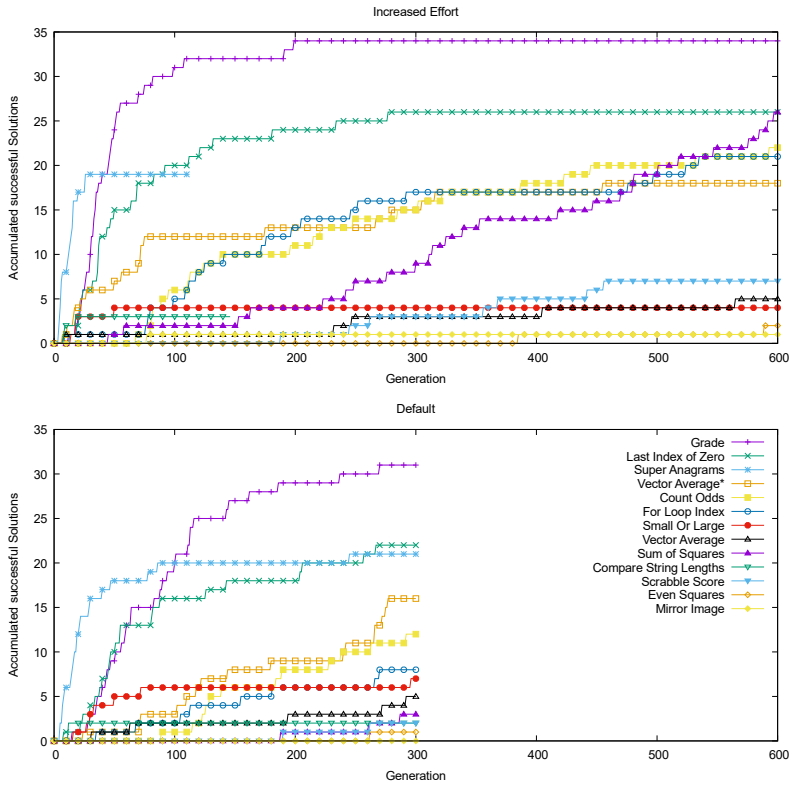

Fig. 1. Accumulated successful solutions over generations over 100 runs.

TABLE IV
RESULTS OF USING AN INCREASED TRAINING DATA. THE TABLE SHOWS THE NUMBER OF SUCCESSFUL SOLUTIONS ON TRAINING AND TEST. THE DIFFERENCE TO THE INCREASED EFFORT SETTING WITH THE ORIGINAL DATASET IS SHOWN IN BRACKETS.

Problem Name	Test	Training
Compare String Lengths	12 (+9)	100 (0)
Grade	29 (–5)	93 (–4)
Last Index of Zero	41 (+15)	79 (+15)
Mirror Image	0 (–1)	24 (–70)
Small Or Large	18 (+14)	88 (+1)
Super Anagrams*	0 (0)	45 (–53)

at least to some degree, as is expected in typical supervised learning. In most cases, the number of runs that only solve the training set is still double compared to the ones that generalize to the test set. Further investigation is needed to better understand the problem and solve it.

One approach by Helmuth et al. [5] is a post simplification process that showed to improve generalization of programs as well as that smaller programs tend to generalize better. Therefore, it might be worth to run experiments with smaller tree sizes.

VI. DISCUSSION

From the analysis in this paper, multiple signals have become apparent that hint at what the next steps should be to increase the success rate on your problems.

The first signal for problems with low success rate is the check if training was solved significantly more often then test. If that is the case, increasing the computational effort has little effect. The step to take in this case is to adapt the dataset to accurately represent the problem or adding more data if available. This is not always possible for real world or when problems are used for comparing different methods. As data sets of some problems in the benchmark suite used in this paper have already been adapted to improve performance before, it is certainly of interest to see what is required to solve others that are still not solved.

The second signal is a low number of solutions that solve training and test. An increased population size has improved most problems. Additionally, an increased number of generations further improved some problems. It has not yet been identified what type of problems may benefit from the increased number of generations at this moment. This is left for future work.

The improvement of success rates is an iterative process as increasing the computational effort can result in an increase of solutions that solve the training, but may fail test and adapting the dataset can lead to requiring more computational effort. If neither of those two methods improves the results anymore, other steps have to be considered, like using better operators or adapting the search space e.g. by changing the function set.

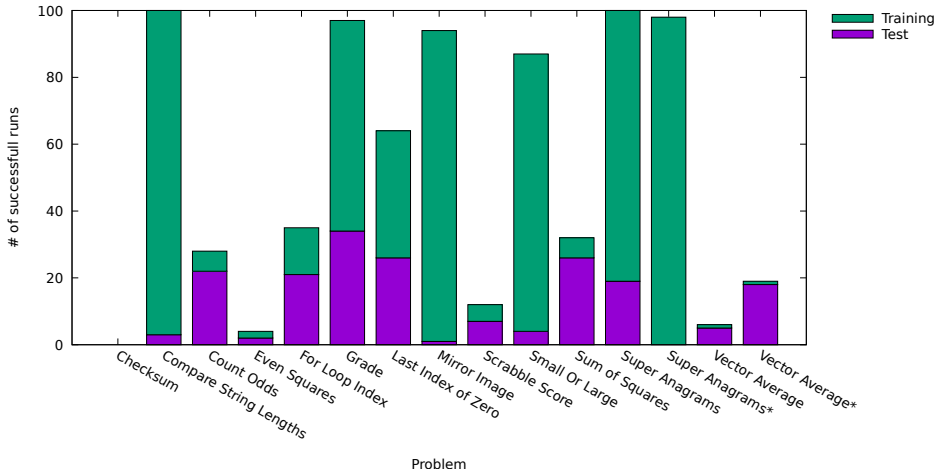

Fig. 2. Number of runs which successfully solve the training and test set per problem.

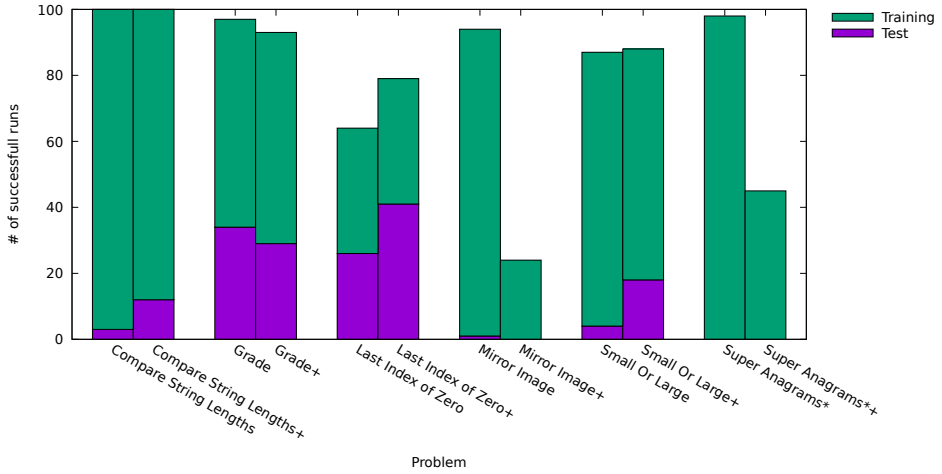

Fig. 3. Number of runs which produce successful solutions that solve training and test with the larger training set. + marks the problems that were run with the increased training set.

VII. Conclusion

We analysed the general program synthesis benchmark suite by categorizing problems into groups depending on their success rate to date using G3P. A subset of problems with low success rates was used to show that G3P is capable to further improve on the problems with increased effort and that parameters of the benchmark suite should be adapted depending on the difficulty of the problem. This could help to have better comparisons when using different approaches and operators, as comparing two suboptimal results due to stopping runs early may have little meaning. A number of signals that help to choose the next steps to improve success rates have been given in the previous section. As program synthesis is a complex problem, it requires a higher computational effort compared to other problems tackled with GP. Increasing population size and the number of generations improved the number of found solutions and showed a statistically significant difference. Simply increasing computational effort does not always mean that GP will get better results, as it can get stuck in local optima or plateaus.

In many cases more computational effort lead to an increase of runs that solved the training but not the test data. This problem was countered by adding additional training samples to the data set, which increased the number of solutions that also solve the test set correctly. In other cases, it prevents that runs stop prematurely due to solutions that only solve the training data but not test.

Both approaches increase the number of solutions found that are successful but increase the computational effort. On the one hand, at least in the worst-case scenario that no solutions are found, and the run only stops due to reaching the maximum number of generations the extra computational effort can be high. On the other hand, it is more likely to find a solution, the solution is more reliable, at least when using more training data, and due to advancements in CPU's computational power has steadily increased and should not be the main concern.

Some of the problems in the general program synthesis benchmark suite are not solved to date and even more have a small success rate. This work is a first step towards enhancing it to understanding its problems better and to identify the effort it takes to solve program synthesis problems. Especially it shows that the available effort should be adjusted per problem instead of using the same parameters independent of the difficulty of the problem tackled. The goal is to have a complete benchmark suite that can also be of benefit when encountering new similar problems. Further refinement of the datasets for certain problems is required as has been done before and has been shown in this study.

VIII. Future Work

Parameter settings in this and previous papers have been chosen either using settings from other papers or due to preliminary experiments. Tuning GP parameters for program synthesis could be a way to further improve results in program synthesis possibly without increasing the computational effort. This could be done by an automatic toolbox like irace [23] or SPOT [24].

As some problems of the benchmark suite used in this paper, have not yet or rarely been solved by G3P or other methods, a focus should be placed on these problems. A detailed analysis of those problems could help to understand why it is difficult for GP to find correct solutions and to further improve GP in the domain of program synthesis.

As mentioned in Section V-D, many programs do not generalize well, even though a larger training set was used. Further analysis is needed to check if runs with smaller tree sizes are able to evolve more successful programs or if other strategies could improve performance and counter overfitting.

Acknowledgments

This research is based upon works supported by the Science Foundation Ireland, under Grant No. 13/IA/1850.

References

[1] S. Gulwani, "Dimensions in program synthesis," in *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, ser. PPDP '10. New York, NY, USA: ACM, 2010, pp. 13–24.

[2] E. Kitzelmann, *Inductive Programming: A Survey of Program Synthesis Techniques*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 50–73.

[3] T. Helmuth and L. Spector, "General program synthesis benchmark suite," in *GECCO '15: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*. Madrid, Spain: ACM, 11-15 Jul. 2015, pp. 1039–1046.

[4] S. Forstenlechner, D. Fagan, M. Nicolau, and M. O'Neill, "A grammar design pattern for arbitrary program synthesis problems in genetic programming," in *EuroGP 2017: Proceedings of the 20th European Conference on Genetic Programming*, ser. LNCS, M. Castelli, J. McDermott, and L. Sekanina, Eds., vol. 10196. Amsterdam: Springer Verlag, 19-21 Apr. 2017, pp. 262–277.

[5] T. Helmuth, N. F. McPhee, E. Pantridge, and L. Spector, "Improving generalization of evolved programs through automatic simplification," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '17. New York, NY, USA: ACM, 2017, pp. 937–944.

[6] E. Pantridge, T. Helmuth, N. F. McPhee, and L. Spector, "On the difficulty of benchmarking inductive program synthesis methods," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '17. New York, NY, USA: ACM, 2017, pp. 1589–1596.

[7] R. Moll, "ijava - an online interactive textbook for elementary java instruction: Demonstration," *J. Comput. Sci. Coll.*, vol. 26, no. 6, pp. 55–57, Jun. 2011.

[8] C. L. Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, Dec 2015.

[9] L. S. T. M. Helmuth, "Detailed problem descriptions for general program synthesis benchmark suite," School of Computer Science, University of Massachusetts Amherst, Tech. Rep., 2015.

[10] L. Spector and A. Robinson, "Genetic programming and autoconstructive evolution with the push programming language," *Genetic Programming and Evolvable Machines*, vol. 3, no. 1, pp. 7–40, Mar. 2002.

[11] D. R. White, J. Mcdermott, M. Castelli, L. Manzoni, B. W. Goldman, G. Kronberger, W. Jaśkowski, U.-M. O'Reilly, and S. Luke, "Better gp benchmarks: Community survey results and proposals," *Genetic Programming and Evolvable Machines*, vol. 14, no. 1, pp. 3–29, Mar. 2013.

[12] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: ACM, 2011, pp. 317–330.

[13] S. Katayama, "Systematic search for lambda expressions," in *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005.*, 2005, pp. 111–126.

[14] ——, *Recent Improvements of MagicHaskeller*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 174–193.

[15] A. L. Gaunt, M. Brockschmidt, R. Singh, N. Kushman, P. Kohli, J. Taylor, and D. Tarlow, "Terpret: A probabilistic programming language for program induction," *CoRR*, vol. abs/1608.04428, 2016.

[16] S. Muggleton, "Inductive logic programming," *New Generation Computing*, vol. 8, no. 4, pp. 295–318, Feb 1991.

[17] D. J. Montana, "Strongly typed genetic programming," *Evol. Comput.*, vol. 3, no. 2, pp. 199–230, Jun. 1995.

[18] M. O'Neill and C. Ryan, *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Norwell, MA, USA: Kluwer Academic Publishers, 2003.

[19] A. Agapitos and S. M. Lucas, *Learning Recursive Functions with Object Oriented Genetic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 166–177.

[20] P. A. Whigham, "Grammatically-based genetic programming," in *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, J. P. Rosca, Ed., Tahoe City, California, USA, 9 Jul. 1995, pp. 33–41.

[21] L. Spector, "GitHub repository: The push programming language and the pushgp genetic programming system implemented in clojure." 2016, [Online; accessed 08-November-2017]. [Online]. Available: https://github.com/lspector/Clojush

[22] T. Helmuth, L. Spector, and J. Matheson, "Solving uncompromising problems with lexicase selection," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 5, pp. 630–643, Oct 2015.

[23] M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, T. Stützle, and M. Birattari, "The irace package: Iterated racing for automatic algorithm configuration," *Operations Research Perspectives*, vol. 3, pp. 43–58, 2016.

[24] T. Bartz-Beielstein, C. Lasarczyk, and M. Zaefferer, "Sequential parameter optimization," in *Proceedings Congress on Evolutionary Computation 2005 (CEC'05)*, Edinburgh, Scotland, 2005, p. 1553.