2023 IEEE Latin American Conference on Computational Intelligence (LA-CCI) | 979-8-3503-4807-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/LA-CCI58595.2023.10409432

Many-objective Grammar-guided Genetic Programming with Code Similarity Measurement for Program Synthesis

Ning Tao*, Anthony Ventresque†‡, and Takfarinas Saber†§
*School of Computer Science, University College Dublin, Dublin, Ireland
Email: ning.tao@ucdconnect.ie
†Lero – the Irish Software Research Centre
‡School of Computer Science and Statistics, Trinity College Dublin, Dublin, Ireland
Email: anthony.ventresque@tcd.ie
§School of Computer Science, University of Galway, Galway, Ireland
Email: takfarinas.saber@universityofgalway.ie

***Abstract*—The approach known as Grammar-Guided Genetic Programming (G3P) is widely acknowledged as a highly effective method for program synthesis, which involves automatically generating code based on high-level formal specifications. Given the increasing quantity and scale of open software repositories and generative artificial intelligence techniques, there exists a significant range of methods for retrieving or generating source code using textual problem descriptions. Therefore, in light of the prevailing circumstances, it becomes imperative to introduce G3P into alternative means of user intent, with a specific focus on textual depictions. In our previous work, we assessed the potential for G3P to evolve programs based on bi-objectives that combine the similarity to the target program using four different similarity measures and the traditional input/output error rate. The result showed that such an approach improved the success rate for generating correct solutions for some of the considered problems. Nevertheless, it is noteworthy that despite the inclusion of various similarity measures, there is no single measure that uniformly improves the success rate of G3P across all problems. Instead, certain similarity measures exhibit effectiveness in addressing specific problems while demonstrating limited efficacy in others. In this paper, we would like to expand the bi-objective framework with different similarity measures to a many-objective framework to enhance the general performance of the algorithm to more range of problems. Our experiments show that compared to the bi-objective G3P (BOG3P), the Many-objective G3P (MaOG3P) approach could achieve the best result of all BOG3P algorithms with different similarity measures.**

***Index Terms*—Program Synthesis, Grammar-Guided Genetic Programming, Code Similarity, Many-Objective Optimisation**

I. Introduction

The field of artificial intelligence (AI) has witnessed remarkable advancements, revolutionizing various domains ranging from healthcare to finance. Among its transformative applications, automatic programming stands out as a promising area that holds tremendous potential for revolutionizing software development. This emerging field leverages AI techniques to analyse problem domains, understand user intent, and produce executable code. In recent years, the proliferation of open-source software repositories and the advent of generative AI approaches have spurred a surge in the development of methods for retrieving and generating source code based on textual problem descriptions. These advancements have paved the way for new possibilities in software development, enabling developers to focus more on high-level problem-solving and design rather than the intricacies of coding.

Program synthesis, one of the best ways of automatic programming, aims to automatically generate executable programs from high-level specifications. Numerous algorithms have been suggested for program synthesis utilising diverse programming languages and forms of user intent: Beltramelli introduced pix2code [1], a Convolutional Neural Network (CNN) based system that utilises user-provided graphical user interface screenshot images to generate web development interface code, specifically HTML/CSS. Bassil and Alwani [2] devised a method that employs a Context-Free Grammar (CFG) parser and a finite-state machine-based lexical analyser to generate HTML code. Niaz et al. [3] introduced an algorithm that maps transitions and states, enabling the generation of Java code from Unified Modeling Language (UML) diagrams and state patterns. In the study by Tao et al. [4]–[6], a G3P system is presented that produces a Python program by utilising textual task descriptions along with input/output examples. Boutekkouk [7] introduced a system that utilises Visual Basic and Action Language to generate C codes by effectively leveraging UML diagrams. However, despite the range of approaches available, Genetic Programming (GP) [8] remains the leading contender in tackling program synthesis problems [9].

GP evolves programs by starting with an unfit population and gradually transforming them through operations inspired by natural genetic processes [8]. Among the notable GP systems, PushGP [10] stands out as one of the most effective. It generates programs using the stack-based language called Push, which is specifically designed for program synthesis

tasks. While PushGP has demonstrated its effectiveness, its reliance on a specialised language poses challenges when it comes to real-world scenarios where other programming languages are more prevalent. To overcome this limitation, G3P [11] integrates language grammar into GP, enabling constraints on the syntactic structure of programs. Nevertheless, even though G3P and PushGP have achieved notable accomplishments, their scalability is constrained by the limitation of solely relying on the input/output error rate for program evaluation when addressing more substantial and intricate program synthesis problems. Tao et al. [5] extended the G3P system to a bi-objective framework by leveraging both a similarity measure and input/output error rate as objectives to guide the evolutionary search process. Combining both objectives, BOG3P improved the success rate of program synthesis problems. However, the performance of BOG3P is intricately linked to the choice of similarity measure, as certain measures may excel in addressing specific problems but exhibit poor performance in others.

This paper aims to extend the BOG3P approach to incorporate a many-objective framework for program evolution, considering various similarity measures and input/output examples. Through an extensive experimental evaluation on a widely recognised program synthesis benchmark, we demonstrate the efficacy of MaOG3P in achieving superior results for each BOG3P variant across the considered problem set.

The remaining sections are organised as follows: Section II provides an overview of the background. Section III elucidates the specific metrics employed for evaluating code similarity. Section IV delineates our many-objective approach in detail. Section V expounds upon the experimental setup. Section VI presents an in-depth analysis and discussion of the outcomes. Finally, Section VII encapsulates the overall conclusions and the future work.

II. Background and Related Work

A. Genetic Programming

GP is an evolutionary methodology that allows for the generation of programs aimed at accomplishing specific tasks. GP initiates with a population of randomly generated programs, which may initially lack fitness for the intended purpose. Through iterative processes utilising operators akin to natural genetic mechanisms (such as crossover, mutation, and selection), GP progressively evolves these programs to discover improved solutions. Throughout the years, diverse GP systems have been proposed, each possessing its unique characteristics (e.g., GP [8], Linear GP [12], Cartesian GP [13]).

B. Grammar-Guided Genetic Programming

While a variety of GP systems exist, G3P stands out as one of the most successful. What distinguishes G3P is its utilisation of grammar as a guiding principle to ensure syntactically correct program evolution. Grammars offer great flexibility as they can be defined externally to the GP system, allowing representation of a broad range of problems, including program synthesis [14], traffic systems management [15], and wireless communications scheduling [16]–[20]. G3P encompasses various GP variants, with notable examples being Context-Free Grammar Genetic Programming (CFG-GP) introduced by Whigham [21] and grammatical evolution [14].

The G3P system, as introduced by Forstenlechner [11], introduces a composite and self-adaptive grammar that effectively addresses various synthesis problems. This innovation overcomes the limitation of having to tailor or adapt grammar for each specific problem. By defining several small grammars, each corresponding to a data type that defines the function or program to be evolved, G3P enables the reuse of these grammars across different problems. This approach ensures a compact search space by excluding unnecessary data types.

C. Many-Objective Optimisation

Many-objective optimisation (MaOO) pertains to the concurrent optimisation of more than three objective functions. Given that the evaluation of evolved code in program synthesis problems can be assessed from diverse perspectives, we propose that it is more appropriate to conceptualise it as a many-objective optimisation problem.

In the MaOO problem, the output is a set of non-dominated solutions, which can be defined as follows: Let S be the set of all evolved programs for a given program synthesis problem. For all $x \in S$, $O = [O_1(x), ..., O_k(x)]$ is the vector containing the k objective values for the solution x. It is said that a program x_1 dominates another program x_2 (also written as $x_1 \succ x_2$), if and only if $\forall i \in \{1, ..., k\}, O_i(x_1) \leqslant O_i(x_2)$ and $\exists t \in \{1, ..., k\}$ such that $O_i(x_1) < O_i(x_2)$. We also say that x_i is a non-dominated program if there is no other program x_j that dominates x_i. The set of all non-dominated programs form what is called a Pareto front: in this set, it is impossible to find any program better in all objectives than any of the other programs in the set.

III. Program Similarity Detection Approaches

The measurement of similarity in source code holds various applications, such as identifying duplicate code, detecting plagiarism, enabling code search, facilitating the discovery of similar bug fixes [22], and offering code recommendations [23]. In this study, we have chosen four highly ranked similarity measures from [24] to assess their effectiveness in code synthesis when employed within the G3P framework.

A. FuzzyWuzzy

FuzzyWuzzy, an open-source Python library [25], is designed for string matching and is built upon the difflib Python library. The library contains different similarity functions, including $TokenSortRatio$ and $TokenSetRatio$. In an unexpected discovery, Ragkhitwetsagul et al. [24] found that the string matching algorithm demonstrates remarkable efficacy in measuring code similarity. $TokenSortRatio$ function first tokenises the string by removing punctuation, and changing capitals to lowercase. After tokenisation, it sorts the tokens alphabetically and then joins them together to calculate the

matching score. In comparison, $TokenSetRatio$ takes out the common tokens instead of sorting them.

B. Cosine

In conjunction with the conventional code similarity detector, we also employed cosine similarity as a measure of similarity between two source codes. The subsequent steps outline our approach for measuring similarity using cosine similarity.

1) Preprocessing: The source program undergoes tokenisation, which involves the removal of indentation details, including whitespace, brackets, newline characters, and other formatting symbols.

2) Frequency Calculation: For every token sequence, we calculate the frequency associated with each token.

3) Cosine Similarity Computation: The similarity score between the two programs is determined by calculating the cosine value between their term frequency vectors, denoted as vectors A and B, as shown in Eq. 1.

$$\cos(\mathbf{A},\mathbf{B}) = \frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n}\mathbf{A}_i\mathbf{B}_i}{\sqrt{\sum_{i=1}^{n}(\mathbf{A}_i)^2}\sqrt{\sum_{i=1}^{n}(\mathbf{B}_i)^2}} \quad (1)$$

C. CCFinder

CCFinder, an advanced technique developed by Kamiya et al. [26], focuses on detecting code clones in large-scale source code using a token-based approach. The technique employs a four-step process for identifying code clones: (I) Lexical Analysis: Token sequences are generated from the input source code files by implementing the lexical rules specific to the programming language. (ii) Transformation: The system utilises transformation rules on the token sequence to standardise the program's structure, enabling the identification of code clones, even in code written with diverse expressions. (iii) Clone Matching: The computation of code clone matches is performed using the suffix-tree matching algorithm. (iv) Formatting: Every clone pair is reported along with line information within the source file.

CCFinder was originally designed to cater to large-scale programs. However, considering the simplicity of the codes involved in our evaluation, we have made the following modifications and simplifications to the original tool:

- Since our primary objective is to acquire a similarity score between two code fragments, we calculate this score by dividing the length of the code clone by the maximum length among the source files:

$$Similarity(x,y) = \frac{Len(Clone(x,y))}{Max(Len(x),Len(y))} \quad (2)$$

  where $Clone(x,y)$ denotes the longest code clone between the codes x and y and $Len(x)$ denotes the length (in terms of number of characters) of the code x.
- To simplify the process of matching code clones using the suffix-tree matching algorithm, we utilise a 2D matrix representation (with each dimension representing a token sequence) to determine the length of the longest common token sequence.
- In our study, the mapping information between the token sequence and the source code is eliminated, as reporting the line number is no longer necessary.

D. SIM

SIM [27] is a software tool designed for plagiarism detection in lower-level computer science courses. It employs a token-based approach, utilizing a string alignment technique, to measure the structural similarity between two C programs and identify instances of plagiarism.

The approach consists of two key functions: token generation with formatting and similarity score calculation using alignment. To begin, each source file undergoes lexical analysis to produce a token sequence. Following tokenisation, the token sequence of the second program is segmented into multiple sections, each corresponding to a portion of the original program. These sections are individually aligned with the token sequence of the first source code, enabling the tool to identify similarities even if the program has been plagiarised by altering the function order.

IV. Proposed Approach

Our goal is to exploit textual (natural language) 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) alongside input/output tests to guide the search process of G3P.

In our work, we assume that we avail of an automated technique that is able to take a textual description of a problem in natural language and output a source code (potentially multiple incomplete or not fully fit-for-purpose programs). Such technique could either be based on (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., [28]), (ii) Code Retrieval which seeks to find code snippets that highly match the textual description of the problem from a large source code repositories, or (iii) Generative Pre-trained Transformer (GPT) with Large Language Models (LLMs), which generate code based on the textual description (e.g., ChatGPT or Github Copilot). We make such an assumption to reduce the varying elements in our study (i.e., the quality of the obtained code could vary) and focus purely on the exploitation of the obtained source code in the evolutionary process.

The source code obtained from the above text-to-code process can be considered as a target code against which we could assess the similarity of each program. As described in III, there exists various code similarity algorithms/measures. Assuming that the target code is of decent quality, we have previously shown that the "*similar*" the evolved program is to the target code, the better the evolved code–and thus leveraging a similarity measure as a fitness function (i.e., an additional objective) to guide the search process would help evolve the

correct code [4]. However, it was not clear what similarity measure would be the most suitable [5].

Therefore, in this work, we propose a novel many-objective G3P that combines input/output error rate and multiple similarity measures as objectives to assess the fitness of each evolved program.

Figure 1 shows an overview of our proposed approach. It is a many-objective extension to our previously proposed BOG3P system [5] by leveraging multiple code similarity measures at the same time. The idea is to generate or retrieve a code (or multiple target codes) using a text-to-code technique. The obtained code could then be used as a target code against which we assess the various similarity measures for each evolved program.

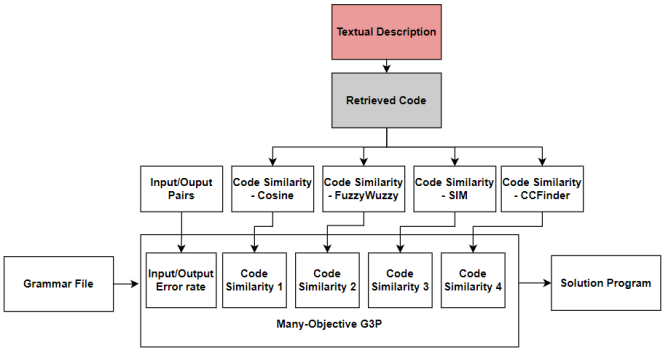

Fig. 1. Overview of our Many-Objective G3P system

Building on the BOG3P system, we propose a MaOG3P system that uses (i) standard input/output error rate (from [11]) and (ii) four code similarity measures (i.e., Cosine, FuzzyWuzzy, SIM, and CCFinder) as independent objectives to evaluate the fitness of each evolved program and guide the evolutionary search process.

In our previous work, we have defined four different similarity measures, corresponding to four different bi-objective G3P algorithms: (i) BOG3P$_{Cosine}$ - Cosine and input/output error rate, (ii) BOG3P$_{Fuzzy}$ - FuzzyWuzzy and input/output error rate, (iii) BOG3P$_{CCFinder}$ - CCFinder and input/output error rate, (iv) BOG3P$_{SIM}$ - SIM and input/output error rate.

It is important to highlight that although BOG3P and MaOG3P utilise different objectives to evaluate the fitness of evolved programs and guide the evolutionary process, we solely deem a program as correct when its input/output error rate reaches zero (i.e., correctly solves all the input/output cases in the Testing set). This approach is driven by the fact that we typically lack assurance regarding the quality of the target code employed for similarity measurement. Our reliance lies in the hope that the target code is either close to the correct program or shares certain code segments. In essence, if we already possessed the correct code, the problem would be solved without the need for an evolutionary search.

The process of the MaOG3P system commences by generating the initial population, which adheres to the chosen context-free grammar for the specific data type relevant to the given problem, as outlined in [11]. Subsequently, the system undergoes iterations involving many-objective tournament selection, crossover, mutation, evaluation of fitness values, and population update. This iterative process continues until the termination condition is met, which in our case is determined by the number of generations.

In this study, the tournament selector was adapted for many-objective purposes, allowing the selection of parents based on different objectives. Specifically, for half of the individuals, both parents were selected based only on the input/output error rate, which serves as the main objective in our evolutionary process. For the remaining individuals, the first parent was selected based on the main objective (input/output error rate), whereas the other parent was selected based on the different similarity measures–each similarity measure being employed one-fourth of the time.

The selection of individuals that can proceed to the next generation is a crucial operation in the GP system. Similar to the concept of parent selection, all objectives are considered for the next-generation selection process. However, in this case, half of the population is chosen based on the main objective, while the remaining half is determined by the four similarity objectives.

V. Experiment Setup

A. Problem Description

The program synthesis problems presented by Helmuth and Spector [29], [30] encompass a collection of coding problems commonly encountered in introductory computer science courses. These problems are accompanied by textual descriptions and two sets of input/output pairs, serving as training and testing data throughout the program synthesis procedure. A comprehensive overview of the characteristics associated with each program synthesis problem considered in our evaluation can be found in Table I.

TABLE I
Description of the Selected Program Synthesis Problems

Problem	Textual Description	# Input/output Pairs	
		Training	Testing
Number IO	Given an integer and a float, print their sum.	25	1000
Smallest	Given 4 integers, print the smallest of them.	100	1000
Median	Given 3 integers, print their median.	100	1000
String Lengths Backwards	Given a vector of strings, print the length of each string in the vector starting with the last and ending with the first.	100	1000
Negative To Zero	Given a vector of integers, return the vector where all negative integers have been replaced by 0.	200	1000

B. Target Program

In our G3P-based program evolution process, we incorporate an oracle that calculates the similarity measure between each evolved program and a target program code derived through a text-to-code transformation. In this study, we aim to primarily investigate the effectiveness of different similarity measures while minimizing the experimental variables, particularly in terms of obtaining a high-quality target program. Consequently, we consider a theoretical scenario where the

oracle is cognizant of a code that effectively solves the problem, but solely provides the similarity rating for the evolved code. Although this assumption does not align with real-world scenarios (since possessing the correct code would render the evolutionary process unnecessary), we anticipate that it will yield valuable insights into the ability of G3P to generate a program solely based on a similarity measure.

The target programs used for evaluating program similarity in the oracle assessment of Number IO, Smallest, Median, String Lengths Backwards, and Negative To Zero are illustrated in Listings 1, 2, 3, 4, and 5 correspondingly.

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

Listing 1. Target program for Number IO

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

Listing 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 3. Target program for Median

```
def StringLengthsBackwards(in0):
    temp = []
    for string in in0:
        temp.append(len(string))
    result = []
    i = len(temp) - 1
    while (i != -1):
        result.append(temp[i])
        i = i - 1
    return result
```

Listing 4. Target program for String Lengths Backwards

```
def negativeToZero(input_list):
    result=[]
    for number in input_list:
        if number < 0:
            result.append(0)
        else:
            result.append(number)
    return result
```

Listing 5. Target program for Negative To Zero

C. Parameter Setting

The general settings for the GP system are 30 Runs; 300 Generations (200 generations for Median, Number IO, and Smallest as in [29]); Population size of 1000; Tournament selection with a tournament size of 7; 0.9 Crossover probability and 0.05 mutation probability (We used the general crossover and mutation operators provided by the HeuristicLab tool); 3 variable for per type; and 1 second max execution time.

VI. Results

In this section, we assess the ability and performance of our proposed approach (i.e., MaOG3P) to evolve correct codes.

Table II shows the number of times (over 30 distinct runs) MaOG3P, G3P and the various BOG3P systems successfully evolved a correct code to each of the considered problems.

For more straightforward problems (i.e., Number IO and Smallest), MaOG3P gets similar results as the best BOG3P system (i.e., with CCFinder or SIM). MaOG3P successfully finds the correct solution in each of the 30 runs. This shows that combining multiple similarity measures in MaOG3P retains the performance of the best BOG3P (i.e., with a single similarity measure).

On the relatively more challenging problem (i.e., "Median"), MaOG3P finds a correct code in more than double the number of runs than G3P. Moreover, MaOG3P also outperforms all the BOG3P systems. This is a indication that combining multiple similarity measures might enable better performance than using the same similarity measures separately (i.e., in BOG3P).

In "Negative To Zero", MaOG3P evolved a correct code in 12 runs out of 30 and ranked 2^{nd} best performance after BOG3P with FuzzyWuzzy (13 runs out of 30). However, MaG3P is noticeably better than BOG3P with the rest of the similarity measures, and MaOG3P almost doubled the number of successful runs compared to G3P. This is a indication that even when MaOG3P is not achieving the best performance, it still achieves close to the best one.

In the problem String Lengths Backwards, MaOG3P successfully evolved a correct program as many times as the best BOG3P (i.e., with SIM). However, MaOG3P achieved its results while combining similarity measures which achieved (in BOG3P) worse results than G3P. This strengths the reliability of MaOG3P and reduces the need for a combinatorial 'tuning' of the best combination of similarity measures.

Overall, the MaOG3P system leverages the advantages offered by each variant of the BOG3P system, which utilises different similarity measures. This integration leads to the attainment of optimal or near-optimal results across all the considered problems. By incorporating many objectives into the evolutionary process, MaOG3P effectively explores the search space, capitalizing on the strengths of each similarity measure employed by the BOG3P system.

The successful outcomes achieved by MaOG3P highlight its ability to handle the complexity and variability inherent in program synthesis problems. By exploiting the strengths of different similarity measures, the system mitigates the limitations associated with relying solely on a single similarity metric. This flexibility allows MaOG3P to adapt to various problem

characteristics, leading to improved performance across a range of program synthesis tasks.

The superior performance of MaOG3P in comparison to the individual BOG3P systems underscores the significance of considering many objectives in program synthesis. By leveraging a diverse set of similarity measures, MaOG3P showcases its robustness and effectiveness in capturing both functional and structural aspects of program code. This comprehensive approach contributes to the overall success and reliability of MaOG3P in generating accurate and high-quality programs.

In summary, MaOG3P stands out as a powerful framework for program synthesis, combining the strengths of various similarity measures employed by the BOG3P system. The system's ability to achieve optimal results across a wide range of problems underscores its effectiveness and versatility in addressing the challenges of program synthesis in a comprehensive and adaptive manner.

TABLE II
NUMBER OF TIMES OUT OF 30 RUNS A CORRECT PROGRAM IS EVOLVED.

Problem	G3P	BOG3P *Cosine*	BOG3P *Fuzzy*	BOG3P *CCFinder*	BOG3P *SIM*	MaOG3P
Number IO	29	29	29	**30**	**30**	**30**
Smallest	29	28	28	**30**	**30**	**30**
Median	4	9	2	5	3	**10**
String Lengths Backwards	2	1	2	0	**3**	**3**
Negative to Zero	7	7	**13**	9	8	12

VII. CONCLUSION AND FUTURE WORK

The objective of this paper is to enhance the G3P approach by introducing a many-objective framework for program evolution. By incorporating different similarity measures and incorporating input/output examples, we aim to demonstrate the effectiveness of the many-objective G3P through an extensive experimental evaluation on a well-established program synthesis benchmark. Our results showcase the superior performance of the many-objective G3P over each variant of the BOG3P across the range of problems examined.

In our future research endeavours, we are committed to overcoming the limitations inherent in our proposed approach by incorporating advanced code generation techniques in conjunction with the MaOG3P framework. By integrating code generation methods into the evolutionary process, we anticipate significant improvements in the performance and effectiveness of the program synthesis process. Furthermore, we recognise the importance of exploring alternative code similarity detection algorithms to complement the existing approaches used in our study. To fully harness the knowledge acquired from the selected similarity algorithms, we also plan to adapt the evolutionary operators employed in our approach. By leveraging advanced code generation techniques, exploring alternative code similarity detection algorithms, and refining the evolutionary operators, we aim to unlock new possibilities and achieve superior results in program synthesis tasks.

Acknowledgment: partially supported by Science Foundation Ireland grant 13/RC/2094_P2 to Lero.

REFERENCES

[1] T. Beltramelli, "pix2code: Generating code from a graphical user interface screenshot," in *ACM SIGCHI*, 2018.
[2] Y. Bassil and M. Alwani, "Autonomic html interface generator for web applications," *arXiv*, 2012.
[3] I. A. Niaz, J. Tanaka *et al.*, "Mapping uml statecharts to java code." in *IASTED Conf. on Software Engineering*, 2004.
[4] N. Tao, A. Ventresque, and T. Saber, "Assessing similarity-based grammar-guided genetic programming approaches for program synthesis," in *OLA*. Springer, 2022.
[5] N. Tao, A. Ventresque, and T. Saber, "Multi-objective grammar-guided genetic programming with code similarity measurement for program synthesis," in *IEEE CEC*, 2022.
[6] N. Tao, A. Ventresque, and T. Saber, "Program synthesis with generative pre-trained transformers and grammar-guided genetic programming grammar," in *IEEE LACCI*, 2023.
[7] F. Boutekkouk, "Automatic systemc code generation from uml models at early stages of systems on chip design," *IJCA*, 2010.
[8] J. R. Koza *et al.*, *Genetic programming II*, 1994, vol. 17.
[9] D. Sobania, M. Briesch, and F. Rothlauf, "Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming," in *GECCO*, 2022.
[10] E. Pantridge and L. Spector, "Pyshgp: Pushgp in python," in *GECCO*, 2017.
[11] 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.
[12] M. Brameier, W. Banzhaf, and W. Banzhaf, *Linear genetic programming*, 2007.
[13] J. F. Miller and S. L. Harding, "Cartesian genetic programming," in *GECCO*, 2008.
[14] M. O'Neill and C. Ryan, "Grammatical evolution: Evolutionary automatic programming in a arbitrary language, volume 4 of genetic programming," 2003.
[15] T. Saber and S. Wang, "Evolving better rerouting surrogate travel costs with grammar-guided genetic programming," in *CEC*, 2020.
[16] D. Lynch, T. Saber, S. Kucera, H. Claussen, and M. O'Neill, "Evolutionary learning of link allocation algorithms for 5g heterogeneous wireless communications networks," in *GECCO*, 2019.
[17] T. Saber, D. Fagan, D. Lynch, S. Kucera, H. Claussen, and M. O'Neill, "A multi-level grammar approach to grammar-guided genetic programming: the case of scheduling in heterogeneous networks," *GPEM*, 2019.
[18] T. Saber, D. Fagan, D. Lynch, S. Kucera, H. Claussen, and M. O'Neill, "Multi-level grammar genetic programming for scheduling in heterogeneous networks," in *EuroGP*, 2018, pp. 118–134.
[19] T. Saber, D. Fagan, D. Lynch, S. Kucera, H. Claussen, and M. O'Neill, "A hierarchical approach to grammar-guided genetic programming the case of scheduling in heterogeneous networks," in *TPNC*, 2018.
[20] T. Saber, D. Fagan, D. Lynch, S. Kucera, H. Claussen, and M. O'Neill, "Hierarchical grammar-guided genetic programming techniques for scheduling in heterogeneous networks," in *CEC*, 2020.
[21] P. A. Whigham, "Grammatical bias for evolutionary learning." 1997.
[22] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do: suggesting solutions to error messages," in *SIGCHI Conference on Human Factors in Computing Systems*, 2010.
[23] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *ICSE*, 2005.
[24] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "A comparison of code similarity analysers," *ESE*, 2018.
[25] A. Cohen, "Fuzzywuzzy: Fuzzy string matching in python," 2011.
[26] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *TSE*, 2002.
[27] D. Gitchell and N. Tran, "Sim: a utility for detecting similarity in computer programs," *ACM Sigcse Bulletin*, 1999.
[28] J. Jeon, X. Qiu, J. S. Foster, and A. Solar-Lezama, "Jsketch: sketching for java," in *ESEC/FSE*, 2015.
[29] T. Helmuth and L. Spector, "General program synthesis benchmark suite," in *GECCO*, 2015.
[30] T. Helmuth and L. Spector, "Detailed problem descriptions for general program synthesis benchmark suite," *University of Massachusetts Amherst*, 2015.