Challenges of Program Synthesis with Grammatical Evolution

Dominik Sobania(✉) and Franz Rothlauf

Johannes Gutenberg University, Mainz, Germany
{dsobania,rothlauf}@uni-mainz.de

Abstract. Program synthesis is an emerging research topic in the field of EC with the potential to improve real-world software development. Grammar-guided approaches like GE are suitable for program synthesis as they can express common programming languages with their required properties. This work uses common software metrics (lines of code, McCabe metric, size and depth of the abstract syntax tree) for an analysis of GE's search behavior and the resulting problem structure. We find that GE is not able to solve program synthesis problems, where correct solutions have higher values of the McCabe metric (which means they require conditions or loops). Since small mutations of high-quality solutions strongly decrease a solution's fitness and make a high percentage of the solutions non-executable, the resulting problem constitutes a needle-in-a-haystack problem. To us, one of the major challenges of future GP research is to come up with better and more adequate fitness functions and problem specifications to turn the current needle-in-a-haystack problems into problems that can be solved by guided search.

Keywords: Program synthesis · Genetic programming · Grammatical evolution · Software engineering · Needle-in-a-haystack

1 Introduction

Program synthesis, a technique to generate source code in a high-level programming language that meets a certain specification [9], is a relevant research topic in the field of evolutionary computation (EC) with the potential to improve real-world software development. An example showing this potential is the work by Harman et al. [10] in which a translation feature was synthesized by using EC and automatically integrated into the Pidgin instant messaging system.

Grammatical evolution (GE) [22] is a variant of genetic programming (GP) that is suitable for program synthesis, because the used Backus-Naur-Form (BNF) grammar allows GE to express high-level programming languages or subsets of these languages with all their required properties (e.g., conditions, loops, or typing constraints). Inspired by the benchmark suite by Helmuth et al. [13,14], which contains several program synthesis problems selected from introductory programming tasks, some recent work uses grammar-guided approaches for solving program synthesis problems [5,6,15]. For example, Forstenlechner et al. [7]

sorted and classified the problems of the benchmark suite according to the success of G3P, a grammar-guided approach. They found that some problems were easy, whereas others could not be solved a single time. As the reasons for these huge differences in performance are unclear, the next logical step is to study the complexity of the problems and what makes difficult problems difficult for GE.

This work analyzes the behavior of GE as well as the structure of a representative set of program synthesis benchmark problems with common metrics from the EC and software development domains. In a first step, we analyze how robust human-designed reference implementations (solutions that correctly solve a given benchmark problem) are with respect to small modifications of its genotype. In the second step, we analyze by using common and standard software metrics the functions generated by GE during search as well as the resulting problem structure and problem complexity.

Section 2 presents work relevant to the domain of program synthesis with GE. In Sect. 3, we describe the used software metrics, the selected program synthesis problems from the benchmark suite, and the structure of the used GE approach. Following this, in Sect. 4, we describe our experimental setting and discuss the findings. Section 5 concludes the paper.

2 Related Work

There are two major trends for the synthesis of source code with EC: grammar-guided approaches [5–7,15,24] in contrast to approaches based on the stack-based programming language Push [11,12,17]. Both types of methods support the use of multiple data types (e.g., Boolean, integer, float, or string). Grammar-guided approaches, like GE [22], enforce syntax rules and the typing of a programming language by using a BNF grammar, whereas Push [25] ensures correct typing by using separate stacks for each required data type.

For program synthesis with EC, Krawiec [18] already identified some challenges. The most obvious challenge is the large search space. Every additional programming language construct (e.g., a control structure, or a function) leads to a dramatic increase of possible combinations. Even worse, the influence of a programming language construct on the program's behavior is context-dependent as the same instruction in a different setting may lead to completely different results. Furthermore, in a programming language, desired functionality can be expressed in multiple ways (see the multiple-attractor problem [1]). This makes it hard for guided evolutionary search to find a program with the desired functionality and structure. This is also relevant if the evolved program should be improved or maintained by human software developers as they expect human-readable code and not overly complex, but correct, synthesized program code. Therefore, evolved code should not only have the desired functionality but also follow a human-like coding style [24]. Another unsolved problem is how to measure whether a program has the desired functionality. For example, the well-known benchmark suite for program synthesis [13,14] checks the correctness of a program with large sets of test cases. Unfortunately, the use of test cases does not allow to appropriately measure generalization as even Dijkstra [2, p. 864] pointed out that *"program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence"*.

A variety of GE papers analyzed different aspects of the algorithm, like the influence of grammars [16,20], the genotype-phenotype mapping [4,21], or the initialization method [3,24]. However, to our knowledge, there is no work so far that performs a systematic analysis of GE's behavior on a representative set of program synthesis benchmark problems using standard and common software metrics.

3 Methodology

This section presents the software metrics required for analysis, the selected benchmark problems, and the structure of the GE approach.

3.1 Software Metrics

In our experiments, we use software metrics that are directly applied to a function's source code as well as metrics that measure properties of a function's abstract syntax tree (AST). For generating the AST from a given Python function, we use the Python module `astdump`[1]. We use the following software metrics:

– **Lines of code (LOC)**: the number of lines of a function's source code including the function's signature. Comments and empty lines are not relevant in this work because the used grammar does not support them.
– **McCabe metric**: the number of decision branches defined by a piece of code added to the minimum value which is one [19]. Decision branches arise in source code, e.g., through conditions and loops. For calculating the McCabe metric, we use the Python module `radon`[2].
– **AST depth**: the number of edges on the path from an AST root node to its deepest leaf node.
– **AST nodes**: the number of nodes in an AST.

3.2 Program Synthesis Problems

For our experiments, we selected four problems from the 29 problems defined in the program synthesis benchmark suite [13,14]. To obtain a representative subset of test problems, we selected problems with different complexity and different data types necessary for a correct solution. We selected the following problems:

– **Number IO**: return the sum of a given integer and a float.
– **Small or Large**: for a given integer n, return "small" if $n < 1,000$, "large" if $n \geq 2,000$, and an empty string if $1,000 \leq n < 2,000$.

[1] https://pypi.org/project/astdump/.
[2] https://pypi.org/project/radon/.

– **Count Odds**: return the number of odds in a given vector of integers.
– **Smallest**: return the smallest of four given integers.

The training sets consist of 100 cases for each problem, except for Number IO where it consists of 25 and Count Odds where it consists of 200. The test sets consist of 1, 000 cases for each problem, except for Count Odds (2, 000).

```
def count_odds(numlist0):
    num0 = num1 = num2 = num3 = num4 = 0     # Initialization
    for num1 in numlist0:
        if num1 % 2 == 1:
            num0 = num0 + 1
    return num0
```

Fig. 1. The reference implementation for the Count Odds problem. We shortened the initialization part.

For each of the considered benchmark problems, we defined a hand-written reference implementation that correctly solves the problem and which resembles a solution written by a human software developer. Figure 1 exemplarily shows the reference implementation for the Count Odds problem. Since the reference implementations are consistent with the BNF grammar used in the experiments (see Sect. 3.3), they also contain an initialization part for all possible variables (shortened for readability in the figure). For reproducibility, all reference implementations are available online[3].

Table 1. Properties of the reference implementations. The values in brackets are without the not required part of the initialization.

Benchmark problem	LOC	McCabe metric	AST depth	AST nodes
Number IO	12 (3)	1 (1)	5 (5)	74 (19)
Small or Large	18 (9)	3 (3)	6 (6)	92 (35)
Count Odds	13 (6)	3 (3)	7 (7)	84 (34)
Smallest	12 (3)	1 (1)	7 (7)	85 (34)

To assess the structure and complexity of the problems, Table 1 shows the calculated software metrics (see Sect. 3.1) for our reference implementations. The values in brackets show the software metrics without the not required part of the variable initialization.

As we can see, the selected benchmark problems cover a range from 12 to 18 LOC, respectively 3 to 9 LOC without the not required part of the initialization.

[3] https://gitlab.rlp.net/dsobania/ge-program-synthesis/tree/master/reference.

The complexity, measured by the McCabe metric, ranges from 1 to 3. The AST-based metrics are distributed in a similar way. As expected, the initialization part has no influence on the McCabe metric and the AST depth.

3.3 GE Grammar and Fitness Function

For our experiments, we use a standard GE approach with a BNF grammar. Since the GE only uses the training set during a run (which means that we make no assumptions on the type of problem), we created a very expressive grammar which supports all 29 problems from the program synthesis benchmark suite [13,14]. The resulting BNF grammar consists of 31 production rules and supports conditions, loops, numbers, strings, Booleans, and lists. Slicing of strings and lists is also possible. Figure 2 shows an excerpt of the used BNF grammar. For reproducibility, the complete grammar is available online[4].

```
<main> ::= def small_or_large(num0): NEWLINE INDENT num1 = num2 =
           num3 = num4 = 0 NEWLINE bool0 = bool1 = bool2 = False
           NEWLINE numlist0 = [] NEWLINE numlist1 = [] NEWLINE
           numlist2 = [] NEWLINE str0 = str1 = str2 = "" NEWLINE
           strlist0 = [] NEWLINE strlist1 = [] NEWLINE
           strlist2 = [] NEWLINE <stmt> return <expr_string>
<stmt> ::= <stmt> <stmt> | <var_numeric> = <expr_numeric> NEWLINE |
           <var_bool> = <expr_bool> NEWLINE | ...
```

Fig. 2. An excerpt of the used BNF grammar. The shown first production rule is designed for the Small or Large benchmark problem.

The only problem-specific adaptation of the BNF grammar is the first production rule, which defines – for each considered problem – the function arguments, ensures an initialization of all variables, and defines the return type. For example, Fig. 2 lists inter alia the first production rule for the Small or Large benchmark problem. The rest of the grammar is identical for all problems. The indentation style, which is mandatory in the Python programming language, is realized by newline, indent, and dedent markers in the grammar. These markers are replaced before the evaluation.

For the evaluation of a solution (function), we use the same fitness function for all benchmark problems defined as

$$f(S, T, p_{inv}, p_{err}) = \begin{cases} p_{inv}\,|T| & \text{if } S \text{ is invalid,} \\ p_{err}\,|T| & \text{if } S \text{ causes a run-time error,} \\ \sum\limits_{t_i \in T} d(S, t_i) & \text{else,} \end{cases} \tag{1}$$

where S is a candidate solution (a generated Python function), T is the training set, t_i is the ith element of T, p_{inv} is the penalty for invalid solutions, p_{err}

[4] https://gitlab.rlp.net/dsobania/ge-program-synthesis/tree/master/grammar.

is the penalty for candidate solutions causing a run-time error, and $d(S, t_i)$ is a function that returns 0 if the candidate solution produces the correct output and 1 otherwise. Thus, the fitness of a candidate solution is increased by 1 for every element of the training set that is not correctly solved. If a candidate solution is invalid, because the genotype-phenotype mapping is not successful, we assign the penalty p_{inv} to all elements of the training set. If a candidate solution causes a run-time error (e.g., an index error, a division by zero error, or an endless loop), we apply analogously the penalty p_{err}.

4 Experiments and Discussion

Sections 4.1 and 4.2 analyze the neighborhood (fitness and structure) of the reference implementations for the selected program synthesis problems. Section 4.3 analyzes the properties of the Python functions evolved during a GE run. In Sect. 4.4, we study how the fitness of solutions depend on the number of AST nodes, the AST depth, and the McCabe metric.

4.1 Robustness of Reference Implementations: Part I

We use random walks to study how robust the reference implementations are with respect to small modifications of the genotype. A problem (or more precise, the GE genotype of a reference implementation of a problem) is robust if small modifications of the genotype have only low effect on the properties and fitness of the corresponding phenotype. We use robustness in the sense of locality [21], where small changes of a genotype should correspond to small changes of the phenotype.

For our study, we created a corresponding GE genotype for each of the reference implementations (phenotypes). Then, we iteratively apply random mutations to the active codons of the GE genotype. After each mutation, we calculate the fitness (Eq. 1) of the solution, the number of not correctly solved training cases as well as the percentage of invalid solutions, solutions that cause a run-time error, and solutions that are still executable.

Since we have defined a large and expressive BNF grammar, we use an integer-based genome with a length of 250 (number of codons) and a codon size of $1,000$. As we do not need all codons of the genotype for encoding a reference implementation, we fill all non-used (inactive) codons with random integers. As the mutations may introduce endless loops, we limit the fitness evaluation to 3 s. If the evaluation is not completed within this time, it will be aborted and the solution counts as a run-time error. In all experiments, we set the invalid penalty $p_{inv} = 2$ and the run-time error penalty $p_{err} = 1.5$. Furthermore, we use no wrapping in the genotype-phenotype mapping process.

Figures 3, 4, 5, 6, 7, 8, 9 and 10 show results for all four benchmark problems. The plots on the left show the average fitness as well as the average number of not correctly solved training cases (denoted as wrong cases) over the number of random mutations. For the fitness plot, we consider all solutions including the ones that cause a run-time error or are invalid. For the plots showing the number of not correctly solved training cases, we consider only solutions that are executable. We average results over 5,000 runs. In each run, we iteratively apply a finite number of random mutation steps starting from the (correctly working) reference implementation. As intended, the reference implementations always have a fitness of zero. The worst solutions, which are functions that are invalid, have a fitness of p_{inv} times the number of training cases. The plots on the right show the percentage of invalid solutions, solutions causing a run-time error, and executable solutions over the number of random mutations (averaged over 5,000 runs).

The fitness plots (left) show similar behavior for all problems. Even very few changes of the GE genotype of only one or two mutations strongly decrease the fitness of a solution. For example, one or two mutations applied to the reference implementation of the Small or Large problem reduces the average fitness from 0 (reference implementation) to 85 or 128, respectively. After a few more mutations, the average fitness is close to 200, which indicate invalid solutions. There are small differences in how fast the solutions become infeasible depending on the considered problem. For example, the fitness slope for the Smallest problem (Fig. 9) is slightly lower compared to the other benchmark

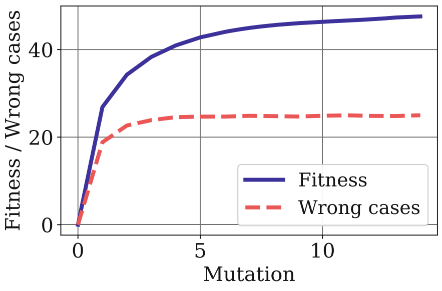

Fig. 3. Fitness/wrong cases over mutations for the Number IO problem.

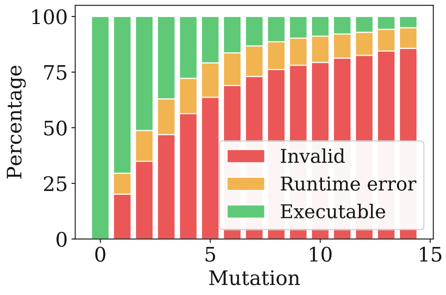

Fig. 4. Percentage of result types over mutations for the Number IO problem.

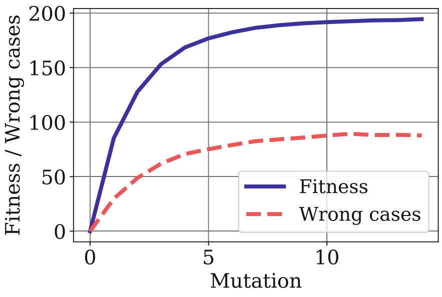

Fig. 5. Fitness/wrong cases over mutations for the Small or Large problem.

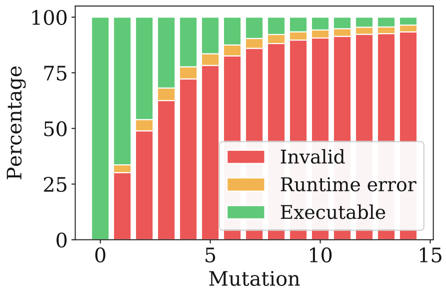

Fig. 6. Percentage of result types over mutations for the Small or Large problem.

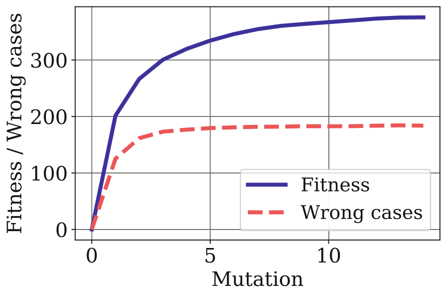

Fig. 7. Fitness/wrong cases over mutations for the Count Odds problem.

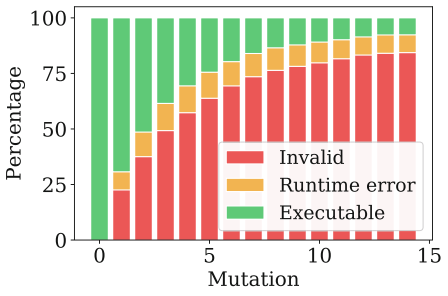

Fig. 8. Percentage of result types over mutations for the Count Odds problem.

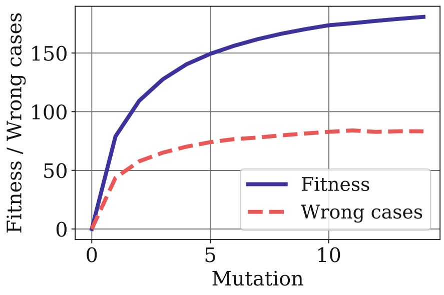

Fig. 9. Fitness/wrong cases over mutations for the Smallest problem.

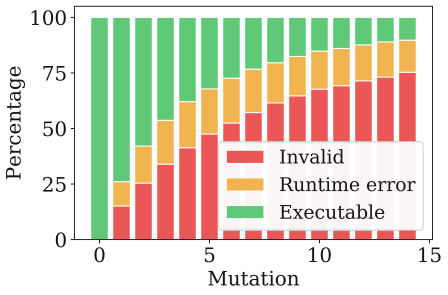

Fig. 10. Percentage of result types over mutations for the Smallest problem.

problems. Since the number of invalid solutions and run-time errors strongly influence a solution's fitness, we also plot the average number of wrongly solved training cases (denoted as wrong cases). For the Number IO (Fig. 3) and the Count Odds problem (Fig. 7), only a few mutations strongly increase the number of wrongly solved training cases; for the Small or Large (Fig. 5) and the Smallest problem (Fig. 9), the mutations have a slightly lower negative influence.

The figures on the right side plotting the percentage of invalid solutions, solutions with a run-time error, and executable solutions confirm the findings and show that after only a few mutations, a large percentage of the solutions are non-executable. For example, for the Small or Large problem, two random mutations of the genotype lead to an average percentage of less than 50% executable solutions. One reason for this high percentage of invalids is the large grammar with many non-terminals. To make the grammar more robust, Schweim et al. [23] suggest to reduce the grammar's average branching factor, e.g., by a lower arity of the functions or adding more terminals to the grammar. Another way to downsize the grammar is the use of domain knowledge, e.g. the textual problem description of the program to be synthesized (cf. Hemberg et al. [15]).

In summary, the reference solutions are not robust against small changes of the genotype. Step-wise random mutations strongly reduce the percentage of executable functions. After about 10 mutations, less than 20% of the solutions are executable. We expect that the high percentage of non-executable solutions in the neighborhood of the reference implementations make it difficult for guided search approaches like GE to find correct solutions.

4.2 Robustness of Reference Implementations: Part II

We also present results for the robustness of the reference implementations with respect to the software metrics defined in Sect. 3.1. For the same experimental setting and identical experimental runs as described in the previous section, we now present results on how the structure and complexity (measured by the software metrics presented in Sect. 3.1) of the reference implementations change when applying subsequent mutations.

Figures 11, 12, 13 and 14 present the average LOC, McCabe metric, number of AST nodes, and AST depth over the number of random mutations. The software metrics are calculated for the complete functions including the variable initialization part (e.g., for Number IO the smallest possible value of LOC is 12). For the analysis, we excluded invalid solutions because for such solutions no well-formed phenotype exists.

The results show only small changes of LOC and McCabe metric over the number of mutations. We observe a slight difference between the Number IO problem (Fig. 11) and Smallest Problem (Fig. 14) on the one hand, where the reference implementations have low LOC and McCabe metric values, and the more complex Small or Large problem (Fig. 12) and Count Odds problem (Fig. 13) on the other hand, where the reference implementations have slightly higher values for LOC and McCabe metric. For the easier problems (Number IO and Smallest), iterative mutations do not significantly change the LOC and McCabe metric values; for the more complex problems (Small or Large and Count Odds), mutations slightly decrease LOC and McCabe metric. For example, for the Small or Large problem, the average LOC decreases from 18 to around 15. Thus, on

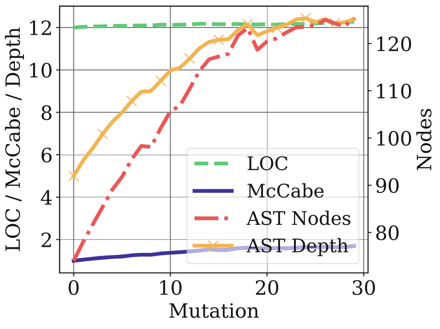

Fig. 11. Software metrics over mutations for the Number IO problem.

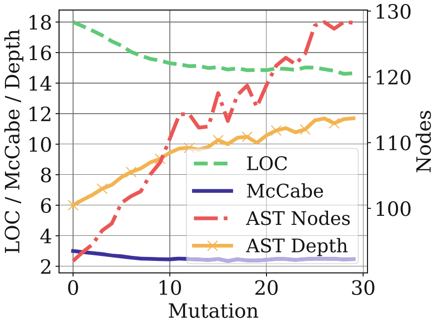

Fig. 12. Software metrics over mutations for the Small or Large problem.

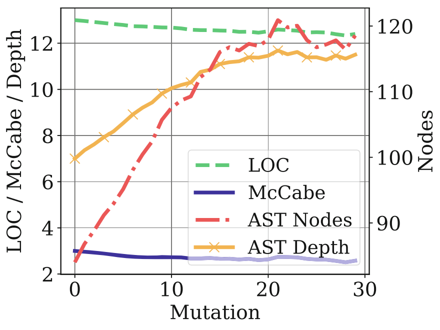

Fig. 13. Software metrics over mutations for the Count Odds problem.

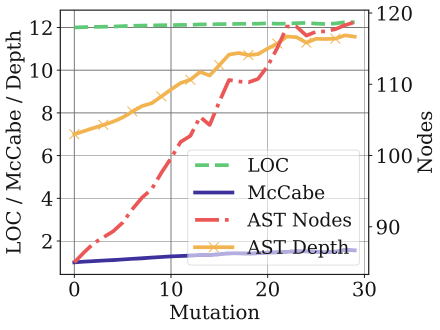

Fig. 14. Software metrics over mutations for the Smallest problem.

average small iterative random changes of the genotype of the reference implementations tend to either reduce or keep constant a solution's size measured by LOC and complexity measured by the McCabe metric.

In contrast, iterative random mutations of the reference implementation strongly increase the average number of AST nodes as well as AST depth for all benchmark problems. 20 random mutations increase the average number of AST nodes by more than 25 nodes; analogously, average AST depth goes up by about 4 nodes. For example, the largest AST depth values of the reference implementations is 7 (Count Odds and Smallest), which increases to more than 10 after about 18 mutations.

Thus, random mutations do not significantly increase the number of lines of code of a function but strongly increase the average length and complexity of each line of code. This leads to more complex and long lines of code. The resulting programs (with high complexity and length of a line of code) are difficult to understand (and not really maintainable) by human programmers.

4.3 Search Behavior of GE

Human programmers that develop correct solutions for the existing benchmark problems often make use of conditions and loops (compare Table 1 for the resulting properties of our – human-coded – reference implementations). This section studies the metrics of functions evolved during a GE run.

The GE uses a population of 25,000 individuals, an integer-based genome of length 250 with a codon size of 1,000. As before, we use no wrapping. We use tournament selection of size 7 and set the crossover probability to 0.7 and the mutation probability to 0.03. As before, we stop the evaluation of a solution after 3 s and set $p_{inv} = 2.0$ and $p_{err} = 1.5$. We stop each GE run after 50 generations.

Table 2 shows the number of test cases, the average and the standard deviation of correctly solved test cases, and the success rate (number of runs that found a correct solution) for the benchmark problems. Results are averaged over 100 runs. We show results for the best solution found during a run.

Table 2. GE performance for the benchmark problems.

Benchmark problem	#Test cases	#Correctly solved cases		Success rate
		Average	Std. dev.	
Number IO	1000	1000.0	0.0	100
Small or Large	1000	531.5	20.2	0
Count Odds	2000	243.9	81.6	0
Smallest	1000	805.5	106.2	14

For the Number IO problem, all GE runs find a correct solution; for the Smallest problem, only 14% of the runs find a correct solution. For the two other benchmark problems, GE does not find a correct solution, nevertheless, evolves solutions that solve some of the test cases. Thus, GE finds correct solutions only for relatively simple problems (Number IO and Smallest), where the reference implementation has a McCabe metric value of one (see Table 1). For the two other, more difficult, problems, where the reference solutions implemented by a human programmer have a McCabe metric of three (see Table 1), GE is not able to evolve a single solution that solves the problem.

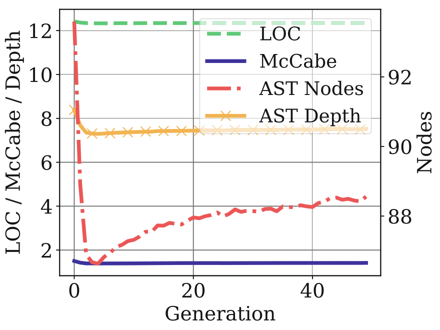

Fig. 15. Software metrics over generations for the Smallest problem.

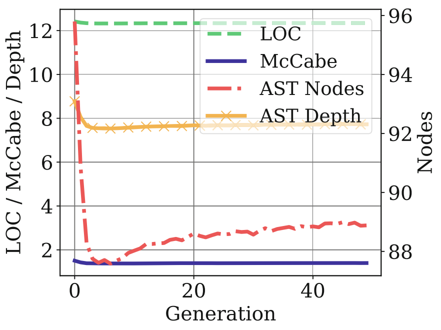

Fig. 16. Software metrics over generations for the Small or Large problem.

To better understand the differences in GE performance, we analyze the development of the software metrics during a GE run. Figures 15 and 16 plot the average LOC, McCabe metric, number of AST nodes, and AST depth of the solutions evolved by the GE over the number of generations for the Smallest and the Small or Large problem. For both problems, LOC, McCabe metric, and AST depth slightly decrease in the first generations. Afterwards, these values remain about constant. More interestingly, the average number of AST nodes slightly decreases in the first generations (from around 95 down to 87) followed by a slight increase approaching a value lower than the initial one. Comparing these findings with the metrics of the reference implementations (Table 1), the GE finds solutions with similar values of LOC, AST depth, and number of AST nodes. A major difference lies in the McCabe metric, where the GE only evolves solutions with average McCabe metrics of around one. However, to solve the (more difficult) Small or Large problem, higher McCabe metric values would be necessary. For the Smallest problem, the reference implementation has a McCabe metric of only one, which makes the problem easy and allows GE to sometimes solve the problem. The results for the other studied benchmark problems are similar, but are omitted due to space limitations.

The results indicate that evolutionary search is not able to generate more complex solutions with a higher McCabe metric. Thus, GE has problems to correctly use conditions and loops within a solution. Indeed, to evolve a solution with high fitness that uses a condition or loop, many elements of a programming language must fit together and the parameters of the condition or loop must be appropriately set.

```
def smallest(num1, num2, num3, num4):
    # Initialization
    numlist0.append(5)
    return min((num1), min(min((num4), num3), num2))
```

Fig. 17. Correct solution found for the Smallest problem (initialization part is omitted).

```
def small_or_large(num0):
    num1 = num2 = num3 = num4 = 0     # Initialization
    numlist0 = list(reversed(list(range(num0 + num2 + (-1000), 2))))
    return "small"[:len(numlist0)]
```

Fig. 18. Best solution found for the Small or Large problem (shortened initialization).

Consequently, we perform a visual inspection of the source code of the solutions found by GE. Figure 17 shows an example of a (correct) solution found for the Smallest problem. A solution for this problem should return the smallest of four given integers. Unfortunately, the BNF grammar only contains a minimum function $min(a, b)$ that accepts two inputs a and b. Thus, the solutions evolved by GE combines $min()$ multiple times with the four input variables as parameters. The solution found by GE is similar to the reference implementation.

Figure 18 shows the best found solution for the Small or Large problem. To solve this problem, a human programmer would use conditions. GE is not able to solve the problem, but only finds solutions that are correct for some cases. In none of the solutions returned by the GE, conditions have been used in some useful way. Instead, GE finds solutions that mimic conditions by performing many nested simple operations on inputs. For example, the shown example solution uses string slicing and the length of a generated list to return either "small" or an empty string solving correctly around two thirds of the test cases.

In summary, problems are difficult for evolutionary search if they require the usage of conditions and loops (solutions with a higher McCabe metric). Finding such structures is difficult for GE as correct solutions with conditions and loops are difficult to construct (many variables and programming language constructs have to be set correctly to get a useful condition or loop) and solutions using loops easily become non-executable when applying mutations.

4.4 Search for the Needle in a Haystack

To better understand what makes a problem difficult for GE, we study how the fitness of solutions depend on the number of AST nodes, the AST depth, and the McCabe metric. For the Smallest problem, Figs. 19, 20 and 21 plot the average as well as the absolute best fitness of all visited solutions of all 100 runs over the number of AST nodes, AST depth, and McCabe metric, respectively. Figures 22, 23 and 24 show results for the Small or Large problem. We also plot the position of the reference implementation (with fitness 0). The plots include all non-invalid solutions that have been generated during the 100 GE runs.

For the Smallest problem, the reference implementation has a McCabe metric of only 1, an AST size of 85, and an AST depth of 7. The plots show that GE finds many solutions that have similar metric values compared to the reference implementation and high fitness (relevant is the best solution found for a given number of AST nodes, depth, or McCabe metric). Solutions with lower or higher values of AST nodes and depth tend to be worse (higher fitness values). Analogously, a higher value of the McCabe metric leads to worse solutions.

The situation is different for the more difficult Small or Large problem, where the reference implementation has a McCabe metric of 3, an AST size of 92, and an AST depth of 6. GE finds solutions with similar values for AST size and depth as well as McCabe metric, but none of the found solutions has high fitness. Instead, for a given value of a metric, all best found solutions have relatively high fitness values independently of the value of the McCabe metric, number of AST nodes, and AST depth. Thus, the fitness landscape (with respect to metrics

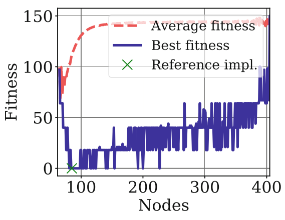

Fig. 19. Fitness over AST nodes (Smallest problem).

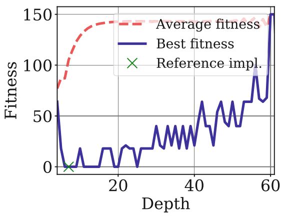

Fig. 20. Fitness over AST depth (Smallest problem).

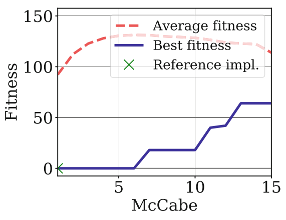

Fig. 21. Fitness over McCabe metric (Smallest p.).

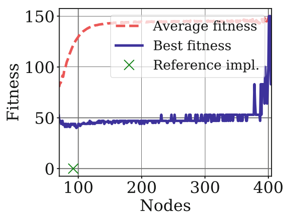

Fig. 22. Fitness over number of AST nodes (Small or Large problem).

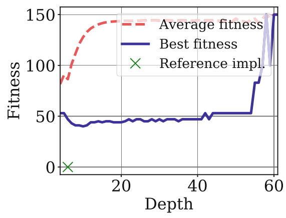

Fig. 23. Fitness over AST depth (Small or Large problem).

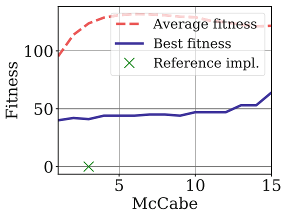

Fig. 24. Fitness over McCabe metric (Small or Large problem).

measuring size, complexity, or structure of a solution) does not guide (in contrast to the Smallest problem) evolutionary search towards promising solutions but the problem of finding a correct solution is a needle-in-a-haystack problem [8]. When searching through the search space, GE cannot exploit relevant information on where promising solutions are, but finding a correct solution becomes the task of finding a solution with fitness 0 (the needle) in a search space where all other solutions have a fitness of around 50 or worse (the haystack).

5 Conclusions

Program synthesis is an emerging EC research topic with the potential to improve real-world software development. Grammar-guided approaches like GE are suitable for program synthesis as they can express high-level programming languages or subsets of these languages with all their required properties like conditions, loops, or typing constraints. However, program synthesis is a complex problem and researchers as well as practitioners should know about the challenges of this domain. Therefore, this work analyzed the behavior of GE on a representative set of program synthesis benchmark problems using standard and common software metrics like LOC, McCabe metric, or the number of nodes and depth of an AST.

First, we analyzed how robust reference implementations – where each of the hand-written implementations is a correct solution for a benchmark problem – are with respect to small modifications of its genotype. We found that small changes strongly decrease a solution's fitness, make a high percentage of the solutions non-executable, and also have a negative impact on a solution's structure measured by the software metrics. Iterative mutations generate solutions with sometimes a lower number of LOC and McCabe metric but simultaneously strongly increase the number of AST nodes and AST depth. Such solutions do not make use of conditions or loops but contain complex and long code lines.

Second, we studied the properties of functions generated during a GE run. We found that GE is not able to solve program synthesis problems, where correct solutions have higher values of the McCabe metric (which means they require conditions or loops). Evolving such high-quality solutions with higher values of the McCabe metric is a difficult task for GE, as a reasonable use of conditions or loops requires the correct and simultaneous setting of many variables and programming language constructs. Our analysis shows that finding high-quality solutions with a McCabe metric larger than one becomes the task of finding a solution with fitness 0 (the needle) in a search space where all other solutions have a worse fitness value (the haystack).

We conclude that program synthesis is a highly relevant problem and the collection and formulation of program synthesis benchmark problems provides the EC researchers relevant goals. However, the current problem specification and especially the definition of the fitness functions do not allow guided search as the resulting problem constitutes a needle-in-a-haystack problem. The structure of the search space provides no meaningful information for heuristic search to evolve more complex optimal solutions that require conditions or loops. Therefore, we see one of the main challenges for future GP research to come up with better fitness functions and problem specifications to turn the current needle-in-a-haystack problems into problems that can be solved by guided search.

References

1. Altenberg, L.: Open problems in the spectral analysis of evolutionary dynamics. In: Menon, A. (ed.) Frontiers of Evolutionary Computation. GENA, vol. 11, pp. 73–102. Springer, Boston (2004). https://doi.org/10.1007/1-4020-7782-3_4
2. Dijkstra, E.W.: The humble programmer. Commun. ACM **15**(10), 859–866 (1972)
3. Fagan, D., Fenton, M., O'Neill, M.: Exploring position independent initialisation in grammatical evolution. In: IEEE Congress on Evolutionary Computation, pp. 5060–5067. IEEE (2016)
4. Fagan, D., O'Neill, M., Galván-López, E., Brabazon, A., McGarraghy, S.: An analysis of genotype-phenotype maps in grammatical evolution. In: Esparcia-Alcázar, A.I., Ekárt, A., Silva, S., Dignum, S., Uyar, A.Ş. (eds.) EuroGP 2010. LNCS, vol. 6021, pp. 62–73. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12148-7_6
5. Forstenlechner, S., Fagan, D., Nicolau, M., O'Neill, M.: A grammar design pattern for arbitrary program synthesis problems in genetic programming. In: McDermott, J., Castelli, M., Sekanina, L., Haasdijk, E., García-Sánchez, P. (eds.) EuroGP 2017. LNCS, vol. 10196, pp. 262–277. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-55696-3_17
6. Forstenlechner, S., Fagan, D., Nicolau, M., O'Neill, M.: Extending program synthesis grammars for grammar-guided genetic programming. In: Auger, A., Fonseca, C.M., Lourenço, N., Machado, P., Paquete, L., Whitley, D. (eds.) PPSN 2018. LNCS, vol. 11101, pp. 197–208. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99253-2_16
7. Forstenlechner, S., Fagan, D., Nicolau, M., O'Neill, M.: Towards understanding and refining the general program synthesis benchmark suite with genetic programming. In: IEEE Congress on Evolutionary Computation. IEEE (2018)
8. Goldberg, D.E.: Genetic algorithms as a computational theory of conceptual design. In: Rzevski, G., Adey, R.A. (eds.) Applications of Artificial Intelligence in Engineering VI, pp. 3–16. Springer, Dordrecht (1991). https://doi.org/10.1007/978-94-011-3648-8_1

9. Gulwani, S., Polozov, O., Singh, R., et al.: Program synthesis. Found. Trends® Program. Lang. **4**(1–2), 1–119 (2017)
10. Harman, M., Jia, Y., Langdon, W.B.: Babel Pidgin: SBSE can grow and graft entirely new functionality into a real world system. In: Le Goues, C., Yoo, S. (eds.) SSBSE 2014. LNCS, vol. 8636, pp. 247–252. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09940-8_20
11. 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. ACM, New York (2017)
12. 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. ACM, New York (2018)
13. Helmuth, T., Spector, L.: Detailed problem descriptions for general program synthesis benchmark suite. Technical report, University of Massachusetts Amherst, School of Computer Science (2015)
14. Helmuth, T., Spector, L.: General program synthesis benchmark suite. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1039–1046. ACM, New York (2015)
15. Hemberg, E., Kelly, J., O'Reilly, U.M.: On domain knowledge and novelty to improve program synthesis performance with grammatical evolution. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1039–1046. ACM (2019)
16. Hemberg, E., McPhee, N., O'Neill, M., Brabazon, A.: Pre-, in-and postfix grammars for symbolic regression in grammatical evolution. In: IEEE Workshop and Summer School on Evolutionary Computing 2008, pp. 18–22 (2008)
17. Jundt, L., Helmuth, T.: Comparing and combining lexicase selection and novelty search. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1047–1055. ACM, New York (2019)
18. Krawiec, K.: Behavioral Program Synthesis with Genetic Programming, vol. 618. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-27565-9
19. McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **SE-2**(4), 308–320 (1976)
20. O'Neill, M., Ryan, C., Nicolau, M.: Grammar defined introns: an investigation into grammars, introns, and bias in grammatical evolution. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 97–103. Morgan Kaufmann Publishers Inc., San Francisco (2001)
21. Rothlauf, F., Oetzel, M.: On the locality of grammatical evolution. In: Collet, P., Tomassini, M., Ebner, M., Gustafson, S., Ekárt, A. (eds.) EuroGP 2006. LNCS, vol. 3905, pp. 320–330. Springer, Heidelberg (2006). https://doi.org/10.1007/11729976_29
22. Ryan, C., Collins, J.J., Neill, M.O.: Grammatical evolution: evolving programs for an arbitrary language. In: Banzhaf, W., Poli, R., Schoenauer, M., Fogarty, T.C. (eds.) EuroGP 1998. LNCS, vol. 1391, pp. 83–96. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055930
23. Schweim, D., Thorhauer, A., Rothlauf, F.: On the non-uniform redundancy of representations for grammatical evolution: the influence of grammars. In: Ryan, C., O'Neill, M., Collins, J.J. (eds.) Handbook of Grammatical Evolution, pp. 55–78. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78717-6_3

24. Sobania, D., Rothlauf, F.: Teaching GP to program like a human software developer: using perplexity pressure to guide program synthesis approaches. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1065–1074. ACM, New York (2019)
25. Spector, L., Robinson, A.: Genetic programming and autoconstructive evolution with the push programming language. Genet. Program Evolvable Mach. **3**(1), 7–40 (2002). https://doi.org/10.1023/A:1014538503543