Multi-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, Ireland
Email: ning.tao@ucdconnect.ie, anthony.ventresque@ucd.ie
†Lero – the Irish Software Research Centre
‡School of Computer Science, National University of Ireland, Galway, Ireland
Email: takfarinas.saber@nuigalway.ie

***Abstract*—Grammar-Guided Genetic Programming (G3P) 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. G3P 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 (potentially several partial snippets) 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). 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 all four considered similarity measures. However, in its default configuration, G3P is not as successful with similarity measures as it is with the classical input/output error rate when solving program synthesis problems. Therefore, we propose a novel multi-objective G3P approach that combines the similarity to the target program and the traditional input/output error rate. Our experiments show that compared to the error-based G3P, the multi-objective G3P approach could improve the success rate of specific problems and has great potential to improve on the traditional G3P system.**

***Index Terms*—Program Synthesis, Grammar-Guided Genetic Programming, Code Similarity, Multi-Objective Optimization**

I. Introduction

Automation is a method that allows a machine to take the place of a human in any process. Recent studies have looked at automation in computer programming, where the goal is to make a programmer's job easier by providing them with various tools and approaches for generating programming code based on their high level intent–a process known as *program synthesis*. Many algorithms have been proposed for automatic code generation using different programming languages, varying from procedural (e.g., C [1], [2]), to object-oriented (e.g., Python [3], and Java [4]), to scripting and markup (HTML [5], [6]). In addition to the diversity in the target language, there is also a large diversity in the automation approaches and in the forms of user intent: Bassil and Alwani [5] implemented a system that generates an HTML interface using a finite-state machine-based lexical analyzer and a Context-Free Grammar (CFG) parser. Beltramelli developed pix2code [6], a tool that generates HTML/CSS interface code based on GUI screenshot images using Convolutional Neural Network (CNN). Boutekkouk [2] used Visual Basic and Action Language to exploit Unified Modelling Language (UML) diagrams and generate system C code. Niaz et al. [4] proposed a state and transition mapping method to generates Java code from UML statecharts and state patterns. Li et al. [3] created AlphaCode, a python code generation system which uses large transformer language models and large-scale sampling to solve unseen competitive programming problems. However, despite this diversity, Genetic Programming (GP [7]) in its multiple representations, remains one of the most successful and popular approaches to tackle program synthesis problems based on input and output examples.

PushGP [8] 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 tasks). In Push, every variable type (e.g., strings, integers, etc.) has its own stack, facilitating 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 [9]) is another efficient GP system that evolves programs based on a specified grammar syntax. Besides its efficiency at solving program synthesis problems, using syntax grammar enables G3P to produce syntactically correct programs with respect to any arbitrary programming languages as long as there is a grammar to define them. The use of grammar makes G3P particularly easy to switch from one system to another and to adapt from one language to another [9]. This flexibility makes G3P widely recognised as one of the most successful program synthesis approaches.

A recent comparative study [10] has evaluated the ability of both G3P and PushGP to solve several program synthesis problems from a well-studied program synthesis benchmark [11], [12] 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 more significant and more complex program synthesis problems.

With the growing number and size of open software repositories (i.e., databases for sharing and commenting source code, such as Github and Stack Overflow, etc.) 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 [3], [13], [14]. 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 similar programs.

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 start by detailing and assessing the similarity-based G3P [15] whereby the classical input/output error rate is substituted with a similarity measure: FuzzyWuzzy (used in text processing), Cosine Similarity (natural language processing), CCFinder (software clone detection), or SIM (plagiarism detection).

We also extend the state-of-the-art G3P system to a multi-objective framework by leveraging both a similarity measure and input/output error rate as objectives to guide the evolutionary search process. By combining both objectives, we aim to improve the success rate of program synthesis problems when introducing code retrieval/generation from textual problem descriptions.

Our experimental evaluation on a well-known program synthesis benchmark shows that G3P was able to solve at least one problem a few times with each of the similarity measures–but with a lower success rate compared to using the classical input/output error rate. Our proposed multi-objective G3P successfully manages to improve the success rate over the state-of-the-art G3P on all the considered programs. However, there is no similarity measure that improves G3P's success rate on all problems. Therefore, to take advantage of textual problem descriptions and their subsequent text-to-code approaches in G3P, we need to either consider an ensemble of code similarity measures, design better-fitted similarity measures, or adapt the evolutionary operators to take advantage of program similarities.

The rest of the paper is structured as follows: Section II summarises the background and work related to our study. Section III describes the similarity metrics used as code similarity in our evaluation. Section IV presents our similarity-based approach. Section V details our experimental setup. Section VI reports and discusses the results of our experiments. Finally, Section VII concludes this work and discusses our future studies.

II. Background and Related Work

In this section we present the material which forms our research background and also the classical solutions found in the literature.

A. Genetic Programming

Genetic programming (GP) is an evolutionary approach that enables us to devise programs that perform particular tasks. GP starts with a population of random programs (often not very fit for purpose) and iteratively evolves them to search for better programs 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 [7], Linear GP [16], Cartesian GP [17]).

B. Grammar-Guided Genetic Programming

While various GP systems exist, G3P is among the most successful GP systems. What is unique to G3P is its use of 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 [18], managing traffic systems [19], and scheduling wireless communications [20]–[24]. Grammar-Guided Genetic Programming is a variant of GP that uses grammar as the representation, with most famous variant variants are Context-Free Grammar Genetic Programming (CFG-GP) by Whigham [25] and grammatical evolution [26].

The G3P system proposed in [9] presents 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. 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.

C. 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 has challenged the software engineering community for decades.

The former (i.e., source code from the 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., [27]), and (ii) Code Retrieval which seeks to find code snippets that highly match the textual description of the problem from a large source code repositories.

The latter (i.e., textual description from source code) mainly increased the readability of source code and assisted software engineers 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 summaries for the source code (e.g., [14]).

D. Multi-Objective Optimisation

Multi-objective Optimisation (MOO) involves the simultaneous optimisation of more than one objective function. Since the evolved code from the program synthesis problem can be evaluated from different perspectives (e.g., input/output error rate, similarity to target code, the structure of the code, etc.), we believe that it would be better modelled as a multi-objective optimisation problem.

In MOO 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.

In this study, we evolve the programs with two objectives (i.e., input/output error rate and the degree of similarity against a target code). Since there is no guarantee on the quality of the target code (it is supposed only to be partially fit for the problem at hand), the similarity measure is only used to guide the evolutionary search. Therefore, a program is considered correct only when its input/output error rate is zero.

III. Program Similarity Detection Approaches

Measuring similarity between source code is a fundamental activity in software engineering. It has multiple applications, including identifying duplicate code in source code, code clone, plagiarism detection, code search, finding similar bug fixes [28] and code recommendation [29]. Dozens of similarity detection algorithms have been proposed last few decades, which can be classified into metrics, text, token, tree, and graph-based approaches based on the representation [30]. We selected four top-ranked similarity measures to evaluate their code synthesis capability when used within G3P.

A. Cosine

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) Preprocessing: 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 Calculation: For each token sequence of the source program, we compute the frequency of each token.

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

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

B. FuzzyWuzzy

FuzzyWuzzy [31] is a string matching open-source python library based on the difflib python library. It uses the Levenshtein Distance to calculate the differences between sequences. The library contains different similarity functions, including $TokenSortRatio$ and $TokenSetRatio$. Ragkhitwetsagul et al. [30] 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. In comparison, $TokenSetRatio$ takes out the common tokens instead of sorting them.

C. CCFinder

CCFinder [32] is a token-based clone detecting 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 simple, 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:

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

  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.
- 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.

D. SIM

SIM [33] 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 with special tokens. 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.

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., [27]), or (ii) Code Retrieval which seeks to find code snippets that highly match the textual description of the problem from a large source code repositories (e.g., Stack Overflow or Github). 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. Such measure can then be considered as a fitness function (i.e., an objective). Therefore, assuming that the target code is of decent quality, the similar the evolved program is to the target code, the better the evolved code.

This paper starts by devising a G3P algorithm that solely uses code similarity measures to assess its effectiveness to evolve programs. Then, we propose a novel multi-objective G3P that combines both input/output error rate and similarity measure as objectives to assess the fitness of each evolved program.

Figure 1 shows an overview of our proposed approach. It is a multi-objective (in the present paper, we limit the study to a bi-objective instance) extension of the G3P system with a code similarity measure as a second objective. 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 similarity of each evolved program.

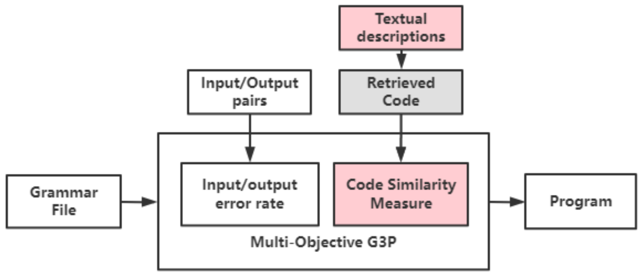

Fig. 1. Overview of our Multi-Objective G3P system

A. Similarity-Based G3P System

We start by devising a single objective similarity-based G3P system. The new G3P system uses code similarity to evaluate the fitness of evolved programs against a target source code, in replacement of the traditional input/output error rate. The rationale for starting with the design of a single objective G3P instead of a multiple objective version is to get some insight into the effectiveness of code similarity within G3P. Therefore, our focus with the single objective similarity-based

G3P is on (i) assessing the capability of G3P to evolve programs and converge towards a target code given in input using similarity measures and (ii) identifying the most suitable measure. Therefore, we create four different variations of the similarity-based G3P:

- G3P$_{Cosine}$: which uses Cosine to assess the code similarity
- G3P$_{Fuzzy}$: which uses FuzzyWuzzy to assess the code similarity
- G3P$_{CCFinder}$: which uses CCFinder to assess the code similarity
- G3P$_{SIM}$: which uses SIM to assess the code similarity

B. Multi-Objective G3P System

Building on the devised single objective similarity-based G3P system, we propose a multiple objective G3P system that uses both the (i) standard input/output error rate (from [9]) and (ii) the code similarity measure as objectives to evaluate the fitness of each evolved program and guide the evolutionary search process.

Since we have defined four different similarity measures, we devise four different multi-objective G3P algorithms:

- MOG3P$_{Cosine}$: which uses Cosine and input/output error rate
- MOG3P$_{Fuzzy}$: which uses FuzzyWuzzy and input/output error rate
- MOG3P$_{CCFinder}$: which uses CCFinder and input/output error rate
- MOG3P$_{SIM}$: which uses SIM and input/output error rate

It is worth noting that while both objectives are used by G3P to assess the fitness of each evolved program and serve as guides for the evolutionary process, we only consider that a program is correct when its input/output error rate is zero. The rationale behind this is that we often have no guarantee of the quality of the target code used for the similarity measure. We can only hope to have a target code that is close to the correct program or share some snippets (i.e., if we had the correct code, then the problem is solved without any evolutionary search).

The multi-objective G3P system starts by creating the initial population based on the selected context-free grammar for the data type used in the corresponding problem as in [9]. Then the system iterates through multi-objective tournament selection, crossover, mutation, fitness value evaluation, and update of the population until satisfying the termination condition (in our case, the number of generation).

Our multi-objective G3P algorithm differs from the original G3P on the following three aspects:

- The additional similarity-based objective calculation in fitness evaluation.
- Adaptation of the parents selection for crossover and mutation operators.
- Modification of the population update strategy.

These three differences are described in detail in the following subsections.

1) Multi-objective Fitness Evaluation: In addition to the traditional objective that calculates the fitness value based on input/output pairs, the second fitness value is evaluated in each generation using code similarity detection algorithms presented in section III. The option for selecting a different similarity detection algorithm as a parameter for the G3P is created, which allows the multi-objective G3P system to evaluate the similarity-based fitness score using different algorithms.

2) Selector: The previous G3P system used tournament selector and lexicase selector as the parent selection operator for crossover and mutation. In this study, the tournament selector adapted to multi-objective purposes as it can form the parent based on both error-based fitness values and similarity-based fitness values. Three variants of tournament selectors are created in this study. The first variant forms the parents using one parent selected by objective one and the other from objective two. Each parent selection operates just like a classical tournament selector but with different objectives. The second one selects the first half of the parents from objective one, and the second-half form the other objective. The third approach is a mixer of the previous two approaches, which selects both parents from objective one for one third, selects both parents from objective two for second one third, and the rest is formed by one parent from objective one and the other from another.

We ran a couple of experiments with the system using different parent formation approaches, and the result shows that the third approach, which forms one-third of the parents using the first objective, one third from the second objective, and the rest using half from the first objective and another half from the second objective, achieved the best performance. The result of the multi-objective G3P system uses the third approach as the multi-objective tournament selector.

3) Update population using multiple objectives: The next generation individual selection for updating the population is another critical operation of the GP system. The decision for which evolved individual goes to next generation is solely chosen by single objective in the previous study. Like the idea of parent selection, both objectives are used for next-generation selection. However, this time it chooses half of the population based on the first objective, and the other objective decides the other half.

V. Experiment Setup

A. Problem Description

Helmuth and Spector [11], [12] introduced a set of program synthesis problems. These problems were based on coding problems that might be found in introductory 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 I describes the characteristics of each of the program synthesis problems considered in our evaluation.

TABLE I
DESCRIPTION AND CHARACTERISTICS 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

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 the 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, 2, 3, 4, and 5 depict the target programs for the oracle assessment of program similarity for Number IO, Smallest, Median, String Lengths Backwards, and Negative To Zero respectively.

```
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(str1, str2, str3):
    result = False
    if(len(str1) < len(str2)):
        if(len(str2) < len(str3)):
            result = True
    return result
```

Listing 4. Target program for String Lengths Backwards

```
def negativeToZero(list):
    result=[]
    for number in 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 parameter setting for the experiment is as close to error-based G3P to compare and analyze how well the similarity-based G3P does. The G3P system in [9] was tested with two selection operators with hundred runs for all 29 problems in the benchmark suite. It successfully solved ten problems using tournament selection and sixteen problems with lexicase selection. Moreover, the success rate for half of the solved problems was under 10% (i.e., 10% of the runs generate a correct program for each of the 10 "solved" problems). For this study, we tested our similarity-based G3P system on five selected problems (Median, Negative to Zero, Number IO, Smallest, and String Lengths Backwards) with a relatively high success rate (more than 10%) to check if the system capable of achieving higher success rate. Besides, we evaluated the results using 30 runs for each problem since we got different results running G3P as reported in [9]. For most of the problems in the suite, it evolves 300 generations, while for easier problems (Median, Number IO, and Smallest), 200 generations are suggested in [12]. The general settings for the GP system are shown in Table II.

TABLE II
EXPERIMENT PARAMETER SETTINGS

Parameter	Setting
Runs	30
Generation	300[a]
Population size	1000
Tournament size	7
Crossover probability	0.9
Mutation probability	0.05
Node limit	250
Variable per type	3
Max execution time	1 second

[a]200 generations for Median, Number IO, and Smallest as in [11]

VI. Results

In this section, we report and discuss the results of our evaluations. In the first subsection, we start by comparing the performance of G3P using each of the similarity measures, and then we compare them against the traditional error-rate-based G3P. In the second subsection, we report on how well the multi-objective G3P performs compared to the classical G3P with input/output error rate as an evolution objective. The performance of each algorithm is in terms of number of times (or independent runs) in which the algorithm successfully evolves at least one correct program (i.e., a program that finds the correct output for all the inputs within the testing input/output pairs).

A. Similarity-Based G3P

Results of the similarity-based G3P systems are reported in this subsection and compared to the error-based G3P. The goal of this experiment is to assess whether G3P is able to evolve a program that solves a program synthesis problem (only known to an oracle) solely based on the used similarity measure.

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 is shown in the Table III. We see from Table III that similarity-based G3P was able to evolve the correct programs for Number IO, Negative To Zero, and Smallest at least once with Cosine, Fuzzy, CCFinder, and SIM. However, similarity-based G3P did not manage to evolve any correct program for Median and String Lengths Backwards. Similarity-based 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 algorithm 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.

We see that G3P with input/output error rate is capable of evolving correct problems to all the considered program synthesis problems. Furthermore, it can also find 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 standard 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).

Overall, we could say that G3P has the potential to evolve programs for synthesis problems using similarity measures. However, no similarity measure seems to work better than the rest and the similarity-based G3P system is not able to achieve a similar result as the error-based G3P system. We are not surprised by the result since programming is a very precise task that the difference of a very tiny change will lead the program to a completely different result. The similarity-based G3P system evolves the solution to make it look similar to the correct program of the solution. However, It can not distinguish between a good and a lousy evolution compared to the error-based fitness function. Nevertheless, the experiment showed that it is tough to converge to the correct program even if we give the correct solution as the system's input.

TABLE III
Number of times out of 30 runs a correct program is found

Problem	G3P Original	G3P Cosine	G3P Fuzzy	G3P CCFinder	G3P CCFinder
Smallest	29	0	0	5	17
Number IO	29	27	0	3	0
Median	4	0	0	0	0
String Lengths Backwards	2	0	0	0	0
Negative to Zero	7	0	1	0	0

B. Multi-Objective G3P System

The results of multi-objective G3P compared to the G3P system in [9] are shown in Table IV. For more straightforward problems, Number IO and Smallest, all four similarity-based G3P systems using different similarity detection algorithms get similar results as the previous G3P system. The multi-objective G3P system using CCFinder and SIM algorithm successfully finds the correct solution for all thirty runs, which offers a more consistent success rate than input/output error-based G3P. The multi-objective G3P system using cosine similarity measure more than doubles the runs that solve all test cases on the relatively more challenging problem (i.e., Median). The other three experiments got a similar result as the error-based G3P system. In the problem Negative To Zero, the system using the FuzzyWuzzy string matching algorithm got six more successful runs compared to the error-based G3P system, while the rest of the G3P systems with other similarity detection algorithms achieved a similar result. In the problem String Lengths Backwards, except that the system using CCfinder was unable to find the correct solution after 30 runs, other methods get similar results.

In general, multi-objective G3P with cosine similarity measure significantly improved on the Median dataset, and the multi-objective G3P system using the FuzzyWuzzy string matching algorithm almost doubled the performance on the Negative To Zero dataset. However, improvements of using a multi-objective G3P system in other problems using other algorithms are minimal.

Overall, Our proposed multi-objective G3P successfully manages to improve the success rate over the state-of-the-art G3P on all the considered programs. However, no similarity measure improves G3P's success rate on all problems.

TABLE IV
NUMBER OF TIMES OUT OF 30 RUNS A CORRECT PROGRAM IS FOUND (I.E., PROBLEM IS SOLVED).

Problem	G3P Original	MOG3P Cosine	MOG3P Fuzzy	MOG3P CCFinder	MOG3P CCFinder
Number IO	29	29	29	30	30
Smallest	29	28	28	30	30
Median	4	9	2	5	3
String Lengths Backwards	2	1	2	0	3
Negative to Zero	7	7	13	9	8

VII. Conclusion and Future Work

G3P is widely recognised as one of the most successful approaches for program synthesis. However, 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.

In this paper, we wanted 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 from textual problem descriptions). We devised and evaluated a novel multi-objective G3P system that uses both the traditional input/output error rate and the code similarity. Our evaluation with multiple code similarity detection measures on a well-known synthesis benchmark suite has shown that our proposed multi-objective G3P successfully manages to improve the success rate over the state-of-the-art G3P on all the considered programs. However, no similarity measure improves G3P's success rate on all problems.

To tackle the limitations our proposed approach, we aim, in a future work, to simultaneously use multiple similarity measures at once in the form of a many objective G3P. Furthermore, we plan to evaluate other code similarity detection algorithms (e.g., tree-based code clone detection) and adapt evolutionary operators to take full advantage of the knowledge obtained from the similarity algorithms.

Acknowledgement

This work was supported by Science Foundation Ireland grant 13/RC/2094_P2 to Lero - the Science Foundation Ireland Research Centre for Software (www.lero.ie).

References

[1] H. Dakhore and A. Mahajan, "Generation of c-code using xml parser," *ISCET*, vol. 2010, 2010.
[2] F. Boutekkouk, "Automatic systemc code generation from uml models at early stages of systems on chip design," *IJCA*, 2010.
[3] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. Mankowitz, E. Sutherland Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," Feb 2022.
[4] I. A. Niaz, J. Tanaka *et al.*, "Mapping uml statecharts to java code." in *IASTED Conf. on Software Engineering*, 2004.
[5] Y. Bassil and M. Alwani, "Autonomic html interface generator for web applications," *arXiv preprint arXiv:1202.2427*, 2012.
[6] T. Beltramelli, "pix2code: Generating code from a graphical user interface screenshot," in *ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, 2018.
[7] J. R. Koza *et al.*, *Genetic programming II*, 1994, vol. 17.
[8] E. Pantridge and L. Spector, "Pyshgp: Pushgp in python," in *GECCO*, 2017.
[9] 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.
[10] S. Forstenlechner, "Program synthesis with grammars and semantics in genetic programming," *Ph. D. dissertation*, 2019.
[11] T. Helmuth and L. Spector, "General program synthesis benchmark suite," in *GECCO*, 2015.
[12] T. Helmuth and L. Spector, "Detailed problem descriptions for general program synthesis benchmark suite," *School of Computer Science, University of Massachusetts Amherst, Tech. Rep.*, 2015.
[13] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *ICPC*, 2018.
[14] C. V. Alexandru, "Guided code synthesis using deep neural networks," in *FSE*, 2016.
[15] N. Tao, A. Ventresque, and T. Saber, "Assessing similarity-based grammar-guided genetic programming approaches for program synthesis," in *OLA*, 2022.
[16] M. Brameier, W. Banzhaf, and W. Banzhaf, *Linear genetic programming*, 2007.
[17] J. F. Miller and S. L. Harding, "Cartesian genetic programming," in *Proceedings of the 10th annual conference companion on Genetic and evolutionary computation*, 2008.
[18] M. O'Neill, M. Nicolau, and A. Agapitos, "Experiments in program synthesis with grammatical evolution: A focus on integer sorting," in *CEC*, 2014.
[19] T. Saber and S. Wang, "Evolving better rerouting surrogate travel costs with grammar-guided genetic programming," in *CEC*, 2020.
[20] 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.
[21] 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.
[22] 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*, pp. 1–39, 2019.
[23] 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.
[24] 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, pp. 118–134.
[25] P. A. Whigham, "Grammatical bias for evolutionary learning." 1997.
[26] M. O'Neill and C. Ryan, "Grammatical evolution: Evolutionary automatic programming in a arbitrary language, volume 4 of genetic programming," 2003.
[27] J. Jeon, X. Qiu, J. S. Foster, and A. Solar-Lezama, "Jsketch: sketching for java," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015.
[28] 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.
[29] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *ICSE*, 2005.
[30] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "A comparison of code similarity analysers," *ESE*, 2018.
[31] A. Cohen, "Fuzzywuzzy: Fuzzy string matching in python," 2011.
[32] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *TSE*, 2002.
[33] D. Gitchell and N. Tran, "Sim: a utility for detecting similarity in computer programs," *ACM Sigcse Bulletin*, 1999.