# Knowledge-Driven Program Synthesis via Adaptive Replacement Mutation and Auto-constructed Subprogram Archives

Yifan He
*University of Tsukuba*
Tsukuba, Japan
he.yifan.xs@alumni.tsukuba.ac.jp

Claus Aranha
*University of Tsukuba*
Tsukuba, Japan
caranha@cs.tsukuba.ac.jp

Tetsuya Sakurai
*University of Tsukuba*
Tsukuba, Japan
sakurai@cs.tsukuba.ac.jp

*Abstract*—We introduce Knowledge-Driven Program Synthesis (KDPS) as a variant of the program synthesis task that requires the agent to solve a sequence of program synthesis problems. In KDPS, the agent should use knowledge from the earlier problems to solve the later ones. We propose a novel method based on PushGP to solve the KDPS problem, which takes subprograms as knowledge. The proposed method extracts subprograms from the solution of previously solved problems by the *Even Partitioning* (EP) method and uses these subprograms to solve the upcoming programming task using *Adaptive Replacement Mutation* (ARM). We call this method PushGP+EP+ARM. With PushGP+EP+ARM, no human effort is required in the knowledge extraction and utilization processes. We compare the proposed method with PushGP, as well as a method using subprograms manually extracted by a human. Our PushGP+EP+ARM achieves better train error, success count, and faster convergence than PushGP. Additionally, we demonstrate the superiority of PushGP+EP+ARM when consecutively solving a sequence of six program synthesis problems.

*Index Terms*—program synthesis, PushGP, adaptive replacement mutation, subprogram archive

## I. Introduction

*Program Synthesis* (PS) are techniques that automatically compose computer programs to solve a certain task. PS is useful in fields such as automatic bug fixing, automatic program completion, and low-level code development. PS is a key issue in Artificial General Intelligence [1]. *Genetic Programming* (GP) [2] is an Evolutionary Algorithm that searches computer programs by selecting and updating a population of program candidates. Some GP variants [3]–[5] can solve problems in a famous PS benchmark suite [6] efficiently.

However, the difference between GP and a human programmer is still obvious. As an Evolutionary Algorithm, GP heavily utilizes random sampling; while a human programmer does not write random programs. Humans write programs based on their knowledge, either the domain knowledge about the problem or the programming skills from previous experiences.

Recently, several studies [7]–[11] have attempted to incorporate knowledge in PS, improving the synthesis performance. However, some of these methods have drawbacks in requiring extra information [7], [8] and human efforts [11].

In our prior study, we proposed the *Adaptive Replacement Mutation* (ARM) [11]. The ARM is a mutation method designed for a well-known GP variant called PushGP [3]. ARM uses subprograms from an archive as knowledge, automatically selecting useful subprograms from the archive according to the search history. Although the ARM provides a way to use existing knowledge from an archive, the archive itself was made by a human. Moreover, it is questionable whether the subprograms written by humans are included in the programs generated by PushGP.

In this study, we focus on the task where an agent is required to solve a sequence of PS problems. The agent should learn knowledge from each problem in the sequence and apply this knowledge to improve its performance in later problems. Ideally, this procedure should not require human interference or extra external information. We call this task the ***Knowledge-Driven Program Synthesis*** (KDPS) problem (**Section III**).

We propose a novel method to solve the KDPS problem based on PushGP [3]. This method takes subprograms as knowledge. The proposed method consecutively solves programming tasks, extracts subprograms from the solution of solved problems, and uses subprograms to solve an upcoming problem. To extract subprograms, we propose *Even Partitioning* (EP) which divides a solution into several parts with equal lengths. To use these subprograms, we apply ARM [11]. We name our method PushGP+EP+ARM. The details of the proposed methods, including EP and ARM, are given in **Section IV**.

We analyze the proposed method in two KDPS tasks. The "composite task" (**Section V**) includes three "composite" PS problems. For each problem, PushGP+EP+ARM prepares the knowledge archive based on the component problems. The "sequential task" (**Section VI**) has six problems that must be solved in sequence, including the composite and component problems of the previous "composite task". PushGP+EP+ARM updates the knowledge archive at each step of the sequence. PushGP+EP+ARM achieves a better overall success rate and convergence speed in the composite task, and in the later stages of the sequential task, showing that it can create a useful knowledge archive. However, the comparison

with a human-curated archive shows that there is still room for improvement.

Our main contributions are as follows.

1) We introduce a new type of task called the KDPS problem. KDPS includes a sequence of single PS problems. The agent is required to solve the single PS problems, extract knowledge, and use it in the later PS problems.

2) We propose EP to extract subprograms from the solution of a solved problem. We propose PushGP+EP+ARM to solve KDPS problems based on EP and our previous work on ARM [11].

3) We provide our implementation of the proposed method based on PyshGP [12] and experimental scripts in an online repository [1]

## II. BACKGROUND

### A. Program Synthesis

PS, also known as automatic programming, focuses on building an intelligent agent that writes computer programs to solve specific tasks with minimal human effort. Programs are sequences of instructions and a task could be described as a set of I/O examples. For instance, a task that adds two integers could be describes as $\{$`[in=(1,1), out=2]`, `[in=(1,2), out=3], ...`$\}$.

Therefore, PS can be formalized as the optimization problem in (1), which searches a sequence of instructions to minimize the difference between actual and expected program outputs. In (1), $\mathbf{p}$ is a program while $P_{instr}$ includes all feasible programs based on the instruction set $instr$; $in_i$ and $out_i$ are the $i$-th I/O example.

$$\min_{\mathbf{p} \in P_{instr}} \Sigma_{i=1}^{N} ||\mathbf{p}(in_i) - out_i|| \qquad (1)$$

### B. PushGP

Since Koza's first work on his tree-based GP [2], several variants of GP [3], [13]–[15] have been applied to solve PS problems. Among these methods, we highlight PushGP [3], [5], [14], which generates programs based on a Turing-complete language called Push.

Push uses the list to store a program and runs it using multiple stacks of different data types based on the following rules.

1) To execute an instruction, the interpreter pops the required arguments from the corresponding stacks.

2) After executed the instruction, the results are pushed to the corresponding stacks.

3) If the interpreter cannot find enough arguments from the stacks, the instruction will be skipped.

Based on the third rule, any sequence of Push instructions can form a valid Push program. Helmuth et al. proposed a variant of PushGP using lexicase selection and uniform mutation by addition and deletion (UMAD) [3]. This variant was further improved by applying down-sampled lexicase selection [5].

### C. Incorporating knowledge in Program Synthesis

An obvious difference between GP and human programmers is that a human can learn knowledge from his experiences and use knowledge in upcoming tasks. Several recent works have attempted to incorporate knowledge in PS [7]–[11], [16]. Some studies require extra information such as text description [7] and human-written code [8], [11], [16].

Helmuth et al. proposed to transfer the instructions from the solution of other problems to construct the instruction set of PushGP [9]. Wick et al. proposed to use the whole individuals of a problem as a part of the initial population when solving similar problems [10]. Both studies [9], [10] extract knowledge from the solutions of the problems that have been solved by GP.

Recently, we came up with a mutation method that allows using subprograms in a prepared archive with PushGP [11]. Compared with the study by Helmuth [9], a subprogram can capture more information from the solution than a single instruction. Compared to Wick's study [10], the way to use an external archive of knowledge might be more efficient when the number of the problems to transfer is large. However, in our prior study [11], the subprogram archives are extracted by hand from human-written solutions. This step is non-trivial and requires a lot of human effort. Also, considering the case where a GP is able to extract knowledge from programs it wrote in the past, it is questionable whether the source programs would look similar to human written ones.

## III. KNOWLEDGE-DRIVEN PROGRAM SYNTHESIS

Human programmers can learn from the programming problems that they have solved before and apply what they have learned to solve the upcoming problems. We suggest that the ability of learning is vital for generating complex programs.

Therefore, we introduce a type of task where an agent is required to solve a sequence of PS problems. Usually, these PS problems are related to each other. When solving a new problem, the agent should use the knowledge learned from previously solved problems. We call this type of task *Knowledge-Driven Program Synthesis* (KDPS).

The formalization of KDPS is presented in (2). $S^j$ is one of the $M$ PS problems in (1) to solve, containing $N_j$ pairs of I/O. $\hat{\mathbf{p}}^j$ and $K^j$ are the solution program and the knowledge that learned from solving $S^j$, respectively. $S^j$ is solved by Solve$(\cdot)$ based on the knowledge from the previously solved problems $K^{j-1} \cup \ldots K^0$. $K^0$ is the initial knowledge before solving the first problem $S^1$. By default, $K^0$ is empty.

$$\begin{cases} & \hat{\mathbf{p}}^j, K^j \leftarrow \text{Solve}(S^j | K^{j-1} \cup \cdots \cup K^0) \\ s.t. & S = \{S^1, \ldots, S^M\} \\ & S^j = \{(in_1^j, out_1^j), \ldots, (in_{N_j}^j, out_{N_j}^j)\} \\ & K^0 = \emptyset \end{cases} \qquad (2)$$

To learn knowledge from solving a problem, it is not necessary to use its solution program. However, as an initial
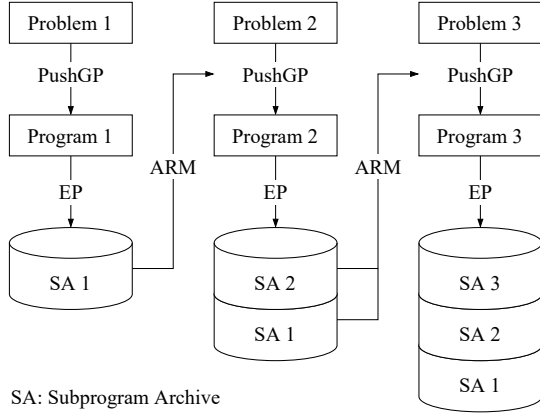
Fig. 1. An example of solving Knowledge-Driven Program Synthesis with the proposed method. The method solves problems with PushGP, extract subprograms with *Even Partitioning* (EP), and uses subprograms with *Adaptive Replacement Mutation* (ARM).

step, we assume all the knowledge from solving $S^j$ are extracted from its solution programs $\hat{\mathbf{p}}^j$ as in (3).

$$K^j \leftarrow \text{Extract}(\hat{\mathbf{p}}^j) \qquad (3)$$

## IV. PROPOSED METHOD

To solve KDPS problems, we propose a method based on PushGP [3] to consecutively solve programming tasks, extract knowledge from the solutions, and utilize knowledge in the next problem. This method is entirely automated.

We use subprograms, the sub-sequences of Push instructions, as knowledge. The subprograms hold partial information about the original program. Moreover, any sequence of Push instructions is valid to run. Therefore, we can easily take a subprogram and use it in a different program.

**Fig. 1** illustrates an example of solving a KDPS problem (a sequence of three PS problems) with our proposed method. Our method solves Problem 1 with PushGP [3] and extracts Subprogram Archive (SA) 1 from its solution (Program 1) with *Even Partitioning* (EP). This archive is used by *Adaptive Replacement Mutation* (ARM) [11] when searching for the solution of Problem 2. Similarly, we use SA 1 and SA 2 when solving Problem 3. The details of EP and ARM are provided in the next two subsections.

### A. Even Partitioning

EP is a simple method that divides a program into $n$ parts with equal lengths. For example, a program with 15 instructions is divided into subprograms with lengths of (3, 3, 3, 3, 3) if $n = 5$; the same program is divided into subprograms with lengths of (4, 4, 4, 3) if $n = 4$.

Before dividing the solution program, a simplification operation is performed to remove the redundant instructions (i.e., instructions without enough arguments to execute). After the dividing step, the subprograms are stored into an archive for the future use.

---

**Algorithm 1** PushGP with Adaptive Replacement Mutation

**Require:** subprogram archive $K$ where every subprogram $\mathbf{k}$ holds a quality metric $Q_{\mathbf{k}} = 0$;

1: $P \leftarrow$ initialize_population();
2: **while** termination criteria is not satisfied **do**
3:     $P' \leftarrow \emptyset$;
4:     **for** $i \leftarrow 1$ to $|P|$ **do**
5:        $\mathbf{p} \leftarrow$ lexicase_selection($P$);
6:        **if** rand() $< r_{arm}$ **then**
7:           **if** rand() $< r_{prop}$ **then**
8:              $\mathbf{k} \leftarrow$ proportional_selection($K$);
9:           **else**
10:             $\mathbf{k} \leftarrow$ random_selection($K$);
11:           $\mathbf{p}' \leftarrow$ replacement_mutation($\mathbf{p}, \mathbf{k}$);
12:           **if** $f(\mathbf{p}') \prec f(\mathbf{p})$ **then**
13:             $Q_{\mathbf{k}} \leftarrow Q_{\mathbf{k}} + 1$;
14:        **else**
15:           $\mathbf{p}' \leftarrow$ umad_mutation($\mathbf{p}$);
16:        $P' \leftarrow P' \cup \{\mathbf{p}'\}$;
17:     $P \leftarrow P'$;

---

### B. Adaptive Replacement Mutation

ARM [11] is a mutation method proposed for PushGP, which incorporates a prepared archive of subprograms during the search. ARM contains a Replacement Mutation (RM) and an adaptive strategy.

RM requires a parent candidate (of length $l_1$) from the PushGP population and a subprogram (of length $l_2$) from the prepared archive. RM replaces a random partition (of length $l_2$) of the parent candidate using the subprogram. If $l_1 < l_2$, the entire parent is replaced by the subprogram.

The adaptive strategy in ARM is designed to automatically select helpful subprograms when an archive contains both helpful and unhelpful subprograms. The idea is similar to parameter adaptation in many Self-adaptive Evolutionary Algorithms such as JADE [17]. Originally, ARM collects and stores the helpful subprograms that improve the parent candidates by RM into a working archive. This working archive, which consists of helpful subprograms, is used as one of the sources to select subprograms in the later generations.

In this study, we use a slightly different implementation of ARM from the original [11], using proportional selection to select subprograms rather than collecting subprograms in an extra archive. The proportion to select a subprogram depends on how many times that it improves parents.

We provide the pseudo code of PushGP (using lexicase selection and UMAD mutation) with ARM in **Algorithm 1**. $Q_{\mathbf{k}}$ is the count that a subprogram $\mathbf{k}$ improves the parents during the search. The probability in proportional selection is computed as in (4).

$$p(\mathbf{k}) = \frac{Q_{\mathbf{k}}}{\Sigma_{\mathbf{k}_i \in K} Q_{\mathbf{k}_i}} \qquad (4)$$

$r_{arm}$ is the probability to perform ARM and $r_{prop}$ is the probability to perform the proportional selection of subpro-

grams. In Line 12 of **Algorithm 1**, the symbol "$\prec$" means "better than". In our implementation, a solution is better than another if it solves more I/O cases (i.e., contains more "0" in its error vector).

In some cases, the subprograms may include more inputs than the current problem (e.g., a subprogram contains `input_3` while the current problem only takes two inputs). We replace the input in the subprograms with a random input of the current problem.

## V. Experiment I: Composite KDPS Task

In this experiment, we focus on an intermediate step of the KDPS problem (**Fig. 1**). That is, to solve a composite problem with our proposed method after solving its sub-problems.

### A. Comparison methods

We compare the following three methods.

- **PushGP+EP+ARM**: PushGP with ARM; the subprogram archives are extracted using EP.
- **PushGP+HP+ARM**: PushGP with ARM; the subprogram archives are extracted by human (HP: human partitioning).
- **PushGP**: the original PushGP [3].

### B. Experimental procedures

We use PushGP [3] to solve three problems in PSB1 [6]. They are "small or large" (SL), "compare string lengths" (CSL), and "median" (MD). We take the best and shortest solutions among 25 runs (after 5000 steps of simplification) of the three problems to generate subprogram archives. For PushGP+EP+ARM, We use EP to get five equal-length subprograms for every best and shortest solution automatically. The subprograms used by PushGP+HP+ARM are partitions of the same solutions, however, devised by a human.

We then solve the composite problems of SL, CSL, and MD. When solving a composite problem, PushGP+EP+ARM and PushGP+HP+ARM will use subprogram archives generated from solutions of the corresponded sub-problems by EP and HP, respectively. We compare the three methods on three composite problems.

- **Median String Length (MDSLEN)**: given 3 strings, print the median of their lengths.
- **Small or Large Median (SLMD)**: given 4 integers $a$, $b$, $c$, $d$, print "small" if median$(a,b,c) < d$ and "large" if median$(a,b,c) > d$ (and nothing if median$(a,b,c) = d$).
- **Small or Large String (SLSTR)**: given a string $n$, print "small" if len$(n) < 100$ and "large" if len$(n) \geq 200$ (and nothing if $100 \leq$ len$(n) < 200$).

MDSLEN is the composite problem of MD and CSL; SLMD is composed from SL and MD; SLSTR is a composite of SL and CSL.

### C. Parameter settings

For all three methods, we use a population size of 1000 and a maximum generation of 300. The UMAD mutation in all three methods is set with addition rate of 0.09 and deletion rate of 0.0826 based on Helmuth's study [3]. For PushGP+EP+ARM and PushGP+HP+ARM, the rate to perform ARM $r_{arm}$ is 0.1 and the rate to perform the proportional selection of subprograms $r_{prop}$ is 0.5 based on the original paper of ARM [11]. For every algorithm, we run 25 repetitions on every problem.

### D. Experimental results

**Fig. 2** presents the error in the training period of the three methods in 25 runs. The value on a line segment is the p-value of the Wilcoxon rank sum test between two groups. The p-value is marked with an asterisk and red color if the difference between two groups is significant with a 95% family confidence level (i.e., the individual confidence level is computed by Šidák correction[2]). The proposed PushGP+EP+ARM outperforms the original PushGP with a significant difference in the training error. However, compared with PushGP+HP+ARM, the difference is not statistically significant.

**Fig. 3** shows the success counts in the test period of the three comparison methods in 25 runs. We count a run as a successful run only when it passes all I/O cases in both training and testing data. In **Fig. 3**, the number above a bar is the success count; the value on a line segment is the p-value of Fisher's exact test between two groups. The p-value is marked with an asterisk and red color if the difference between two groups is significant with a 95% family confidence level (i.e., the individual confidence level is computed by Šidák correction[2]). Compared to PushGP, our PushGP+EP+ARM achieves higher success counts, however, without statistical significance. Compared to the method using human-made subprograms (PushGP+HP+ARM), our proposed method achieves a lower success count on MDSLEN, a higher success count on SLMD, and an equal success count on SLSTR. However, these differences are not significant.

We provide a comparison of the best train error by generations in **Fig. 4**. PushGP+EP+ARM holds a much faster convergence speed compared to PushGP; however, it is slightly slower than PushGP+HP+ARM.

In the case of solving the sub-problems and then the composite problems, PushGP+EP+ARM achieves a better performance in train error, success count, and faster convergence, compared to the original PushGP [3]. However, its performance in both success count and convergence speed is worse than PushGP+HP+ARM without statistical significance.

## VI. Experiment II: Sequential KDPS Task

In this second experiment, we test the entire KDPS process in **Fig. 1**. That is, using PushGP+ARM+EP to solve a sequence

---

[2]The individual confidence levels in **Section V** and **Section VI** are 99.4% and 99.1%, respectively.
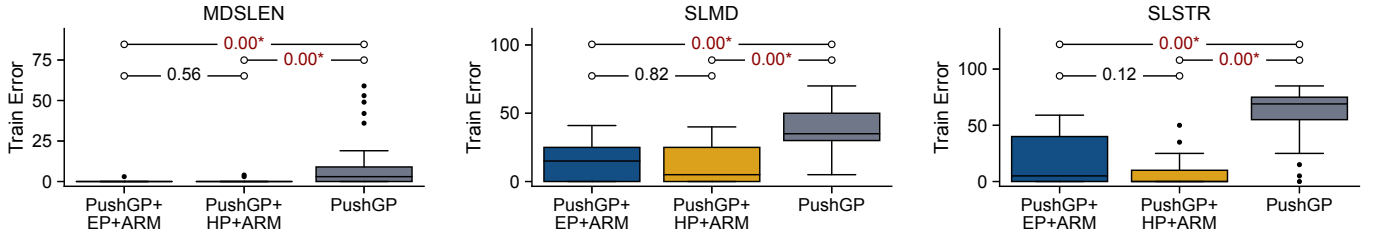
Fig. 2. **Experiment I**: Train error in 25 runs. The value on a line segment is the p-value of Wilcoxon rank sum test between two groups. The p-value is marked with an asterisk and red color if the difference between two groups is significant. PushGP+ARM+EP holds a significantly lower train error than PushGP and a comparable train error compared to PushGP+ARM+HP.
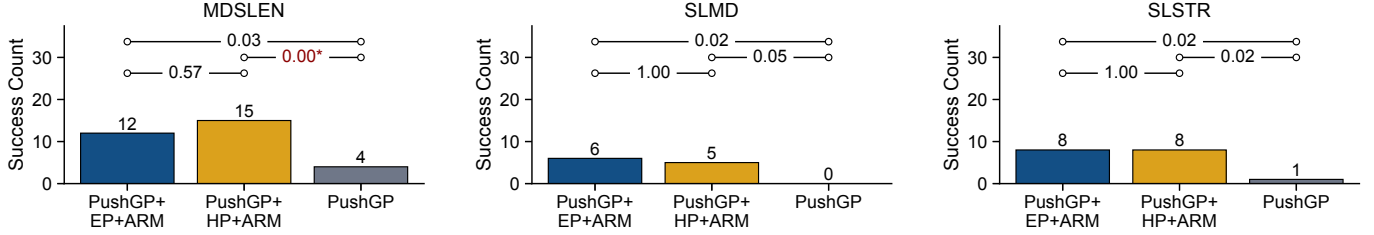


Fig. 3. **Experiment I**: Test success count in 25 runs. The number above a bar is the success count. The value on a line segment is the p-value of Fisher's exact test between two groups. The p-value is marked with an asterisk and red color if the difference between two groups is significant. PushGP+ARM+EP holds higher success counts than PushGP but without statistical significance.
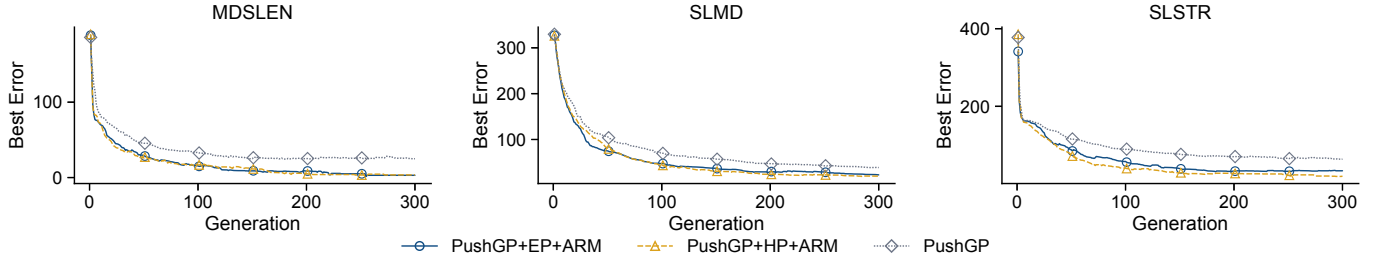


Fig. 4. **Experiment I**: Average of the best train error in the population by generations. PushGP+ARM+EP converges much faster than PushGP but slightly slower (SLSTR) or at the similar speed (MDSLEN and SLMD) compared to PushGP+ARM+HP.

of problems. They are the six problems in the last experiment, namely SL, CSL, MD, MDSLEN, SLMD, and SLSTR. Every time a problem is solved, we extract subprograms from its solution and store them in the archive. This archive will be used when solving the next problem.

### A. Comparison methods

- **PushGP+EP+ARM**: solving a sequence of PS problems with PushGP+EP+ARM in the way as in **Fig. 1**; every time a problem is solved, the subprograms are extracted by EP from its solution and added to the archive. This archive is used by PushGP+ARM when solving the next problem.
- **PushGP**: solving a sequence of PS problems independently using PushGP [3].

### B. Experimental procedures

We solve six problems, namely MD, CSL, SL, MDSLEN, SLMD, and SLSTR. The first three problems are from PSB1 [6]. They do not share any sub-problems. The last three problems are the composite problems of MD, CSL, and SL (**Section V**). Any pair of the three composite problems share a sub-problem.

For PushGP+EP+ARM, we run a procedure as in **Fig. 1**. We solve the first problem (MD) with the original PushGP (i.e., PushGP+ARM with an empty archive) and the rest problems with PushGP+ARM. We initialize an empty subprogram archive when solving the first problem. For every problem, we run 25 repetitions and take the best and shortest program. We use EP to extract five subprograms from the best and shortest programs. These subprograms are stored in the archive that

we initialized before and used in solving the later problems by PushGP+ARM. For PushGP, we solve the six problems independently in the same order with PushGP+EP+ARM. However, no subprograms are stored and used.

We provide results of solving the six problems in two different orders. Order 1 solves simple problems at first and later harder ones while Order 2 is a reverse order of Order 1.

- **Order 1**: MD→CSL→SL→MDSLEN→SLMD→SLSTR
- **Order 2**: SLSTR→SLMD→MDSLEN→SL→CSL→MD

We use the same parameter settings as in **Section V**. We present the results and the statistical test in a similar manner as in **Section V**.

### C. Experimental results of Order 1

In **Fig. 5**, PushGP+EP+ARM holds a significantly lower train error than PushGP on the three composite problems, while the difference on the rest three problems is not significant. **Fig. 6** shows the test success count of PushGP+EP+ARM and PushGP. The test success count of PushGP+EP+ARM is lower than PushGP on CSL and SL but higher than PushGP on MSDLEN, SLMD, and SLSTR. All these difference is not statistically significant through Fisher's exact test. On MD, the difference between the two methods is very small since it is solved by two equivalent methods. **Fig. 7** provides the best error in the population by generations of both methods. It is obvious that PushGP+EP+ARM converges faster than PushGP on all problems except MD.

### D. Experimental results of Order 2

In **Fig. 8**, PushGP+EP+ARM holds a lower train error than PushGP on all problems without statistical significance. **Fig. 9** shows the test success count of PushGP+EP+ARM and PushGP. The test success count of PushGP+EP+ARM is higher than PushGP on on most of the problems except MD. Especially on CSL, PushGP+EP+ARM gets 16 success while PushGP only gets 8. However, all these difference is not statistically significant through Fisher's exact test. **Fig. 10** provides the best error in the population by generations of both methods. It is obvious that PushGP+EP+ARM converges faster than PushGP on all problems except SLMD.

Therefore, when solving a sequence of problems, PushGP+EP+ARM achieves a better optimization performance (i.e., train error and convergence speed). This performance finally leads to a higher test success count, however, without statistical significance.

### E. Discussion

We find that the PushGP+EP+ARM in **Section V** is better than the PushGP+EP+ARM in **Section VI** with Order 1, in terms of test success on the three composite problems. Though their algorithms are the same, they have at least two differences.

1) In **Section VI**, PushGP+EP+ARM adds five subprograms to the archive after solving one problem. Therefore, the size of the archive is 15, 20, and 25 when solving MDSLEN, SLMD, and SLSTR, respectively. However, in **Section V**, PushGP+EP+ARM uses archives with 10 subprograms (five for one sub-problem). A larger subprogram archive makes it harder to select helpful subprograms by the adaptive strategy in **Algorithm 1**.

2) CSL and SL are solved in different conditions in the two experiments. In **Section VI**, CSL is solved with subprograms from MD; SL is solved with subprograms from MD and CSL. However, no subprogram is used when solving MD, CSL, and SL in **Section V** (i.e., they are solved by the original PushGP [3]). Moreover, MD, CSL, and SL do not share the same sub-problems. Therefore, solving CSL and SL with PushGP+EP+ARM is not as good as with the original PushGP (as shown in the first three subfigures of **Fig. 6**). Thus, the subprograms extracted in **Section VI** is not as good as in **Section V**. These subprograms further influence the performance of PushGP+EP+ARM in **Section VI** in solving the later problems MDSLEN, SLMD, and SLSTR. This issue is called "negative transfer".

In **Section VI** (problems in Order 2), we find that PushGP+EP+ARM holds a lower success count on MD. However, on MD, the training error of all runs with PushGP+EP+ARM is 0 (**Fig. 8**). Moreover, **Fig. 10** shows PushGP+EP+ARM converges much faster than PushGP. This observation may indicate an over-fitting issue with the proposed method.

### VII. Conclusions

In this study, we introduced a problem called Knowledge-Driven Program Synthesis (KDPS) problem. KDPS requires an agent to solve a sequence of related PS problems. To solve KDPS, we proposed a method based on PushGP [3]. This method consecutively solves programming tasks, extracts subprograms from the solutions as knowledge, and uses these subprograms to solve the next problem. To extract subprograms from the solution of a solved problem, we proposed the Even Partitioning (EP) method; to use these subprograms, we applied Adaptive Replacement Mutation (ARM) [11].

We compared our proposed method (PushGP+EP+ARM) with the original PushGP [3] and a method extracting subprograms by humans (PushGP+HP+ARM). Our PushGP+EP+ARM achieved a significantly better train error, success count, and convergence speed than PushGP. The performance of our proposed method is slightly worse than PushGP+HP+ARM. We further compared our PushGP+EP+ARM with the original PushGP in solving a sequence of problems. Our method achieved a better train error and convergence speed. Our PushGP+EP+ARM also holds a higher test success count, however, without statistical significance.

The current method to automatically construct the subprogram archive is rather naive. We would like to improve this method in our future work. In the discussion section in **Section VI**, PushGP+EP+ARM has limitations in dealing with the growing subprogram archive after solving more problems. Moreover, PushGP+EP+ARM also suffers from the "negative
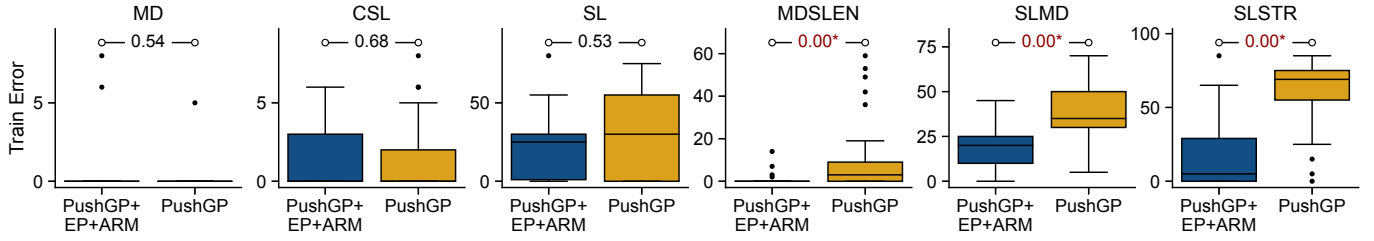
Fig. 5. **Experiment II, Order 1**: Train error in 25 runs. The value on a line segment is the p-value of Wilcoxon rank sum test between two groups. The p-value is marked with an asterisk and red color if the difference between two groups is significant. PushGP+EP+ARM holds a significantly lower train error than PushGP on MDSLEN, SLMD, and SLSTR.
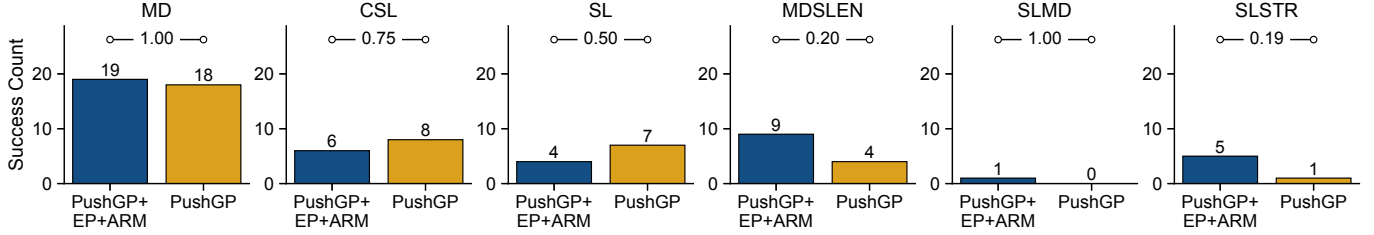


Fig. 6. **Experiment II, Order 1**: Test success count in 25 runs. The number above a bar is the success count. The value on a line segment is the p-value of Fisher's exact test between two groups. The p-value is marked with an asterisk and red color if the difference between two groups is significant. PushGP+EP+ARM holds a higher success count than PushGP on MD, MDSLEN, SLMD, and SLSTR; however, without statistical significance.
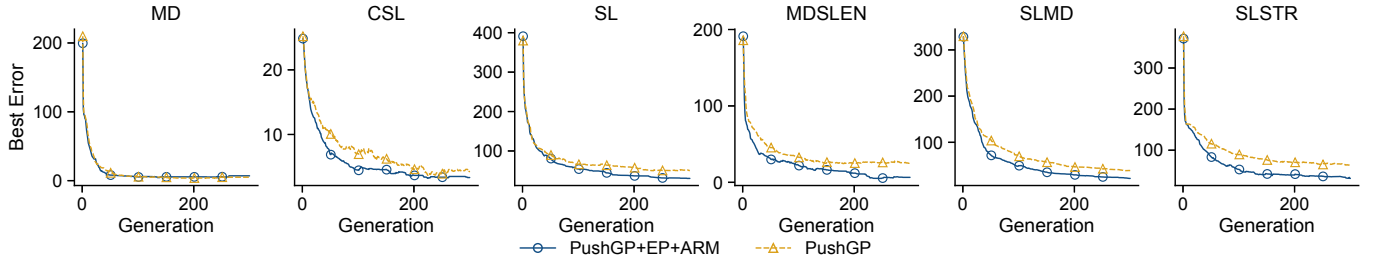


Fig. 7. **Experiment II, Order 1**: Average of the best train error in the population by generations. PushGP+EP+ARM converges much faster than PushGP on the five problems except MD.

transfer" of the subprograms from the unrelated problems. A part of our future work is to fix these issues. Methods such as a more efficient adaptation or filtering strategy on subprograms are promising to solve the two limitations. Additionally, the strategy of extracting and using knowledge could be applied to problems other than program synthesis, such as training soft robotics [18].

REFERENCES

[1] M. O'Neill and L. Spector, "Automatic programming: The open issue?" *Genetic Programming and Evolvable Machines*, vol. 21, no. 1, pp. 251–262, 2020.

[2] J. Koza, "On the programming of computers by means of natural selection," *Genetic programming*, 1992.

[3] T. Helmuth, N. F. McPhee, and L. Spector, "Program synthesis using uniform mutation by addition and deletion," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2018, pp. 1127–1134.

[4] J. Kelly, E. Hemberg, and U.-M. O'Reilly, "Improving genetic programming with novel exploration-exploitation control," in *European Conference on Genetic Programming*. Springer, 2019, pp. 64–80.

[5] T. Helmuth and L. Spector, "Explaining and exploiting the advantages of down-sampled lexicase selection," in *ALIFE 2020: The 2020 Conference on Artificial Life*. MIT Press, 2020, pp. 341–349.

[6] T. Helmuth and L. Spector, "General program synthesis benchmark suite," in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, 2015, pp. 1039–1046.

[7] E. Hemberg, J. Kelly, and U.-M. O'Reilly, "On domain knowledge and novelty to improve program synthesis performance with grammatical evolution," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2019, pp. 1039–1046.

[8] D. Sobania and F. Rothlauf, "Teaching gp to program like a human software developer: using perplexity pressure to guide program synthesis approaches," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2019, pp. 1065–1074.

[9] T. Helmuth, E. Pantridge, G. Woolson, and L. Spector, "Genetic source sensitivity and transfer learning in genetic programming," in *ALIFE 2020: The 2020 Conference on Artificial Life*. MIT Press, 2020, pp.
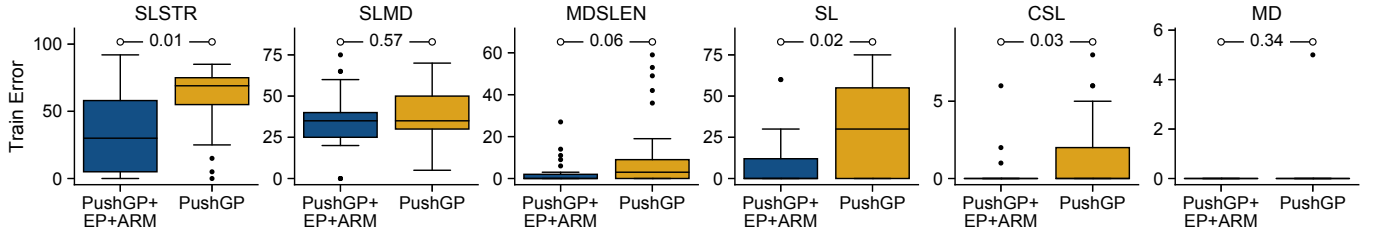
Fig. 8. **Experiment II, Order 2**: Train error in 25 runs. The value on a line segment is the p-value of Wilcoxon rank sum test between two groups. The p-value is marked with an asterisk and red color if the difference between two groups is significant. PushGP+EP+ARM holds a lower train error than PushGP on all the problems without statistical significance.
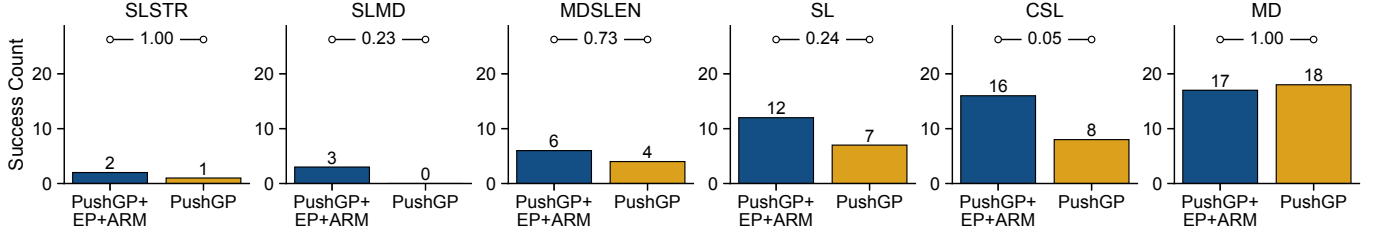


Fig. 9. **Experiment II, Order 2**: Test success count in 25 runs. The number above a bar is the success count. The value on a line segment is the p-value of Fisher's exact test between two groups. The p-value is marked with an asterisk and red color if the difference between two groups is significant. PushGP+EP+ARM holds a higher success count than PushGP on SLSTR, SLMD, MDSLEN, SL, and CSL; however, without statistical significance.
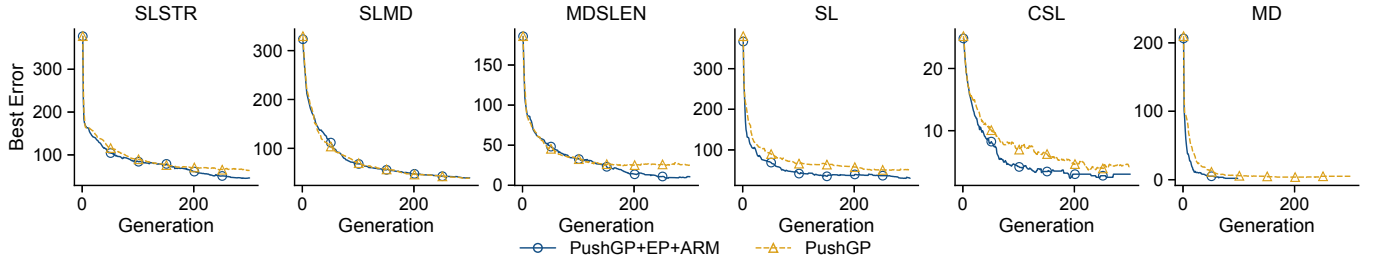


Fig. 10. **Experiment II, Order 2**: Average of the best train error in the population by generations. PushGP+EP+ARM converges much faster than PushGP on the five problems except SLMD.

303–311.

[10] J. Wick, E. Hemberg, and U.-M. O'Reilly, "Getting a head start on program synthesis with genetic programming," in *European Conference on Genetic Programming (Part of EvoStar)*. Springer, 2021, pp. 263–279.

[11] Y. He, C. Aranha, and T. Sakurai, "Incorporating sub-programs as knowledge in program synthesis by pushgp and adaptive replacement mutation," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2022.

[12] E. Pantridge and L. Spector, "Pyshgp: Pushgp in python," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2017, pp. 1255–1262.

[13] M. O'Neill and C. Ryan, "Grammatical evolution," *IEEE Transactions on Evolutionary Computation*, vol. 5, no. 4, pp. 349–358, 2001.

[14] L. Spector, "Autoconstructive evolution: Push, pushgp, and pushpop," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, vol. 137, 2001.

[15] S. Forstenlechner, D. Fagan, M. Nicolau, and M. O'Neill, "Extending program synthesis grammars for grammar-guided genetic programming," in *International Conference on Parallel Problem Solving from Nature*.

Springer, 2018, pp. 197–208.

[16] J. Lehman, J. Gordon, S. Jain, K. Ndousse, C. Yeh, and K. O. Stanley, "Evolution through large models," *arXiv preprint arXiv:2206.08896*, 2022.

[17] J. Zhang and A. C. Sanderson, "Jade: adaptive differential evolution with optional external archive," *IEEE Transactions on evolutionary computation*, vol. 13, no. 5, pp. 945–958, 2009.

[18] F. Pigozzi, Y. Tang, E. Medvet, and D. Ha, "Evolving modular soft robots without explicit inter-module communication using local self-attention," *arXiv preprint arXiv:2204.06481*, 2022.