Assessing Similarity-Based Grammar-Guided Genetic Programming Approaches for Program Synthesis

Ning Tao[1,2][0000−0002−8154−547X], Anthony Ventresque[1,2][0000−0003−2064−1238], and Takfarinas Saber[1,3][0000−0003−2958−7979]

[1] Lero – the Irish Software Research Centre, Ireland
[2] School of Computer Science, University College Dublin, Dublin, Ireland
ning.tao@ucdconnect.ie, anthony.ventresque@ucd.ie
[3] School of Computer Science, National University of Ireland, Galway, Ireland
takfarinas.saber@nuigalway.ie

Abstract. Grammar-Guided Genetic Programming is widely recognised as one of the most successful approaches for program synthesis, i.e., the task of automatically discovering an executable piece of code given user intent. Grammar-Guided Genetic Programming has been shown capable of successfully evolving programs in arbitrary languages that solve several program synthesis problems based only on a set of input-output examples. Despite its success, the restriction on the evolutionary system to only leverage input/output error rate during its assessment of the programs it derives limits its scalability to larger and more complex program synthesis problems. With the growing number and size of open software repositories and generative artificial intelligence approaches, there is a sizeable and growing number of approaches for retrieving/generating source code based on textual problem descriptions. Therefore, it is now, more than ever, time to introduce G3P to other means of user intent (particularly textual problem descriptions). In this paper, we would like to assess the potential for G3P to evolve programs based on their similarity to particular target codes of interest (obtained using some code retrieval/generative approach). We particularly assess 4 similarity measures from various fields: text processing (i.e., FuzzyWuzzy), natural language processing (i.e., Cosine Similarity based on term frequency), software clone detection (i.e., CCFinder), plagiarism detector(i.e., SIM). Through our experimental evaluation on a well-known program synthesis benchmark, we have shown that G3P successfully manages to evolve some of the desired programs with three of the used similarity measures. However, in its default configuration, G3P is not as successful with similarity measures as with the classical input/output error rate at evolving solving program synthesis problems.

Keywords: Program Synthesis · Grammar-Guided Genetic Programming · Code Similarity · Textual Description · Text To Code

1 Introduction

Genetic Programming (GP [16]) is an efficient approach to evolve code using high-level specifications, hence it is the most popular approach to tackle program synthesis problems (i.e., the task of automatically discovering an executable piece of code given user intent) for software engineering [25, 24] and testing [26]. Various GP systems with different representations have been designed over time to tackle the diverse program synthesis problems.

PushGP [22] is one of the most efficient GP systems. PushGP evolves programs in the specially purpose-designed Push language. (i.e., a stack-based language designed specifically for program synthesis task). In Push, every variable type (e.g. strings, integers, etc.) has its own stack, which facilitates the genetic programming process. Despite its efficiency, PushGP's dependence to Push (a language that is not commonly used in practice and that is hard to interpret) hinders its exploitability and lowers our ability to improve upon it.

Grammar-Guided Genetic Programming (G3P [7]) system is another efficient GP system that evolves programs based on a specified grammar syntax. Besides its efficiency at solving program synthesis problems, the use of a syntax grammar enables G3P to produce programs that are syntactically correct with respect to any arbitrary programming languages definable through a grammar. The use of a grammar makes G3P particularly easy to move from one system to another and to adapt from one language to another [7]. This flexibility elevates G3P to be widely recognised as one of the most successful program synthesis approaches.

A recent comparative study [6] has evaluated the ability of both G3P and PushGP to solve several program synthesis problems from a well-studied program synthesis benchmark [11, 10] based only on a set of input-output examples. The study found that G3P achieves the highest success rate at finding correct solutions when it does find any. The study also found that PushGP is able to find correct solutions for more problems than G3P, but PushGP's success rate for most of the problems was very low. However, despite G3P's and PushGP's successes, the restriction on the evolutionary systems to only leverage the input/output error rate during their assessment of the programs they derive limits their scalability to larger and more complex program synthesis problems.

Following on the big data trend [4] and the growing number and size of open software repositories (i.e., databases for sharing and commenting source code) and generative artificial intelligence approaches (generative deep learning) there is a sizeable and growing number of approaches for retrieving/generating source code based on textual problem descriptions. Therefore, it is now, more than ever, time to introduce G3P to other means of user intent (particularly textual problem descriptions). Code retrieval and code generation techniques might output several incomplete snippets or not fully fit for purpose codes–which often makes them impossible to exploit in their form. Therefore, in this work, we propose an approach whereby such code guides the search process towards programs that are similar.

In this paper, we would like to assess the potential for G3P to evolve programs based on their similarity to particular target codes of interest which would have

been retrieved or generated using some particular text to code transformation. We particularly assess 4 similarity measures from various fields: text processing (i.e., FuzzyWuzzy), natural language processing (i.e., Cosine Similarity), software clone detection (i.e., CCFinder), plagiarism detector(i.e., SIM). The ultimate goal is the ability to identify the most suitable program similarity measure to guide the program synthesis search/evolutionary process when introducing code retrieval/generation from textual problem descriptions.

Through our experimental evaluation on a well-known program synthesis benchmark, we show that G3P successfully manages to evolve some of the desired programs with three of the used similarity measures. However, in its default configuration, G3P is not as successful with similarity measures as with the classical input/output error rate at evolving solving program synthesis problems. Therefore, in order to take advantage of textual problem descriptions and their subsequent text to code approaches in G3P, we need to either design better-fitted similarity measures, adapt our evolutionary operators to take advantage of program similarities, and/or combine similarity measures with the traditional input/output error rate.

The rest of the paper is structured as follows: Section 2 summarises the background and work related to our study. Section 3 describes our approach and details the similarity metrics used as code similarity in our evaluation. Section 4 details our experimental setup. Section 5 reports and discusses the results of our experiments. Finally, Section 6 concludes this work and discusses our future study.

2 Background and Related Work

In this section we present the material which forms our research background.

2.1 Genetic Programming

Genetic programming (GP) is an evolutionary approach that enables us to devise programs. GP starts with a population of random programs (often not very fit for purpose), and iteratively evolves it using operators analogous to natural genetic processes (e.g., crossover, mutation, and selection). Over the years, a variety of GP systems have been proposed–each with its specificity (e.g., GP [16], Linear GP [2], Cartesian GP [19]).

2.2 Grammar-Guided Genetic Programming

While there is a variety of GP systems, G3P is among the most successfully GP systems. What is unique to G3P is its use of a grammar as a guideline for syntactically correct programs throughout the evolution. Grammars are widely used due to their flexibility as they can be defined outside of the GP system to represent the search space of a wide range of problems including program synthesis [20], evolving music [17], managing traffic systems [31] evolving aircraft

models [3] and scheduling wireless communications [29, 18, 28, 30, 27]. Grammar-Guided Genetic Programming is a variant of GP that use grammar as the representation with most famous variants are Context-Free Grammar Genetic Programming (CFG-GP) by Whigham [33] and grammatical evolution [21].

The G3P system proposed in [7] puts forward a composite and self-adaptive grammar to address different synthesis problems, which solved the limitation of grammar that has to be tailored/adapted for each problem. In [7], several small grammars are defined–each for a data type that defines the function/program to be evolved. Therefore, G3P is able to reuse these grammars for different problems while keeping the search space small by not including unnecessary data types.

2.3 Problem Text Description to/from Source Code

The ability to automatically obtain source code from textual problem descriptions or explain concisely what a block of code is doing have challenged the software engineering community for decades.

The former (i.e., source code from textual description) was aimed at automating the software engineering process with a field mostly divided into two parts: (i) Program Sketching which attempts to lay/generate the general code structure and let either engineers or automated program generative approaches fill the gaps (e.g., [14]), and (ii) Code Retrieval which seeks to find code snippets that highly match the textual description of the problem in large code repositories.

The latter (i.e., textual description from source code) was mostly to increase the readability of source code and assist software engineering with their debugging, refactoring, and porting tasks. Several works have attempted to either provide meaningful comments for specific lines/blocks (e.g., [13]) or to generate brief summaries for the source code (e.g., [1]).

3 Similarity-Based G3P

In this section, we report on how similarity-based G3P perform on selected three problems from [10]. The G3P system in [7] uses error rate fitness function based on given input and output data for evolving the next generation, while similarity-based G3P presented in this section uses code similarity value to the given correct program.

3.1 Proposed Approach

Our ultimate goal is to exploit textual descriptions of user intent in the program synthesis process in combination with current advances in code retrieval/generation (even if such techniques potentially generate multiple incomplete or not fully fit for purpose programs) to guide the search process of G3P. To this end, in this work, we devise a similarity-based G3P system, which uses code similarity to evaluate the fitness of evolved programs against a target source code instead of input/output error rate. The focus of this particular work is not on generating

target source code but on (i) assessing the capability of G3P to evolve programs using similarity measures and (ii) identifying the most suitable measure.

3.2 Program Similarity Assessment Approaches

Measuring similarity between source code is a fundamental activity in software engineering. It has multiple applications including duplicate/clone code location, plagiarism detection, code search, security bugs scanning, vulnerability/bugs identification [9] and code recommendation [12]. There have been proposed dozens of similarity detection algorithms since the last few decades, which can be classified into metrics, text, token, tree, and graph-based approaches based on the representation [23]. We selected four top-ranked similarity measures to evaluate their code synthesis proneness when used within G3P.

Cosine Similarity In addition to the standard code similarity detector, we also used cosine similarity to measure the similarity between two source codes. The following steps illustrate how we measured similarity using cosine similarity:

1. Prepossessing: The source program is tokenized by removing indentation information, including white spaces, brackets, newline characters, and other formatting symbols. Arithmetic operators and assignment symbols were kept as they can provide meaningful structural information.
2. Frequency Computation: For each token sequence of the source program, we compute the frequency of each token.
3. Cosine Similarity Computation: We calculate the similarity score with the cosine formula based on the token frequencies of each source code.

FuzzyWuzzy [5] is a string matching open-source python library based on difflib python library. It uses Levenshtein Distance to calculate the differences between sequences. The library contains different similarity functions including $TokenSortRatio$ and $TokenSetRatio$. Ragkhitwetsagul et al. [23] surprisingly found that the string matching algorithm also works pretty well for measuring code similarity. $TokenSortRatio$ function first tokenizes the string by removing punctuation, changing capitals to lowercase. After tokenization, it sorts the tokens alphabetically and then joins them together to calculate the matching score. While $TokenSetRatio$ takes out the common tokens instead of sorting them.

CCFinder [15] is a token-based clone detection technique designed for large-scale source code. The technique detects the code clone with four steps:

1. Lexical Analysis: Generates token sequences from the input source code files by applying a lexical rule of a particular programming language. All source files are tokenized into a single sequence to detect the code clone with multiple files. White spaces and new line characters are removed to detect clone codes with different indentation rules but with the same meaning.

2. Transformation: The system applies transformation rules on token sequence to format the program into a regular structure, allowing it to identify code clones even in codes written with different expressions. Furthermore, all identifiers (e.g., variables, constants, and types) are replaced with special symbols to detect clones with different variable names and expressions.
3. Clone Matching: The suffix-tree matching algorithm is used to compute the matching of the code clones.
4. Formatting: Each clone pair is reported with line information in the source file. This step also contains reformatting from the token sequence.

CCFinder was designed for large-scale programs. Since the codes involved in our evaluation are elementary, the following modifications and simplifications are made to the original tool:

- Given that we are only interested in obtaining a similarity score between two pieces of code, we divide the length of the code clone by the maximum between the lengths of the source files:

$$Sim(x, y) = \frac{Len(Clone(x, y))}{Max(Len(x), Len(y))} \tag{1}$$

  where $Clone(x, y)$ denotes the longest code clone between x and y.
- The matching of code clones using the suffix-tree matching algorithm is simplified by getting the length of the longest common token sequence using a 2D matrix (each dimension representing the token sequence).
- The mapping information between the token sequence and the source code is removed since reporting the line number is no longer needed in our study.

SIM [8] is a software tool for measuring the structural similarity between two C programs to detect plagiarism in the assignment for lower-level computer science courses. It is also a token-based plagiarism detection tool that uses a string alignment technique to measure code similarity.

The approach comprises two main functions, generating tokens with formatting and calculating the similarity score using alignment. Each source file is first passed through a lexical analyzer to generate a token sequence. Like the common plagiarism detection system, the source code is formatted to standard tokens with white space removal representing arithmetic or logical operators, different symbols, constant or identifiers. After tokenization, the token sequence of the second program is divided into multiple sections, each representing a piece of the original program. These sections are then aligned with the token sequence of the first source code separately, which allows the tool to detect the similarity even the program is plagiarised by modifying the order of the functions.

4 Experiment Setup

4.1 General Program Synthesis Benchmark Suite

Helmuth and Spector [11, 10] introduced a set of program synthesis problems. These problems were based on coding problems that might be found in intro-

ductory computer science courses. Helmuth and Spector provide a textual description as well as two sets of input/output pairs for both training and testing during the program synthesis process. Table 1 describes the characteristics of each of the program synthesis problems considered in our evaluation.

Table 1. Description and characteristics of the selected program synthesis problems

Problem	Textual Description	# Input/Output Pair	
		Training	Testing
Number IO	Given an integer and a float, print their sum.	25	1000
Median	Given 3 integers, print their median.	100	1000
Smallest	Given 4 integers, print the smallest of them.	100	1000

4.2 Target Programs

To evolve our programs through G3P, we consider an oracle that computes the similarity measure of each evolved program to a target program code obtained using some text to code transformation. In this work, we wish to focus our analysis on the similarity measures and reduce the varying elements in our experiments (particularly in terms of ability to obtain a target program of good quality). Therefore, we consider the theoretical case where the oracle is aware of a code that solves the problem, but it is only reporting the similarity of the evolved code to it. While this assumption is not applicable in real life (i.e., if we know the correct code, then the problem is already solved without requiring any evolution), we hope to get enough insight from it on the capability of G3P to reproduce a program only based on a similarity measure.

Listings 1.1, 1.2, and 1.3 depict the target programs for the oracle assessment of program similarity for Number IO, Smallest, and Median respectively.

```
def numberIO(int1, float1):
    result = float(int1 + float1)
    return result
```

Listing 1.1. Target program for Number IO

```
def smallest(int1, int2, int2, int3):
    result = min(int1,min(int2,min(int3,int4)))
    return result
```

Listing 1.2. Target program for Smallest

```
def median(int1, int2, int3):
    if int1 > int2:
        if int1 < int3:
            median = int1
        elif int2 > int3:
            median = int2
        else:
            median = int3
    else:
        if int1 > int3:
```

```
            median = int1
        elif int2 < int3:
            median = int2
        else:
            median = int3
    return median
```

Listing 1.3. Target program for Median

4.3 G3P Parameter Settings

In our evaluation, we use the same parameter settings as those defined for G3P [7]. We only introduce a unique varying element (i.e., the fitness function based on a particular similarity measure). We repeat our evaluations 30 times for each problem and each G3P version (each version with its specific similarity measure). The general settings for the G3P system are shown in Table 2.

Table 2. Experiment parameter settings

Parameter	Setting	Parameter	Setting
Runs	30	Mutation probability	0.05
Generation	200	Node limit	250
Population size	1000	Variable per type	3
Tournament size	7	Max execution time	1 s
Crossover probability	0.9		

5 Results

In this section, we report and discuss the results of our evaluations. First, we start by comparing the performance of G3P using each of the similarity measures, then we compare them against the traditional error-rate based G3P.

5.1 Comparison of Similarity Measures

The result of the similarity-based G3P is reported in this subsection. The goal of this experiment is to assess G3P's ability to evolve a program solving a program synthesis problem (only known to an oracle) based on the similarity measure.

Figure 1 shows the number of runs (out of 30) where G3P manages to evolve the correct program for each of the program synthesis problems while using one of the four considered similarity measures as the fitness function.

We see from Figure 1 that G3P was able to able to evolve the correct programs for Number IO and Smallest at least once with Cosine, CCFinder and SIM. However, G3P did not manage to evolve any correct program for Median. G3P manages to find the correct program for Number IO in most runs (i.e., 27 out of

30) while using Cosine Similarity. However, the same program fails to find any correct program for Smallest. Similarly, G3P manages to find the correct program with Smallest in 17 runs out of 30 while using SIM, but the same program fails to find any correct program for Number IO. Alternatively G3P with CCFinder finds correct programs for both Number IO and Smallest, but in fewer runs. Overall, we could say that G3P has the potential to evolve programs for synthesis problems using similarity measures. However, there is no similarity measure that seems to work better than the rest and we need to consider combining similarity measures to increase the effectiveness of the approach.

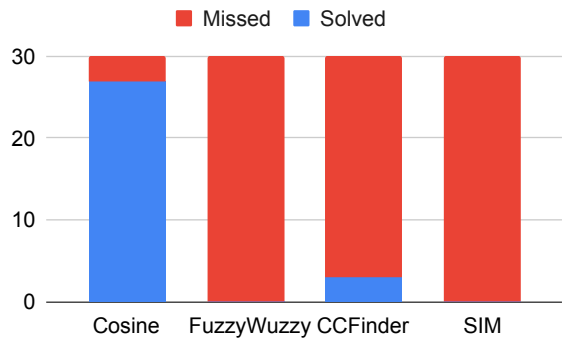
5.2 Comparison Against Error Rate-Based G3P

While we have seen that G3P has the capability to evolve correct programs for some program synthesis, we would like to assess how efficient is this process at evolving correct programs in comparison with the use of input/output error rate.

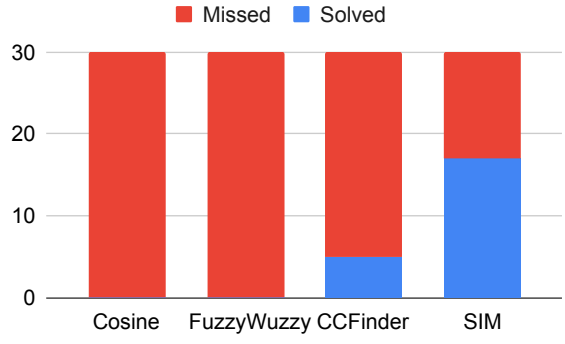
Figure 2 shows the performance of G3P with the input/output error rate to evolve correct programs for each of the considered programs over 30 distinct runs. We see that G3P with input/output error rate is capable to evolve correct problems to all the considered program synthesis problems. Furthermore, it is also capable of finding a correct program in more runs than the different G3P approaches using any similarity measure. Therefore, while we have seen that similarity measures seem promising to guide the G3P search for correct programs to program synthesis problems, they are not reaching the performance level of the traditional input/output error rate. This difference could be explained by the long amount of research that has been carried out to refine and optimise the G3P process with input/output error rate (particularly in terms of designing fit for purpose crossover and mutation operators). Therefore, one potential approach could be to use both the similarity objectives and the error-rate to guide the evolutionary process in a multi-objective approach [32].

6 Conclusion and Future Work

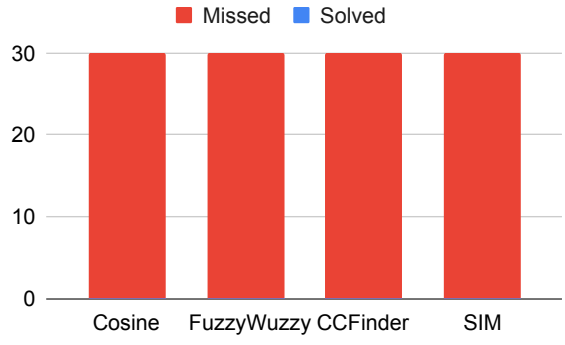
In this paper, we have assessed the potential for G3P to evolve programs based on their similarity to particular target codes of interest. The ultimate goal of this work is the ability to exploit textual descriptions of program synthesis problems as a guide to the evolutionary process in place of the traditional input/output error rate. We particularly assessed the capability of G3P to evolve correct programs using 4 similarity measures from various fields (i.e., Cosine, FuzzyWuzzy, CCFinder, and SIM). Our experimental evaluation on a well-known benchmark dataset has shown that G3P is able to evolve correct programs for some of the considered program synthesis problems. However, we have found that the performance of G3P using similarity measures is lower than G3P with the traditional input/output error rate. Out future work will focus on trying to improve the performance of G3P to take full advantage of such similarity measures alongside the traditional input/output error rate.



(a) Number IO



(b) Smallest



(c) Median

Fig. 1. Number of iterations (out of 30) where G3P manages or fails to evolve the target program with each of the similarity measures.

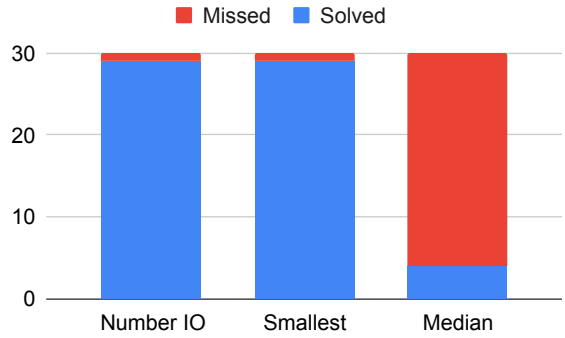


Fig. 2. Number of iterations (out of 30) where G3P with input/output error rate manages/fails to evolve the target program.

Acknowledgement: Supported, in part, by Science Foundation Ireland grant 13/RC/2094_P2.

References

1. Alexandru, C.V.: Guided code synthesis using deep neural networks. In: ACM SIGSOFT. pp. 1068–1070 (2016)
2. Brameier, M., Banzhaf, W., Banzhaf, W.: Linear genetic programming, vol. 1. Springer (2007)
3. Byrne, J., Cardiff, P., Brabazon, A., et al.: Evolving parametric aircraft models for design exploration and optimisation. Neurocomputing **142**, 39–47 (2014)
4. Ciritoglu, H.E., Saber, T., Buda, T.S., Murphy, J., Thorpe, C.: Towards a better replica management for hadoop distributed file system. In: IEEE BigData Congress (2018)
5. Cohen, A.: Fuzzywuzzy: Fuzzy string matching in python (2011)
6. Forstenlechner, S.: Program synthesis with grammars and semantics in genetic programming. Ph. D. dissertation (2019)
7. Forstenlechner, S., Fagan, D., Nicolau, M., O'Neill, M.: A grammar design pattern for arbitrary program synthesis problems in genetic programming. In: EuroGP. pp. 262–277 (2017)
8. Gitchell, D., Tran, N.: Sim: a utility for detecting similarity in computer programs. ACM SIGCSE Bulletin **31**(1), 266–270 (1999)
9. Hartmann, B., MacDougall, D., Brandt, J., Klemmer, S.R.: What would other programmers do: suggesting solutions to error messages. In: SIGCHI. pp. 1019–1028 (2010)
10. Helmuth, T., Spector, L.: Detailed problem descriptions for general program synthesis benchmark suite. University of Massachusetts Amherst (2015)
11. Helmuth, T., Spector, L.: General program synthesis benchmark suite. In: GECCO. pp. 1039–1046 (2015)
12. Holmes, R., Murphy, G.C.: Using structural context to recommend source code examples. In: ICSE. pp. 117–125 (2005)

13. Hu, X., Li, G., Xia, X., Lo, D., Jin, Z.: Deep code comment generation. In: IEEE/ACM ICPC. pp. 200–20010 (2018)
14. Jeon, J., Qiu, X., Foster, J.S., Solar-Lezama, A.: Jsketch: sketching for java. In: ESEC/FSE. pp. 934–937 (2015)
15. Kamiya, T., Kusumoto, S., Inoue, K.: Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering **28**(7), 654–670 (2002)
16. Koza, J.R., et al.: Genetic programming II, vol. 17. MIT press Cambridge, MA (1994)
17. Loughran, R., McDermott, J., O'Neill, M.: Tonality driven piano compositions with grammatical evolution. In: IEEE CEC. pp. 2168–2175 (2015)
18. Lynch, D., Saber, T., Kucera, S., Claussen, H., O'Neill, M.: Evolutionary learning of link allocation algorithms for 5g heterogeneous wireless communications networks. In: GECCO. pp. 1258–1265 (2019)
19. Miller, J.F., Harding, S.L.: Cartesian genetic programming. In: GECCO. pp. 2701–2726 (2008)
20. O'Neill, M., Nicolau, M., Agapitos, A.: Experiments in program synthesis with grammatical evolution: A focus on integer sorting. In: CEC. pp. 1504–1511 (2014)
21. O'Neill, M., Ryan, C.: Grammatical evolution: Evolutionary automatic programming in a arbitrary language, volume 4 of genetic programming (2003)
22. Pantridge, E., Spector, L.: Pyshgp: Pushgp in python. In: GECCO. pp. 1255–1262 (2017)
23. Ragkhitwetsagul, C., Krinke, J., Clark, D.: A comparison of code similarity analysers. Empirical Software Engineering **23**(4), 2464–2519 (2018)
24. Saber, T., Brevet, D., Botterweck, G., Ventresque, A.: Is seeding a good strategy in multi-objective feature selection when feature models evolve? IST (2017)
25. Saber, T., Brevet, D., Botterweck, G., Ventresque, A.: Milpibea: Algorithm for multi-objective features selection in (evolving) software product lines. In: EvoCOP. pp. 274–280 (2020)
26. Saber, T., Delavernhe, F., Papadakis, M., O'Neill, M., Ventresque, A.: A hybrid algorithm for multi-objective test case selection. In: IEEE CEC (2018)
27. Saber, T., Fagan, D., Lynch, D., Kucera, S., Claussen, H., O'Neill, M.: A hierarchical approach to grammar-guided genetic programming the case of scheduling in heterogeneous networks. In: TPNC. pp. 118–134 (2018)
28. Saber, T., Fagan, D., Lynch, D., Kucera, S., Claussen, H., O'Neill, M.: Multi-level grammar genetic programming for scheduling in heterogeneous networks. In: EuroGP. pp. 118–134 (2018)
29. Saber, T., Fagan, D., Lynch, D., Kucera, S., Claussen, H., O'Neill, M.: Hierarchical grammar-guided genetic programming techniques for scheduling in heterogeneous networks. In: CEC (2020)
30. Saber, T., Fagan, D., Lynch, D., Kucera, S., Claussen, H., O'Neill, M.: A multi-level grammar approach to grammar-guided genetic programming: the case of scheduling in heterogeneous networks. GPEM pp. 1–39 (2019)
31. Saber, T., Wang, S.: Evolving better rerouting surrogate travel costs with grammar-guided genetic programming. In: IEEE CEC. pp. 1–8 (2020)
32. Tao, N., Ventresque, A., Saber, T.: Multi-objective grammar-guided genetic programming with code similarity measurement for program synthesis. In: IEEE CEC (2022)
33. Whigham, P.A.: Grammatical bias for evolutionary learning. (1997)