2023 IEEE Latin American Conference on Computational Intelligence (LA-CCI) | 979-8-3503-4807-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/LA-CCI58595.2023.10409384

# Program Synthesis with Generative Pre-trained Transformers and Grammar-Guided Genetic Programming Grammar

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

***Abstract*—Grammar-Guided Genetic Programming (G3P) is widely recognised as one of the most successful approaches to program synthesis. Using a set of input/output tests, G3P evolves programs that fit a defined BNF grammar and that are capable of solving a wide range of program synthesis problems. However, G3P's inability to scale to more complex problems has limited its applicability. Recently, Generative Pre-trained Transformers (GPTs) have shown promise in revolutionizing program synthesis by generating code based on natural language prompts. However, challenges such as ensuring correctness and safety still need to be addressed as some GPT-generated programs might not work while others might include security vulnerabilities or blacklisted library calls. In this work, we proposed to combine GPT (in our case ChatGPT) with a G3P system, forcing any synthesised program to fit the BNF grammar–thus offering an opportunity to evolve/fix incorrect programs and reducing security threats. In our work, we leverage GPT-generated programs in G3P's initial population. However, since GPT-generated programs have an arbitrary structure, the initial work that we undertake is to devise a technique that maps such programs to a predefined BNF grammar before seeding the code into G3P's initial population. By seeding the grammar-mapped code into the population of our G3P system, we were able to successfully improve some of the desired programs using a well-known program synthesis benchmark. However, in its default configuration, G3P is not successful in fixing some incorrect GPT-generated programs–even when they are close to a correct program. We analysed the performance of our approach in depth and discussed its limitations and possible future improvements.**

***Index Terms*—Program Synthesis, Grammar Guided Genetic Programming, Generative Pre-trained Transformers, Large Language Models, Grammar**

## I. Introduction

The advent of the new era of Artificial Intelligence (AI) has led to a significant increase in automation across various application domains. AI has revolutionized the way we interact with technology and has opened up new possibilities for automation in fields such as finance, transportation, and programming. Automation has become an essential tool for businesses to increase efficiency and productivity while reducing costs. A plethora of research on automation in computer programming has been carried out with the aim of making the job of the programmer easier by giving them a variety of tools and methods to produce programming code based on their high-level intent. This process is commonly referred to as program synthesis, which has the potential to significantly reduce the time and effort required for software development while also improving the quality of the code produced.

A multitude of algorithms has been proposed for automatic programming using various programming languages, ranging from object-oriented languages such as Python [1] and Java [2] to procedural languages such as C [3], [4], to scripting and markup languages such as HTML [5], [6]. Despite the target language's diversity, there is also a lot of variation in the techniques and the types of user intent: Beltramelli proposed pix2code [6], a system that produces web development (i.e., HTML/CSS) interface code based on user provided graphical user interface screenshot images using Convolutional Neural Network (CNN). A transition and state mapping algorithm for generating Java code from Unified Modelling Language (UML) diagrams and state patterns developed by Niaz et al. [2]. Bassil and Alwani [5] developed an approach that outputs an HTML code using a Context-Free Grammar (CFG) parser and a finite-state machine-based lexical analyser. A G3P system proposed in [7]–[9] generates a Python program based on textual task description and input/output train and test cases. Boutekkouk [4] proposed a system that generates C codes based on Visual Basic and Action Language to exploit UML diagrams. Using huge transformer language models and large-scale sampling, the AlphaCode developer team created a code generation method to address previously unsolved competitive programming issues[1]. Nevertheless, despite this diversity, Ge-

979-8-3503-4807-1/23/$31.00 ©2023 European Union

[1]https://www.deepmind.com/blog/competitive-programming-with-alphacode

netic Programming (GP [10]) continues to be the competitive method for addressing program synthesis problems [11].

Genetic Programming [10] is a technique of evolving programs, starting from a population of unfit programs, fit for a particular task by applying operations analogous to natural genetic processes. One of the most effective GP systems is PushGP [12]. It generates programs in the stack-based language Push, which is designed specifically for program synthesis tasks. Every data type in Push has its own stack, which facilitates the genetic programming process. While PushGP has been shown to be highly effective at generating code for a variety of problems, its dependence on a specialized language makes it difficult to apply in real-world scenarios where other programming languages are more commonly used.

However, GP has limitations when it comes to evolving programs that are syntactically correct and semantically meaningful. G3P [1] is a technique that addresses this limitation by introducing language grammar to GP, such that the syntactic structure of programs may be constrained. G3P is widely recognised as one of the most successful approaches for program synthesis. G3P has been shown capable of successfully evolving programs in arbitrary languages that solve several program synthesis problems. Despite its success, the restriction on the system to only evolve with the randomly generated population it derives limits its performance to larger and more complex problems.

Following on the Large Language Models (LLMs) and GPT trend, there is a sizeable and growing number of approaches for generating source code based on textual problem descriptions. Therefore, it is now, more than ever, time to introduce G3P to evolve programs based on GPT-generated code, combined with a random initial population. GPT techniques might output: (i) several incomplete snippets or not fully fit-for-purpose codes which often makes them impossible to exploit in their form, (ii) incorrect programs which need improvement, or (iii) programs that include security vulnerabilities or blacklisted library calls that posing a safety/security risk. Therefore, in this work, we propose an approach whereby such GPT-generated programs are mapped into a BNF grammar.

In this paper, we would like to assess the potential for G3P to evolve programs based on the generated code from arbitrary sources (i.e., ChatGPT) using the description of the task. We particularly propose an algorithm that maps a code from arbitrary sources to a program that fits BNF grammar. 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 the code generated from the task prompt. Although, in its default configuration, G3P is not able to solve all the problems with a code close to the correct solution seeded into the initial population.

The rest of the paper is structured as follows: Section II summarises the background and work related to our study. Section III presents our proposed novel G3P approach. Section IV details our experimental setup. Section V reports and discusses the results of our experiments. Finally, Section VI concludes this work and discusses our future studies.

## II. Background and Related Work

This section provides an overview of the research background and related works.

### A. Genetic Programming

GP is an evolutionary algorithm that generates programs based on certain fitness calculations to perform specific tasks. With the goal to produce better programs, GP iteratively evolves a population starting with randomly selected individuals (typically not very well suited for purpose props) using operators similar to natural genetic processes (e.g., crossover, mutation, and selection). A number of GP systems have been proposed over time, each with their own unique characteristics (e.g., GP [10], Linear GP [13], Cartesian GP [14]).

### B. Grammar-Guided Genetic Programming

While there are many other GP systems, G3P is one of the most effective GP systems. G3P is a variant of GP that uses grammar as the representation, with the most famous variants being Context-Free Grammar Genetic Programming (CFG-GP) by Whigham [15] and Grammatical Evolution [16]. The use of grammar as a guideline for creating syntactically sound programs throughout its evolution is what makes G3P unique and powerful. Due to their adaptability and ability to be designed outside of the GP system to describe the search space, grammars are widely used in program synthesis [17], managing traffic systems [18], and scheduling wireless communications [19]–[23].

To handle various synthesis problems, Forstenlechner et al. [1] suggested a G3P system with a composite and self-adaptive grammar, which overcame the drawback of grammar that needed to be customised or adjusted for each problem. There are several short grammars predefined, each for a data type that specifies the function/program to be evolved. As a result, G3P may reuse these grammars for arbitrary problems while minimizing the scope of evolution by excluding unnecessary data types. Predefined grammars are further improved in [24] to handle character data type and recursions, which was not supported in the previous approach.

### C. Large Language Models

LLMs are deep learning algorithms that can identify, condense, translate, forecast, and produce text and other material using knowledge gathered from enormous datasets. LLMs emerged around 2018 [25] and perform well at a wide variety of tasks. They are built up of a neural network with several parameters (usually billions of weights or more), trained via self-supervised learning on a significant amount of unlabeled text. LLMs are among the most successful applications of transformer models.

OpenAI's ChatGPT [26] is a Generative Pre-trained Transformer with a LLM that can produce text that resembles human speech and programming code (full or partial). ChatGPT can produce content in a range of styles and formats as it is trained

on a vast corpus of text. It has been used for a wide range of applications, including content creation, customer service, language translation, and automated program generation.

### D. Abstract Syntax Tree

An abstract syntax tree (AST [27]) is a tree-shaped representation of the structure of source code written in a programming language. It is an abstract data structure that may be used to recreate code that is functionally the same as the original in any language. ASTs are often used to represent code written in a specific programming language, such as Python or C++. ASTs are widely used data structures to represent the structure of program code. It frequently acts as a program's interim representation at various phases and has a significant influence on the final executable code's optimization and performance.

## III. Proposed Approach

The goal of this research is double: (i) generate programs that correctly solve program synthesis problems (ii) while fitting a pre-defined grammar to reduce their security threat and ensure their safety. Additionally, as an added bonus, fitting programs to a BNF grammar enables us to leverage the research on G3P to continue evolving and improving GPT-generated programs. We build a system that is able to generate arbitrary source code, map to another program that fits the BNF grammar, and leverage the resulting program in G3P. The first step of our system is to use a GPT tool (e.g., ChatGPT) to generate a program using a textual task description. In the next step, we use our proposed algorithm to map the GPT-generated code into another program that fits the BNF grammar while handling potential grammar conflicts. Last but not least, the system evolves the population with the seeded program to achieve a better solution. This work is particularly focused on (i) mapping as much information from the GPT-generated code to programs that suit BNF grammar and (ii) improving the performance of the program synthesis by seeding GPT-generated programs in G3P.

Figure 1 shows an overview of our proposed approach. It is an extension of the G3P system with seeding programs from the arbitrary source into the initial population. The system starts by generating a program based on a textual description of the task. Then it maps the GPT-generated code into a program that suits BNF grammar and combines it with a randomly generated initial population to make it ready for evolution. Finally, it evolves the population to generate solutions for the task.

### A. Generate Code Based On Textual Description

The LLMs (i.e., ChatGPT) show amazing potential for generating code based on the user description. For the first step of the research, we used ChatGPT to generate the code by providing the task textual descriptions.

### B. Map GPT-Generated Programs into Programs That Fit BNF Grammar

Transforming arbitrary source code into a program that fits a BNF grammar poses a significant challenge for this research.

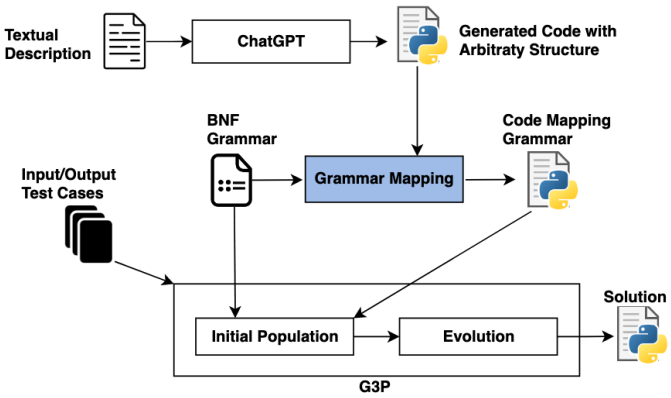


Fig. 1. Overview of our G3P system

The challenges of mapping such a program to a target BNF grammar are mainly (i) the unavailability of operators/function calls used in the source code in the target BNF grammar, (ii) the mismatch of variable names between the source code and the target BNF grammar, and (iii) unsupported structures in the target BNF grammar.

The mapping is applied in the following main steps:

- *Convert GPT-generated source code into AST*: We start mapping by transforming the source code into AST to split each expression. Then we can iterate through each expression and perform the remaining steps.
- *Map variable names*: Before mapping each expression, we applied variable name mapping based on variable type since a fixed number of variables are predefined for G3P evolution. Specifically, we check if the name of the variable in the GPT-generated code is the same as the predefined variable. Then we iterate through predefined variables in G3P that match the type of variable in GPT-generated code and change the name with the first unmapped variable name.
- *Handle each expression*: We build a recursion-based mapping algorithm for each type of expression. We used a top-down approach to build an expression tree that fits the BNF grammar (individuals are represented as a tree in G3P).
- *Insert a dummy expression in case of a conflict*: We replace mismatched variables with a random predefined variable that fits the variable data type. In addition, we insert a dummy expression (a simple assign statement that assigns a random predefined variable to itself) for unsupported expressions in BNF grammar.

In this way, we proposed a novel algorithm that is capable of mapping code from an arbitrary source into a syntactically correct program that suits BNF grammar while retaining as much information as possible.

### C. Evolve Population With Seeded Program

The mapped program from the previous section is seeded into G3P's initial population to guide its evolution more effectively. This approach is used to improve the quality of

the G3P algorithm by providing a starting point that is closer to the desired output.

## IV. Experiment Setup

In this section, we will provide a detailed overview of the experimental setup used in our research.

### A. Program Synthesis Problems

Helmuth and Spector [28], [29] introduced a set of program synthesis problems. It provides a textual description as well as two sets of input/output pairs for both training and testing during the program synthesis process. We used 28 problems (except "String Differences" as in [1], [24] since it requires a data structure as output value containing multiple types) to test the code generation capability of ChatGPT and proposed G3P system.

### B. Generating Program Using ChatGPT

The G3P calculates the fitness value of each individual solution by turning each G3P program into a Python function, running it with all inputs of all test cases, and comparing the returned values with the corresponding output values. However, using the original problem descriptions, verbatim, as prompts to ChatGPT will return code snippets (instead of functions) with console prints (instead of function returns). Thus, seeding such code into G3P's initial population will not be meaningful. To overcome this issue, we slightly modified the original textual description of the benchmark suite to explicitly request that ChatGPT generates a program formatted as a function and that the function returns the result instead of printing it on the console (as it is defined in the benchmark [28], [29]).

### C. Parameter Settings

The general settings for the GP system are: 100 Runs; 300 Generations; Population size of 1000; Tournament selection with tournament size of 7 (we forced at least one of each pair of first 500 parents contain seeded code to guarantee the seeded program is fully utilised); 0.9 Crossover probability and 0.05 mutation probability; 3 variable for per data type; and 1 second max execution time.

## V. Result

In this section, we present and discuss the results of our evaluations.

### A. GPT-generated Code analysis

We compare the results obtained by ChatGPT in our experiments with the results obtained by G3P with tournament selection in this subsection. Table I shows the results obtained by ChatGPT compared to the results obtained by G3P on the benchmark problems. A checkmark (✓) indicates that GPT-generated programs were correct or at least one correct solution has been found with 100 runs. A cross (✗) indicates that the GPT-generated programs were incorrect or the GP evolution did not find a correct solution after 100 runs.

Overall, ChatGPT has demonstrated remarkable power in generating code for benchmark problems, correctly suggesting 26 out of 28 considered problems. Even for the two failed problems, it was able to generate programs that were very close to the correct solution. For "Wallis Pi", it generated a program that calculates the value of $\pi$ instead of $\pi/4$, whereas for "Digits", it successfully separated each digit into a list but was unable to handle the negative sign (i.e.,"-").

TABLE I
ChatGPT Performance On Benchmark Suite Compared with G3P and Grammar Mapping Status For GPT-generated code

| Benchmark Problem | G3P | ChatGPT | Grammar Mapping Status |
|---|---|---|---|
| NumberIO | ✓ | ✓ | ✓ |
| Small Or Large | ✓ | ✓ | ✓ |
| For Loop Index | ✗ | ✓ | ✓ |
| Compare String Lengths | ✗ | ✓ | ✓ |
| Double Letters | ✗ | ✓ | ✓ |
| Collatz Numbers | ✗ | ✓ | ✓ |
| Replace Space with Newline | ✓ | ✓ | ✓* |
| Even Squares | ✗ | ✓ | ✓* |
| Wallis Pi | ✗ | ✗ | ✓ |
| String Lengths Backwards | ✓ | ✓ | ✗ |
| Last Index of Zero | ✓ | ✓ | ✓* |
| Vector Average | ✗ | ✓ | ✓* |
| Count Odds | ✓ | ✓ | ✓ |
| Mirror Image | ✓ | ✓ | ✓ |
| Super Anagrams | ✗ | ✓ | ✗ |
| Sum of Squares | ✗ | ✓ | ✗ |
| Vectors Summed | ✗ | ✓ | ✗ |
| X-Word Lines | ✗ | ✓ | ✗ |
| Pig Latin | ✗ | ✓ | ✗ |
| Negative To Zero | ✓ | ✓ | ✗ |
| Scrabble Score | ✗ | ✓ | ✗ |
| Word Stats | ✗ | ✓ | ✗ |
| Checksum | ✗ | ✓ | ✗ |
| Digits | ✗ | ✗ | ✓ |
| Grade | ✓ | ✓ | ✓ |
| Median | ✓ | ✓ | ✓ |
| Smallest | ✓ | ✓ | ✓ |
| Syllables | ✓ | ✓ | ✓ |
| Number Of Problems Solved | 12 | 26 | 13 |

### B. Map GPT-generated Code into Solutions Fit BNF Grammar

Grammar Mapping Status column in Table I shows the mapping status from the GPT-generated code into solution that fit the BNF grammar. We categorised the conversion status into three groups: "Completely Mapped" (noted as ✓), which is able to map all of the GPT-generated codes into a code that fits the BNF grammar; "Mostly mapped" (noted as ✓*), which indicates that most of the structure has been successfully mapped; and "Poorly mapped" (noted as ✗), which indicates that no informative part has been mapped successfully. Overall, our proposed algorithm successfully mapped GPT-generated codes for 13 problems into programs that fit BNF grammar with no loss of information (i.e., Completely Mapped). It also mapped GPT-generated codes for four problems to programs that are close to being solved (i.e., Mostly mapped). However, it was unable to map GPT-generated solutions for 11 problems due to large grammar

conflicts (i.e., Poorly mapped). The analysis of the mapped code is discussed in the following subsections.

*1) Analysis of Mostly Mapped Programs:* In this subsection, the GPT-generated codes and our grammar-mapped programs are reported for the four mostly mapped programs from table I (i.e., Replace Space with Newline, Even Squares, Last Index of Zero, and Digits). We also analysed the part that we were not able to map correctly in each case.

- Replace Space with Newline: For this problem, everything except the calculation of the new line character using List Comprehension (line 2 in GPT-generated code in Listing 1. and grammar-mapped code in Listing 2.) is successfully mapped.

```
1 res0 = in0.replace(" ", "\n")
2 count = len([c for c in in0 if c != " "])
3 res1 = count
```

Listing 1. GPT-Generated Program for "Replace Space with Newline"

```
1 res0 = in0.replace(' ','\n')
2 i2 = len(s0)
3 res1 = i2
```

Listing 2. Grammar-Mapped Program for "Replace Space with Newline"

- Even Squares: The grammar-mapped program ignored the *step parameter* since BNF grammar does not support for *Range function* (line 1 in Listing 3 and Listing 4). It also replaces *is_integer function* with predefined boolean variable since G3P does not support specific function (line 3 in Listing 3 and Listing 4).

```
1 for i in range(2, in0, 2):
2    if (i**0.5).is_integer():
3        res0.append(i)
```

Listing 3. GPT-Generated Program for "Even Squares"

```
1 for i2 in saveRange(int(2.0), in0):
2    if b0:
3      res0.append(i2)
```

Listing 4. Grammar-Mapped Program for "Even Squares"

- Last Index of Zero: The Listing 5 and Listing 6 shows the GPT-generated program and grammar-mapped program of this problem. It ignored *step parameter* as mentioned in the previous problem and also ignored *break statement*, which is not supported in BNF grammar.

```
1 res0 = -1
2 for i in range(len(in0)-1, -1, -1):
3    if in0[i] == 0:
4        res0 = i
5        break
```

Listing 5. GPT-Generated Program for "Last Index of Zero"

```
1 res0 = -int(1.0)
2 for i2 in saveRange(( len(in0) - int(1.0) ), -
    int(1.0)):
3    if (getIndexIntList(in0, i2) == int(0.0)):
4      res0 = i2
```

Listing 6. Grammar-Mapped for "Last Index of Zero"

- Digits: The *return statement* (line 2 in Listing 7 and Listing 8) is ignored as it is not supported. The negative sign in line 5 in Listing 7 and Listing 8) replaces with "0.0" due to only the number can be appended to the list in BNF grammar.

```
1 if in0 == 0:
2    return [0]
3 res0 = []
4 if in0 < 0:
5    res0.append("-")
6    in0 = -in0
7 while in0 > 0:
8    res0.append(in0 % 10)
9    in0 //= 10
```

Listing 7. GPT-Generated Program for "Digits"

```
1  if (in0 == int(0.0)):
2    b0 = b0
3  res0 = li0
4  if (in0 < int(0.0)):
5    res0.append(int(0.0))
6  while (in0 > int(0.0)):
7    res0.append(mod(in0,int(10.0)))
8    in0 += int(10.0)
```

Listing 8. Grammar-Mapped Program for "Digits"

*2) Poorly Mapped Programs Analysis:* Eleven poorly mapped problems are categorised and analysed in this subsection.

- *"String Lengths Backwards", "Sum of Squares", "Vectors Summed", and "Negative To Zero"*: Advanced Python grammar (i.e., List Comprehension) used for GPT-generated code that BNF grammar is unable to map.
- *"Super Anagrams"*: The BNF grammar for this problem does not support *compound statements* (i.e., *if statements*), which failed to map the structure from the GPT-generated code.
- *"X-Word Lines", "Pig Latin", "Word Stats"*: These problem involves splitting a string, but BNF grammar supports the *Split function* as a part of for loop only. It also used the *join function* for string, which is not supported.
- *"Super Anagrams", "Scrabble Score"*: The GPT-generated code for these problems contains *Dictionary* data type, which is not supported by BNF grammar.

### C. Evolve G3P With Seeded GPT Program

The results of our G3P with a seeded GPT-generated program (that was mapped to the grammar) compared to G3P are shown in Table II. Overall, our proposed G3P system (seeded G3P) successfully solved 16 problems while G3P solved only 12 problems. For the problems "Replace Space with Newline" and "Last Index of Zero", G3P with a ChatGPT program as seed significantly improved the success rate. However, for the remaining two problems in the "Mostly Mapped" category (i.e., "Even Square" and "Digits"), G3P was unable to fix the incomplete/incorrect GPT-generated programs through evolution even though these GPT-generated programs were very close to the correct solution.

## VI. Conclusion and Future Work

One of the most effective methods for program synthesis is widely recognised to be G3P. However, G3P's inability to scale

TABLE II
NUMBER OF TIMES OUT OF 100 RUNS A CORRECT PROGRAM IS FOUND

| Benchmark Problem | G3P | Seeded G3P |
|---|---|---|
| NumberIO | 39 | 100 |
| Small Or Large | 5 | 100 |
| For Loop Index | 0 | 100 |
| Compare String Lengths | 0 | 100 |
| Double Letters | 0 | 100 |
| Collatz Numbers | 0 | 100 |
| Replace Space with Newline | 4 | 43 |
| Even Squares | 0 | 0 |
| Wallis Pi | 0 | 0 |
| String Lengths Backwards | 3 | 4 |
| Last Index of Zero | 28 | 100 |
| Vector Average | 0 | 0 |
| Count Odds | 1 | 100 |
| Mirror Image | 33 | 100 |
| Super Anagrams | 0 | 0 |
| Sum of Squares | 0 | 0 |
| Vectors Summed | 0 | 0 |
| X-Word Lines | 0 | 0 |
| Pig Latin | 0 | 0 |
| Negative To Zero | 7 | 6 |
| Scrabble Score | 0 | 0 |
| Word Stats | 0 | 0 |
| Checksum | 0 | 0 |
| Digits | 0 | 0 |
| Grade | 6 | 100 |
| Median | 45 | 100 |
| Smallest | 93 | 100 |
| Syllables | 5 | 100 |
| Number Of Problems Solved | 12 | 16 |

to larger and more complex program synthesis problems has limited its applicability. Recently, GPTs have shown promise in revolutionizing program synthesis by generating code based on natural language prompts. However, challenges such as ensuring correctness and safety still need to be addressed as some GPT-generated programs might not work while others might include security vulnerabilities or blacklisted library calls. In this work, we proposed to combine GPT (in our case ChatGPT) with a G3P system, forcing any synthesised program to fit the BNF grammar–thus offering an opportunity to evolve/fix incorrect programs and reducing security threats of GPT-generated programs. In our work, we leverage ChatGPT's generated programs in G3P's initial population. We presented a novel algorithm to map an arbitrary source program to a program that applies BNF grammar and successfully mapped the solution generated by ChatGPT to the program that suits BNF grammar. We evaluated our system for problems from the well-known benchmark suite. Our results show that our approach can effectively map Python programs to BNF grammar and improve the performance of G3P for some problems. Our approach has the potential to be used in a wide range of applications, including software development, code optimization, and code generation, and could be extended to other programming languages and grammar.

To tackle the limitations of our proposed approach, we aim, in future work, to take full advantage of the knowledge obtained from the seeded solution by adapting evolutionary operators (the crossover and selection operators) of the G3P system.

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

## REFERENCES

[1] 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.
[2] I. A. Niaz, J. Tanaka *et al.*, "Mapping uml statecharts to java code." in *IASTED*, 2004.
[3] H. Dakhore and A. Mahajan, "Generation of c-code using xml parser," *ISCET*, vol. 2010, 2010.
[4] F. Boutekkouk, "Automatic systemc code generation from uml models at early stages of systems on chip design," *IJCA*, 2010.
[5] Y. Bassil and M. Alwani, "Autonomic html interface generator for web applications," *arXiv*, 2012.
[6] T. Beltramelli, "pix2code: Generating code from a graphical user interface screenshot," in *ACM SIGCHI*, 2018.
[7] N. Tao, A. Ventresque, and T. Saber, "Assessing similarity-based grammar-guided genetic programming approaches for program synthesis," in *OLA*. Springer, 2022.
[8] N. Tao, A. Ventresque, and T. Saber, "Multi-objective grammar-guided genetic programming with code similarity measurement for program synthesis," in *IEEE CEC*, 2022.
[9] N. Tao, A. Ventresque, and T. Saber, "Many-objective grammar-guided genetic programming with code similarity measurement for program synthesis," in *IEEE LACCI*, 2023.
[10] J. R. Koza *et al.*, *Genetic programming II*. MIT press, 1994.
[11] 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.
[12] E. Pantridge and L. Spector, "Pyshgp: Pushgp in python," in *GECCO*, 2017.
[13] M. Brameier, W. Banzhaf, and W. Banzhaf, *Linear genetic programming*. Springer, 2007.
[14] J. F. Miller and S. L. Harding, "Cartesian genetic programming," in *GECCO*, 2008.
[15] P. A. Whigham, "Grammatical bias for evolutionary learning." 1997.
[16] M. O'Neill and C. Ryan, "Grammatical evolution: Evolutionary automatic programming in a arbitrary language, volume 4 of genetic programming," 2003.
[17] M. O'Neill, M. Nicolau, and A. Agapitos, "Experiments in program synthesis with grammatical evolution: A focus on integer sorting," in *IEEE CEC*, 2014.
[18] T. Saber and S. Wang, "Evolving better rerouting surrogate travel costs with grammar-guided genetic programming," in *IEEE CEC*, 2020.
[19] 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.
[20] 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.
[21] 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.
[22] 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 *IEEE CEC*, 2020.
[23] 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.
[24] S. Forstenlechner, D. Fagan, M. Nicolau, and M. O'Neill, "Extending program synthesis grammars for grammar-guided genetic programming," in *PPSN*. Springer, 2018.
[25] C. D. Manning, "Human language understanding & reasoning," *Daedalus*, 2022.
[26] OpenAI, "Gpt-4 technical report," 2023.
[27] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *ICSME*. IEEE, 1998.
[28] T. Helmuth and L. Spector, "General program synthesis benchmark suite," in *GECCO*, 2015.
[29] T. Helmuth and L. Spector, "Detailed problem descriptions for general program synthesis benchmark suite," in *University of Massachusetts Amherst*, 2015.