Program Synthesis on Single-Layer Loop Behavior in Pure Functional Programming

Tzu-Hao Hsu
Taiwan Evolutionary Intelligence Lab
Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan
r11921044@ntu.edu.tw

Chi-Hsien Chang
Taiwan Evolutionary Intelligence Lab
Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan
d07921004@ntu.edu.tw

Tian-Li Yu
Taiwan Evolutionary Intelligence Lab
Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan
tianliyu@ntu.edu.tw

***Abstract*—Program synthesis (PS) is a field devoted to automatically generating computer programs from high-level specifications, and genetic programming (GP) is one commonly-used way to achieve PS. PushGP, operating on a stack-based language, is considered as a state-of-the-art program synthesizer among GPs, while another research trend foucus on the grammar-based languages due to the readability and the ease of maintenance. In this paper, we propose the repetitive structure genetic programming (RSGP), a new grammar-based program synthesizer under the pure functional programming paradigm. RSGP defines a recursive function to simulate the single-layer loop behavior and leverages the minimum redundancy maximum relevance (mRMR) feature selection with the Pearson correlation coefficient (PCC) to select the capable and diverse programs for the next generation. The experiment results show that RSGP outperforms PushGP, CBGP, and HOTGP in terms of the number of successful programs on CountOdds and LastIndexofZero from PSB1, Luhn from PSB2, and 3 out of 4 designed problems. Additionally, the ablation study indicates that using mRMR with PCC does encourage proper problem decomposition with the trade-off of diminishing the search ability within a similar neighborhood. RSGP utilizes an adaptation mechanism to balance the trade-off to automatically fit the needs of different problems.**

***Index Terms*—Program Synthesis, Genetic Programming**

I. Introduction

Program synthesis (PS) is an area focusing on automatically generating computer programs from high-level specifications [1]. Inductive program synthesis, also known as programming by examples (PBE) [2], is a commonly-used approach to PS [3], [4]. Synthesizer alleviates the workload of programmers [1] and enables users without programming language knowledge to create computer programs [3]. Users provide the synthesizer with inputs along with their corresponding outputs, and the synthesizer generates programs based on those examples [5]. The advantage of PBE lies in its simplicity for representing program behaviors [5], [6].

The vast search space makes PS challenging to find solutions through brute-force methods, and hence, PS usually employs heuristic approaches like genetic programming (GP) [7]. Two approaches in PS using GP are stack-based GP and grammar-based GP. In stack-based GP, PushGP [8] was introduced in 2001, while Plush, introduced in 2018 [9], represents programs in PushGP using the linear genome. This approach addresses issues such as the lack of uniformity caused by tree-based GP in mutation. Subsequently, UMAD [10] improves mutation strategies under linear genome representation. Later, CBGP [11] based on PushGP was introduced to enhance the readability. As for grammar-based GP, two of most representative algorithms are the grammatical evolution [12] and G3P [13]. Both approaches utilize rules from the grammar of a programming language to generate and evolve programs. However, the drawback of grammar-based GP is that the search space expands rapidly as the grammar size increases [4]. The following works are addressing this issue. The strong typed genetic programming (STGP) [14] further enforces that arguments of the functions are of the correct data type to avoid unnecessary program generation. One of the research [15] utilizes textual descriptions of problems to reduce the search space, and determines which grammar to use based on keywords mentioned in the description. Another research [16] shows the performance of PS under the pure functional programming paradigm is consistently better than under the imperative programming paradigm in most selected problems. HOTGP [7] also performs pure functional program synthesis and achieves competitive performance on benchmarks compared to other state-of-the-art (SOTA) GP algorithms. This paper compares our proposed method with three representative algorithms: PushGP [8], CBGP [11], and HOTGP [7]. The following provides a brief introduction to each of them.

PushGP [8] is one of the SOTA program synthesizers [7], [17], [18] in GP that operates a stack-based language called Push [19]. In PushGP, operands are pushed into Push and popped out to perform the corresponding operator. Stack-based programming languages significantly differ from commonly-used imperative programming languages; therefore, programmers often face challenges in reading, comprehending, and maintaining them [4].

To address this drawback, CBGP [11] integrates the stack-based programming language with functional programming. While based on PushGP, this approach compiles the program using type-safe abstract syntax trees (AST). This abstraction enables program generation in languages such as Python and addresses the concerns of readability and usability associated

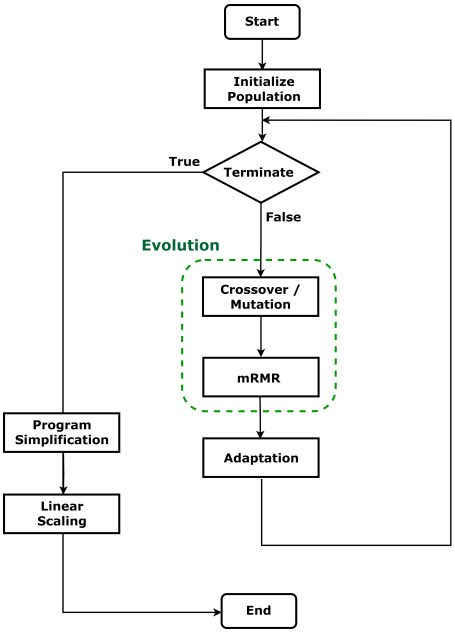

Fig. 1. The flowchart of the RSGP algorithm.

with Push [4]. The results show that CBGP achieves a higher success rate in finding solutions for a subset of test problems compared to other GP methods. However, CBGP fails to solve problems that require non-trivial control flow [20].

To further enhance the grammar-based GP performance on benchmark problems, HOTGP [7] operates Haskell programs that support higher-order functions under the pure functional programming paradigm since pure functional programming avoids side effects due to its fundamental property of referential transparency, ensuring that functions produce the same output for the same input [21].

In this work, we propose the repetitive structure genetic programming (RSGP), a grammar-based genetic programming approach under the pure functional programming paradigm. To simulate the single-layer loop behavior, we define a recursive function that is similar to the fold (reduction) function [22], [23] since the concept of loops, which serves as a basic primitive in most imperative languages, does not exist in pure functional programming. Furthermore, we leverage the minimum redundancy maximum relevance (mRMR) feature selection [24] with the Pearson correlation coefficient (PCC) to select the capable and diverse programs for the next generation. We aim to achieve the SOTA level by addressing the aforementioned issues. Our approach is subject to the following limitations. The program synthesized by RSGP inputs an integer list and outputs a single integer. Additionally, our approach does not yet consider nested loops.

The rest of the paper is organized as follows. Section II presents the details of our proposed method. Section III presents the experiment setting and results. Section IV concludes the paper.

TABLE I
SUMMARY OF SYMBOLS IN THIS PAPER

Symbol	Definition	Value
α	Crossover rate of RSGP.	0.1
w	mRMR redundancy weight.	0.9
r	The Pearson correlation coefficient.	-
e	The fitness of a program.	-
s, s'	The best fitness before and after evolving, respectively.	-
X, y	The example inputs and outputs.	-
P, O	The symbols denote population and offspring, respectively.	-
P_{low}, P_{base}, P_{high}	Three subpopulations in the adaptation.	-

Algorithm 1: Evolution

Input: P: population; o: the number of offspring to generate; X: inputs; y: outputs; w: mRMR redundancy weight;
Output: P: the new population

$O \leftarrow$ empty set
for $i = 1$ *to* o **do**
  $rand \leftarrow$ sample from uniform [0, 1)
  $p_1 \leftarrow$ Exponential ranking selection from P
  if $rand < \alpha$ **then**
    $p_2 \leftarrow$ Exponential ranking selection from P
    $O_i \leftarrow$ SubtreeCrossover(p_1, p_2)
  else
    $O_i \leftarrow$ SubtreeMutation(p_1)
$P \leftarrow$ mRMRSelection(P, O, X, y, w) (Algorithm 2)
return P

II. METHODOLOGY

This section begins with an overview of RSGP procedures and a grammar table used for program initialization. We then detail the recursive function L for simulating single-layer loops. Next, the mRMR selection and the adaptation mechanism are described. Finally, we explain the simplification and linear scaling methods applied to the best program. We summarize the symbols mentioned in this paper in Table I.

A. RSGP Algorithm

Fig. 1 illustrates the flow of the algorithm. Initially, each program in the population is initialized using the *grow* method, which creates program trees within a maximum depth limit. The algorithm then progresses to the evolution stage. At the end of the evolution, mRMR is used to select the population for the next generation, and an adaptation mechanism adjusts the parameter of mRMR. Upon algorithm termination, the best program is simplified to enhance generalization, and linear scaling is applied to address the issue of different scales between the output of the best program and the ground truth. The termination condition is either when RSGP finds the solution or when the evaluation reaches the maximum limit. The program with the best fitness on the train set is called

TABLE II
CONTEXT-FREE GRAMMAR TABLE OF RSGP

1	< program >	::=	< int >
2	< int >	::=	(L < list > < f_func > < g_func >) | (< int_op > < int > < int >) | (if < bool > < int > < int >)
3			| (< list_op > < list >) |< const >
4	< list >	::=	(cdr < list >) | (mapcar < λ > < list >) | (filter < λ_bool > < list >) | (cons < int > < list >) | input
5	< λ >	::=	(if < λ_bool > < λ > < λ >) | (< int_op > < λ > < λ >) |< const >| u
6	< f_func >	::=	(if < f_bool > < f_func > < f_func >) | (< int_op > < f_func > < f_func >) |< const >| u | v | idx
7	< g_func >	::=	(if < g_bool > < g_func > < g_func >) | (< int_op > < g_func > < g_func >) |< const >| u | idx
8	< bool >	::=	(< logic_op > < bool > < bool >) | (not < bool >) | (< bool_op > < int > < int >)
9	< λ_bool >	::=	(< logic_op > < λ_bool > < λ_bool >) | (not < λ_bool >) | (< bool_op > u < λ >)
10	< f_bool >	::=	(< logic_op > < f_bool > < f_bool >) | (not < f_bool >) | (< bool_op > u < f_func >)
11	< g_bool >	::=	(< logic_op > < g_bool > < g_bool >) | (not < g_bool >) | (< bool_op > u < g_func >)
12	< int_op >	::=	+ | - | * | floor | mod
13	< list_op >	::=	car | last
14	< logic_op >	::=	and | or
15	< bool_op >	::=	> | >= | = | /= | < |<=
16	< const >	::=	constant

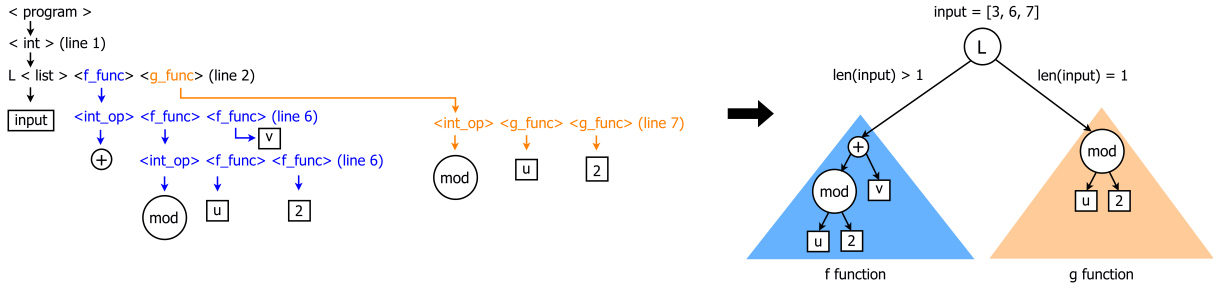

Fig. 2. The example of a program `(L input (+ (mod u 2) v) (mod u 2))` generated using the context-free grammar table.

"the best program". The following subsection introduces the grammar table of RSGP used for program initialization.

Algorithm 1 presents the pseudo-code for the evolution mechanism within a generation of RSGP. In this paper, the population is denoted as P. The number of offspring in each generation is represented by o. The X and y represent the example inputs and outputs, and w is a parameter of mRMR selection. We apply subtree crossover or subtree mutation to the selected programs. The subtree crossover follows Koza's GP [25]. The subtree mutation either replaces a subtree of the parent with a randomly initialized program or selects a subtree of the parent to replace a subtree of a randomly initialized program. To reduce search space, the evolution follows the STGP [14], which enforces data type constraints during the evolution. We use the exponential ranking selection [26], which selects a program based on its fitness rank. The probability of selecting the i^{th} best program from P is in proportion to $(0.9993)^i$ in RSGP.

B. Grammar Table of RSGP

Table II displays the context-free grammar of RSGP utilizing S-expression [27] for representation. As illustrated in Fig. 2, each non-terminal is replaced according to the production rules as shown in Table II. Finally, the program `(L input (+ (mod u 2) v) (mod u 2))` is generated.

`car` represents the first element of the list, and `cdr` represents the rest of the list. For instance, if the list `input` is [1, 2, 3], `(car input)` is 1, and `(cdr input)` is [2, 3]. `last` represents the last element of the list. `mapcar` is used to apply a specified function to each element of a list, producing a new list containing the results. `filter` is used to remove elements from a list that do not satisfy the given condition. `cons` is used to concatenate a value to the front of the list. `/=` represents not equal to. To prevent zero division errors, `floor` and `mod` (line 12) return 0 when the denominator is zero. The non-terminal symbols <λ>, <f_func>, and <g_func> represent integer-returning lambda expressions. The terminal symbols on the right of `::=`, such as `u`, `v`, or `idx`, are input arguments in the integer-returning lambda expressions mentioned earlier. The next subsection will detail `L`, `u`, `v`, `idx`, <f_func>, and <g_func>.

C. Recursive Function for Single-Layer Loop Simulation

To synthesize repetitive structure under the pure functional programming paradigm, we define `L` in our grammar table. The recursive definition of `L` is as follows:

$$L(input,\ idx,\ f,\ g) = \begin{cases} g(\text{car}(input)), & \text{len}(input) = 1 \\ f(\text{car}(input), L(\text{cdr}(input),\ idx{+}1,\ f,\ g)), & \text{otherwise,} \end{cases} \quad (1)$$

where $input$ represents an integer list, and idx represents the index variable. The definitions of `car` and `cdr` follow the last subsection. L is comprised of a base case g (triggered when input length is 1) and a recursive case f. This L simulates

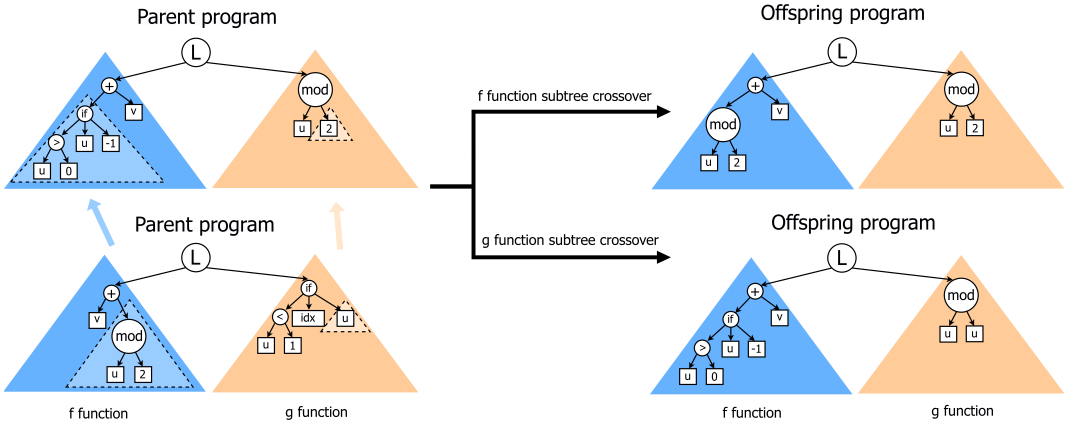

Fig. 3. The subtree crossover mechanism in L function.

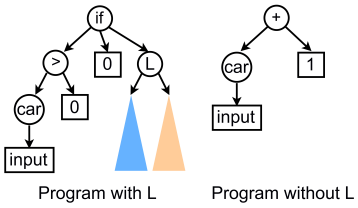

Fig. 4. The examples of the generated programs from our grammar table.

the single-layer loop behavior in imperative programming. Fig. 2 illustrates the expression trees for both functions, with f represented in blue and g in orange. The f function is `(+ (mod u 2) v)`, and the g function is `(mod u 2)` in Fig. 2. Here, `u` is the first element of the input, and `v` is the recursive result of the remaining input. Take the generated program in Fig. 2, which counts how many odd numbers in the input, for example, if *input* is [3, 6, 7], then evaluating L([3, 6, 7], 0, f, g) computes f(3, L([6, 7], 1, f, g)), which yields 2. This result is derived from adding 1 (the result of 3 mod 2) to `v`, where `v` is the outcome of L([6, 7], 1, f, g), which equals 1. Specifically, `v` is the result of recursion on the remaining input. L([6, 7], 1, f, g) computes as f(6, L([7], 2, f, g)), resulting in 1 since the value 6 is even, and L([7], 2, f, g) computes as 1 (the result of 7 mod 2).

In Table II, `<g_func>` specifies the base case options. It requires two input arguments: `u` and `idx`. `<f_func>` specifies the recursive case options. This function requires three input arguments: `u`, `v`, and `idx`. Fig. 3 shows the crossover mechanism in `L`, with both of the two parents belonging to the same part (either blue or orange). Fig. 4 illustrates `L` as an optional element in programs and emphasizes that `L` is not necessarily the root of programs.

D. mRMR Selection and Weight Adaptation

The mRMR algorithm is a feature selection technique used in data mining and machine learning. It aims to select a subset of features that maximizes relevance with the target variable while minimizing redundancy with the selected features. Inspired by this algorithm, we leverage $(\mu + \lambda)$ selection that selects $|P|$ programs from the $|P| + |O|$ candidates for the next generation. Fig. 5 illustrates an example to calculate the mRMR score. The mRMR score of a program in RSGP is determined by its relevance with the ground truth minus w times its redundancy with the already selected programs. The strategy is outlined in Algorithms 2 and 3. We get the output of each program in the population P with the example inputs X, and we measure the relevance with the example outputs y and redundancy with each other programs. Notably, unlike the traditional mRMR method, which utilizes mutual information, our method employs the absolute value of PCC to measure relevance and redundancy. Additionally, we introduce a parameter w to adjust the weight given to redundancy.

Algorithm 2: mRMRSelection

Input: P: population; O: offspring; X: inputs; y: outputs; w: mRMR redundancy weight;
Output: The selected population

```
Q ← P ∪ O
S ← empty set
for i = 1 to |P| do
    for j = 1 to |Q| do
        Calculate mRMRScore(Q_j, S, X, y, w) of Q_j (Algorithm 3)
    S_i ← get the program with the highest mRMRScore from Q
    Remove S_i from Q
return S
```

Our adaptation mechanism is inspired by the paper [28], which specifies the parameters for each subpopulation [29]. Algorithm 4 presents the adaptation framework. Algorithm 5 presents the adaptation mechanism. As depicted in Fig. 6, each generation randomly divides the population P into the three subpopulations P_{low}, P_{base}, and P_{high} with equal sizes, and evolves them independently. The adaptation step adjusts the parameter w based on subpopulation performance, aiming to

Algorithm 3: mRMRScore

Input: Q_j: a candidate program; S: the selected programs; X: inputs; y: outputs; w: mRMR redundancy weight;
Output: mRMR score

$relevance \leftarrow$ absolute value of PCC between the output of Q_j and y
$redundancy \leftarrow 0$
if $|S| > 0$ **then**
  for $i = 1$ *to* $|S|$ **do**
    Increase $redundancy$ by the absolute value of PCC between the output of Q_j and S_i
  Divide $redundancy$ by $|S|$
$score \leftarrow relevance - w \times redundancy$
return $score$

Algorithm 4: Framework of Adaptation

Input: P: population; O: offspring; X: inputs; y: outputs; w: mRMR redundancy weight;
Output: The best program

Randomly initialize population P
while $\neg$*ShouldTerminate* **do**
  $s \leftarrow$ The best fitness from P
  $P_{low}, P_{base}, P_{high} \leftarrow$ Random shuffle and divide P into three equally sized subpopulations
  for $i = 1$ *to* 10 **do**
    $P_{low} \leftarrow$ Evolution(P_{low}, $\frac{|P_{low}|}{2}$, X, y, $w - 0.1$)
    $P_{base} \leftarrow$ Evolution(P_{base}, $\frac{|P_{base}|}{2}$, X, y, w)
    $P_{high} \leftarrow$ Evolution(P_{high}, $\frac{|P_{high}|}{2}$, X, y, $w + 0.1$)
  $w \leftarrow$ Adaptation(P_{low}, P_{base}, P_{high}, s, w) (Algorithm 5)
  $P \leftarrow P_{low} \cup P_{base} \cup P_{high}$
return the best program in P after program simplification and linear scaling

find the optimal w for each problem. If none of the best fitness of any subpopulation is better than s which is the best fitness before evolving in this generation, w remains unchanged. If P_{high} achieves the best fitness, w is increased by 0.1; if P_{low} leads, w is reduced by 0.1.

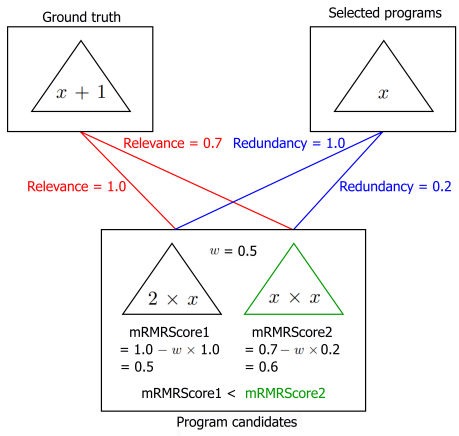

Fig. 5. An example of mRMR score calculation. Programs calculate the relevance with the ground truth and the redundancy with the selected programs. For simplicity, there is only one selected program in this example.

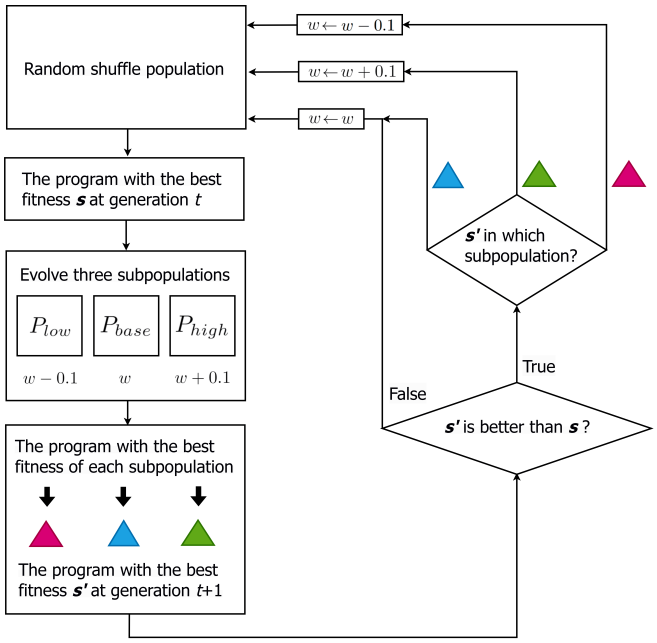

Fig. 6. The adaptation mechanism in RSGP.

E. Program Simplification and Linear Scaling

To improve the program generalizability and readability, simplification of the best program after the evolution is a commonly-used technique in program synthesis genetic programming [7], [8], [11]. We employ two approaches to simplify our program. The first method utilizes the local search strategy [7]: If there is a subprogram of a program with an output type identical to its own, and if the fitness of the subprogram does not worsen, the subprogram replaces the original program. The second one focuses on simplifying boolean statements by removing redundant code segments. For instance, when seeking the maximum of x_1 and x_2, the program `(if (or (> x1 x2) (> x1 x1)) x1 x2)` is generated. If the tentative replacement of `((or (> x1 x2) (> x1 x1))` with its subset `(> x1 x2)` does not worsen the fitness, we keep such replacement; we revert it otherwise.

The final step of RSGP involves applying linear scaling to fit the output of the best program to the ground truth. As shown in Equation 2, the closer the absolute value of PCC between the output of the program and the ground truth is to 1, the better the fitness is. Using PCC, however, may result in a perfectly

Algorithm 5: Adaptation

Input: P_{low}; P_{base}; P_{high}; s: The best fitness of the last generation; w: mRMR redundancy weight;
Output: The new weight

if *Both the best fitness of* P_{low} *and* P_{high} *are not better than* s **then**
  return w
if P_{low} *achieves the best fitness and* $w > 0$ **then**
  return $w - 0.1$
if P_{high} *achieves the best fitness and* $w < 1$ **then**
  return $w + 0.1$
return w

capable program (e.g., x) linearly deviated from the ground truth (e.g. $2x$), and hence linearly scaling is required.

III. Experiment Results

This section introduces test problems to assess the performance of our proposed method, followed by a description of the experimental setup. We then present the results, including an ablation study, and conclude with a discussion. The source code is available at GitHub[1].

A. Test Problems

PSB1 [30] and PSB2 [31] are two well-known general program synthesis benchmark suites in GP that collect introductory programming exercises. We select a subset of problems that meet the limitations of our method to test. We select CountOdds, LastIndexofZero from PSB1, and FuelCost, Luhn from PSB2. In addition, we present 4 novel problems, Minimum (MIN), Minimum positive integer (MPI), Minimum positive integer subtracts maximum positive integer (MPSMN), and Maximum positive integer times maximum positive integer (MPTMN) to further test the ability of looping and problem decomposition. The details of the 4 designed problems are provided in Table III.

B. Experiment Setup

The population size is 1000, and the maximum number of evaluations is 5×10^5 for each run. In order to divide the population into three equal subpopulations during adaptation, and ensure that the population size is divisible by 2, the actual size of each subpopulation is 332. Programs were limited to a maximum depth of 15. The crossover rate α is set to 0.1. The initial mRMR redundancy weight w is set to 0.9. f function, g function, and lambda function were limited to a maximum depth of 3. The fitness function e of a program is as follows:

$$e = \begin{cases} inf, & \text{if error occurs} \\ 1 - |r|, & \text{otherwise,} \end{cases} \tag{2}$$

[1]http://www.github.com/howard0027/RSGP

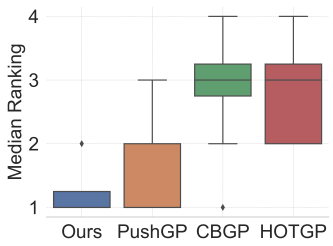

Fig. 7. Median ranking from 8 problems.

where r is the PCC between the output of the program and the ground truth, and inf is infinity. The value of inf is 2^{62} in RSGP. Error occurs if one of the following happens:

- The depth of the program exceeds 15.
- The depth of lambda expression exceeds 3.
- When the program does invalid instructions, such as accessing the first element of an empty list.
- The program executes over 5000 instructions.

If the $|e|$ of the best program on the train set is less than 10^{-12}, RSGP will stop early. A successful program represents the mean absolute error (MAE) of the best program is 0 on the test set after program simplification and linear scaling.

C. Results and Discussions

We compare RSGP with PushGP[2], CBGP[3], and HOTGP[4]. Apart from setting the maximum number of evaluations to 5×10^5, the other configurations remain default. We conduct 100 runs for each test problem. Table IV and Fig. 7 report the results. The results show that RSGP performs the best in terms of the number of successful programs on CountOdds and LastIndexofZero from PSB1, Luhn from PSB2, and 3 out of 4 designed problems. Additionally, the performance of RSGP lags only behind PushGP on FuelCost and MPTMN. The performance of RSGP on the MPTMN problem is much worse than on the MPSMN problem, even though they both can decompose into two subproblems, the minimum positive integer and the maximum negative integer, respectively. The possible reason is that the fitness of the subproblem on the MPTMN problem is not lower on the MPSMN problem. The fitness of the minimum positive integer is about 0.1376 on the MPSMN problem. However, the fitness of the same subproblem is about 0.2345 on the MPTMN problem. This result suggests that problem decomposition is more challenging on the MPTMN problem.

To assess the effect of program simplification, Table IV also presents the generalization rate before and after program simplification. The generalization rate is defined as the ratio of the number of runs that find a successful program to the number of runs that find the best program. The results indicate that program simplification enhances the generalization rate to 1.0 for all the 8 problems.

[2]https://github.com/lspector/Clojush
[3]https://github.com/erp12/cbgp-lite
[4]https://github.com/mcf1110/hotgp

TABLE III
THE DESCRIPTION AND DETAILS FOR EACH PROBLEM

Problem	Description	Input length	Train	Test
MIN	Given a list of integers, find the minimum value in the list. The constant range is [-100, 100].	[1, 50]	100	1000
MPI	Given a list of integers, find the minimum positive value in the list. The given list must contain at least one positive integer. The constant is 0.	[1, 50]	100	1000
MPSMN	Given a list of integers, find the difference from the minimum positive value and maximum negative value in the list. The given list must contain both positive integers and negative integers. The constant is 0.	[2, 50]	100	1000
MPTMN	Given a list of integers, find the product from the minimum positive value and maximum negative value in the list. The given list must contain both positive integers and negative integers. The constant is 0.	[2, 50]	100	1000

TABLE IV
THE NUMBER OF SUCCESSFUL PROGRAMS FOUND IN EACH PROBLEM, THE GENERALIZATION RATE AND THE PROGRAM SIZE OF RSGP BEFORE AND AFTER PROGRAM SIMPLIFICATION. THE BEST RESULTS OF EACH PROBLEM ARE UNDERLINED. VALUES IN BOLD REPRESENT THE PERFORMANCE IS STATISTICALLY SIGNIFICANTLY BETTER THAN THE SECOND-BEST ALGORITHM WITH A P-VALUE OF 0.05. OPTIMUM REPRESENTS THE MINIMUM PROGRAM SIZE FROM ALL OF THE SOLUTIONS. THE WORD "PS" DENOTES PROGRAM SIMPLIFICATION IN THIS TABLE.

Problem	Number of Successful Programs				Generalization Rate of RSGP		Program Size of RSGP		
	RSGP	PushGP	CBGP	HOTGP	Before PS	After PS	Optimum	Before PS	After PS
CountOdds	59	8	0	56	0.98	1.00	10	12	12
LastIndexofZero	**81**	35	9	0	1.00	1.00	14	18	18
FuelCost	30	**45**	0	9	0.97	1.00	12	15	15
Luhn	**5**	0	0	0	1.00	1.00	40	148	111
MIN	**100**	**100**	**100**	0	0.96	1.00	9	9	9
MNP	**93**	79	22	0	0.97	1.00	13	13	13
MPSMN	**80**	39	0	0	0.94	1.00	27	28	27
MPTMN	15	**38**	0	0	1.00	1.00	27	34	27

The other advantage of program simplification lies in reducing the program size by removing unnecessary code. Table IV also presents the smallest program size from the best programs before and after program simplification. Program size is the node count of the corresponding AST. The optimal size of a program represents the size is equal to the minimum program size of any solution. Before program simplification, RSGP finds programs with the optimal size on 2 out of 8 problems. After program simplification, RSGP finds programs with optimal size on 4 out of 8 problems. For instance, in solving the MIN problem, RSGP produces `(L input (if (and (/= u -68) (<= u v)) u v) u)`. After removing the spurious code segment `(/= u -68)` by program simplification, the result is `(L input (if (<= u v) u v) u)`.

RSGP finds a successful program for the Luhn problem in 5 out of 100 runs, while none of the compared algorithms ever solves this problem within 5×10^5 evaluations in 100 runs. To assess the capabilities of the compared algorithms in solving the Luhn problem, we parallelly execute 100 runs for 5 days per run. The average number of evaluations of PushGP, CBGP, and HOTGP are approximately 2.41×10^6, 3.76×10^7, and 1.56×10^6, respectively. The number of evaluations each run far exceeds 5×10^5, and yet none of them finds a successful program. Even though RSGP is subject to limitations in solving specific problems, the results show that RSGP is capable of solving the Luhn problem to some extent.

D. Ablation Study

To verify the effectiveness of mRMR selection with PCC and adaptation in RSGP, we conduct the ablation study involving 3 cases. Table V presents the number of successful programs found in each case. Case 1 employs $(\mu+\lambda)$ selection which selects the $|P|$ programs from the $|P|+|O|$ candidates based on their MAE. Case 2 utilizes mRMR feature selection and uses PCC in the fitness. Case 3 is the method we proposed.

The results show that Case 2, which incorporates mRMR selection and uses PCC in the fitness, improves the performance on 6 out of 8 problems compared to Case 1. The results in Case 2 indicate that selecting capable and diverse programs of mRMR selection improves the performance. Additionally, this improvement is particularly significant in the MPSMN and MPTMN problems, which are specially designed to demonstrate the problem decomposition ability. Such results suggest using mRMR with PCC encourages proper problem decomposition. However, the performance of FuelCost drops.

The possible reason for the drop in the performance of FuelCost is that mRMR avoids selecting programs for the next generation with similar outputs. Some programs contain capable substructures, but they may not be selected because their outputs are similar to the outputs of the selected programs. Therefore, Case 3 utilizes the adaptation mechanism to adjust the mRMR redundancy weight w to fit the needs of different problems. Case 3 leads to substantial improvement in FuelCost with the trade-off of minor drops on the other problems as shown in Table V when compared with Case 2. This trade-off is worthwhile and enables our method to achieve a higher success rate across a wide range of problems.

IV. CONCLUSION

This paper proposed RSGP, which utilized the recursive function L to simulate the single-layer loop behavior under the pure functional programming paradigm. To select the capable

TABLE V
THE NUMBER OF SUCCESSFUL PROGRAMS FOUND IN EACH CASE. THE BEST RESULTS OF EACH PROBLEM ARE UNDERLINED.

Case	Mechanism		Problems							
	mRMR with PCC	Adaptation	CountOdds	LastIndexofZero	FuelCost	Luhn	MIN	MNP	MPSMN	MPTMN
1			39	91	24	0	<u>100</u>	96	32	2
2	✓		<u>60</u>	<u>92</u>	2	<u>5</u>	<u>100</u>	<u>97</u>	<u>80</u>	<u>24</u>
3 (RSGP)	✓	✓	59	81	<u>30</u>	<u>5</u>	<u>100</u>	93	<u>80</u>	15

and diverse programs for the next generation, RSGP leveraged mRMR feature selection with PCC during the evolutionary process. The experiment results showed RSGP outperformed three SOTA algorithms in terms of the number of successful programs on CountOdds and LastIndexofZero from PSB1, Luhn from PSB2, and 3 out of 4 designed problems. The ablation study indicated that using mRMR with PCC does encourage proper problem decomposition with the trade-off of diminishing the search ability within a similar neighborhood. RSGP utilized an adaptation mechanism to balance the trade-off to automatically fit the needs of different problems.

For future work, we would like to explore more general inductive program synthesis methods, including nested loop synthesis, synthesizing the program whose output is not a single value, and enabling a larger grammar table size.

V. ACKNOWLEDGEMENTS

The authors would like to thank the support of the National Science and Technology Council in Taiwan under grant No. NSTC 112-2221-E-002-199 and National Taiwan University under grant No. 112L891103.

The authors would also like to thank the reviewer who shared the paper [32] which suggests eliminating the looping structure due to the difficulties of parallelization. Instead of simulating the looping structure, we would like to investigate generating programs with parallelizable repetitive structures, such as mapping functions.

REFERENCES

[1] C. David and D. Kroening, "Program synthesis: challenges and opportunities," *Philos. Trans. Roy. Soc. A: Math. Phys. Eng. Sciences*, vol. 375, no. 2104, p. 20150403, 2017.

[2] H. Lieberman, *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.

[3] S. Gulwani, "Dimensions in program synthesis," in *Symp. Princ. Pract. Declarative Program.*, 2010, pp. 13–24.

[4] D. Sobania, D. Schweim, and F. Rothlauf, "A comprehensive survey on program synthesis with evolutionary algorithms," *IEEE Trans. Evol. Comput.*, vol. 27, no. 1, pp. 82–97, 2022.

[5] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Foundations and Trends® in Program. Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[6] A. Albarghouthi, S. Gulwani, and Z. Kincaid, "Recursive program synthesis," in *Comput. Aided Verification*. Springer, 2013, pp. 934–950.

[7] M. C. Fernandes, F. O. de França, and E. Francesquini, "HOTGP—Higher-Order Typed Genetic Programming," in *Proc. Genetic Evol. Comput. Conf.*, 2023, pp. 1091–1099.

[8] L. Spector, "Autoconstructive evolution: Push, pushgp, and pushpop," in *Proc. Genetic Evol. Comput. Conf.*, 2001, pp. 137–146.

[9] T. Helmuth, L. Spector, N. F. McPhee, and S. Shanabrook, "Linear genomes for structured programs," *Genetic Program. Theory Pract. XIV*, pp. 85–100, 2018.

[10] T. Helmuth, N. F. McPhee, and L. Spector, "Program synthesis using uniform mutation by addition and deletion," in *Proc. Genetic Evol. Comput. Conf.*, 2018, pp. 1127–1134.

[11] E. Pantridge, T. Helmuth, and L. Spector, "Functional code building genetic programming," in *Proc. Genetic Evol. Comput. Conf.*, 2022, pp. 1000–1008.

[12] C. Ryan, J. J. Collins, and M. O. Neill, "Grammatical evolution: Evolving programs for an arbitrary language," in *Genetic Program.* Springer, 1998, pp. 83–96.

[13] S. Forstenlechner, D. Fagan, M. Nicolau, and M. O'Neill, "A grammar design pattern for arbitrary program synthesis problems in genetic programming," in *Genetic Program.* Springer, 2017, pp. 262–277.

[14] D. J. Montana, "Strongly typed genetic programming," *Evol. Comput.*, vol. 3, no. 2, pp. 199–230, 1995.

[15] E. Hemberg, J. Kelly, and U.-M. O'Reilly, "On domain knowledge and novelty to improve program synthesis performance with grammatical evolution," in *Proc. Genetic Evol. Comput. Conf.*, 2019, pp. 1039–1046.

[16] F. Garrow, M. A. Lones, and R. Stewart, "Why functional program synthesis matters (in the realm of genetic programming)," in *Proc. Genetic Evol. Comput. Conf. Companion*, 2022, pp. 1844–1853.

[17] J. Swan, K. Krawiec, and Z. A. Kocsis, "Stochastic program synthesis via recursion schemes," in *Proc. Genetic Evol. Comput. Conf. Companion*, 2019, pp. 35–36.

[18] E. Pantridge and L. Spector, "Code building genetic programming," in *Proc. Genetic Evol. Comput. Conf.*, 2020, pp. 994–1002.

[19] L. Spector and A. Robinson, "Genetic programming and autoconstructive evolution with the push programming language," *Genetic Program. Evolvable Machines*, vol. 3, pp. 7–40, 2002.

[20] E. Pantridge, T. Helmuth, and L. Spector, "Solving novel program synthesis problems with genetic programming using parametric polymorphism," in *Proc. Genetic Evol. Comput. Conf.*, 2023, pp. 1175–1183.

[21] A. Sabry, "What is a purely functional language?" *J. Functional Program.*, vol. 8, no. 1, pp. 1–22, 1998.

[22] D. R. Smith, "Top-down synthesis of divide-and-conquer algorithms," *Artif. Intell.*, vol. 27, no. 1, pp. 43–96, 1985.

[23] G. Hutton, "A tutorial on the universality and expressiveness of fold," *J. Functional Program.*, vol. 9, no. 4, pp. 355–372, 1999.

[24] H. Peng, F. Long, and C. Ding, "Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27, no. 8, pp. 1226–1238, 2005.

[25] J. R. Koza, "Genetic programming as a means for programming computers by natural selection," *Statist. Comput.*, vol. 4, pp. 87–112, 1994.

[26] T. Blickle and L. Thiele, "A comparison of selection schemes used in evolutionary algorithms," *Evol. Comput.*, vol. 4, no. 4, pp. 361–394, 1996.

[27] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part i," *Commun. ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[28] J. J. Grefenstette, "Optimization of control parameters for genetic algorithms," *IEEE Trans. Syst. Man Cybern.*, vol. 16, no. 1, pp. 122–128, 1986.

[29] J. E. Smith and T. C. Fogarty, "Operator and parameter adaptation in genetic algorithms," *Soft Comput.*, vol. 1, pp. 81–87, 1997.

[30] T. Helmuth and L. Spector, "General program synthesis benchmark suite," in *Proc. Genetic Evol. Comput. Conf.*, 2015, pp. 1039–1046.

[31] T. Helmuth and P. Kelly, "PSB2: the second program synthesis benchmark suite," in *Proc. Genetic Evol. Comput. Conf.*, 2021, pp. 785–794.

[32] F. F. de Vega, "Teaching programming in the 21st century," *J. Comput. Inf. Syst.*, vol. 63, no. 4, pp. 841–852, 2023.