AdaptGen: A Problem-Adaptive Solution Template Generation Technique for Online Programming Platforms

Guowei Zhang, Weiqin Zou, Xiaowei Zhang, Jingxuan Zhang, and Jifeng Xuan

***Abstract*—Online programming platforms that offer various programming tasks play a crucial role in helping programmers enhance their coding skills. Programming tasks posted for different application scenarios may follow the same or similar programming patterns in their solutions. This leads programmers to repeatedly write not just the core code of problem-solving, but also the same basic framework and some peripheral code (such as variable declarations and input/output handling) that are needed to make their solutions executable. Repeatedly writing boilerplate or well-mastered algorithmic frameworks wastes time and adds little value for programmers focused on skill-specific practice. Toward this, we propose to develop AdaptGen, a problem-adaptive code template generation method for online programming platforms. AdaptGen analyzes and extracts solution patterns from various programming problem solutions (i.e., code accepted by online programming platforms) and generates templates tailored to each problem. More specifically, AdaptGen is built on genetic programming and uses a linear hashing sequence encoding strategy to represent solutions. It incorporates selection, crossover, and de-duplication operators to maintain diversity in the evolution process, and a fitness function tailored to generate solution templates. These templates are then abstracted and structured with a flexible core-code-hiding mechanism, enabling programmers of different experience levels to practice efficiently. We evaluated AdaptGen using two datasets from LeetCode and NowCoder, containing a total of 997 tasks and over 3,200 solution categories. Results show that AdaptGen successfully generates usable templates for 77%-84% of solution categories, with 80% of the templates performing well in manual evaluations. It also outperforms seven advanced representative large language models (LLMs), achieving the best overall performance in template quality, consistency, and generation efficiency. To validate AdaptGen's effectiveness in real-world programming environments, we further conduct a user study involving live coding practice by programmers in online programming platforms, which effectively demonstrated its utility in practical application scenarios.**

***Index Terms*—Online programming platforms, solution template, adaptive generation, genetic algorithm.**

I. Introduction

SOLVING programming problems is a proven method to build programmers' programming mindset and improve coding skills [1], [2], [3]. Online coding platforms such as LeetCode[1], Codeforces[2], and NowCoder[3] offer a wide range of programming problems (a.k.a, programming tasks) whose solutions involve the use of various algorithms and data structures, attracting a global community of programmers and gradually becoming as an important media for corporation recruitment of technical talents. It is common to see programming tasks posted for different application scenarios whose solutions actually embed the use of the same/similar algorithms or data structures (which to some extent supports programmers to check whether they fully grasp them) [4].

In this case, programmers need not only to implement the core problem-solving logic, but also to repeatedly write boilerplate code, such as variable declarations, input/output statements, and basic algorithmic frameworks that programmers may already be very familiar with. The repetitive nature of writing these boilerplate segments would detract programmers from learning efficiency and offer limited help in skill development. By flexibly integrating these repetitive components into reusable solution templates, programmers of different expertise can focus more on skill-specific development programming practice.

Some platforms, like LeetCode, provide basic solution frameworks[4], but these are often too general, still requiring programmers to write a significant amount of boilerplate code. While recent Large Language Models (LLMs) have demonstrated impressive end-to-end code generation capabilities, they

Received 30 June 2025; revised 18 April 2026; accepted 5 May 2026. Date of publication 12 May 2026; date of current version 1 June 2026. This work was supported in part by the National Natural Science Foundation of China under Grant 62002161, Grant 62272225, and Grant 62572363, in part by the Open Project Foundation of State Key Lab. for Novel Software Technology, Nanjing University under Grant KFKT2024B35, in part by the Research and Application of Key Technologies in Foundational Software for Intelligent Construction of Digital Scenarios under Grant 2025CXPT098, and in part by the Collaborative Innovation Center of Novel Software Technology and Industrialization. Recommended for acceptance by G. Fraser. *(Corresponding author: Weiqin Zou.)*

Guowei Zhang, Weiqin Zou, and Jingxuan Zhang are with Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China (e-mail: weiqin@nuaa.edu.cn).

Xiaowei Zhang is with the State Key Laboratory of High Performance Server and Storage Technology, Jinan Inspur Data Technology Co., Ltd., Jinan 250101, China.

Jifeng Xuan is with Wuhan University, Wuhan 430072, China.

This article has supplementary downloadable material available at https://doi.org/10.1109/TSE.2026.3692192, provided by the authors.

Digital Object Identifier 10.1109/TSE.2026.3692192

[1]https://LeetCode.com/

[2]https://codeforces.com/

[3]https://www.NowCoder.com/

[4]Example of solution framework: https://LeetCode.com/problems/binary-tree-inorder-traversal/description/

also face challenges in the context of template extraction. Specifically, extracting high-quality templates requires long-scope reasoning across a number of programming solutions. As the number of input solutions increases, the extracted templates become more representative of the underlying problem-solving patterns; however, this increased volume imposes a heavy cognitive load on LLMs. In an end-to-end paradigm, the model must simultaneously handle global coarse-grained structural organization and fine-grained implementation details. This generation paradigm often leads to a loss of controllability and reduced stability. Furthermore, the reliance of LLMs on pre-trained statistical associations and implicit pattern learning may also prevent them from generating templates highly adaptable to specific programming tasks, especially complex ones.

To address these limitations, we propose AdaptGen, a problem-adaptive code template generation method that shifts the focus from probabilistic prediction of LLMs to evolutionary optimization. Specifically, AdaptGen utilizes genetic programming to iteratively refine candidate templates through mechanisms such as natural selection, crossover, and mutation. By incorporating a goal-oriented fitness function and a linear hashing sequence encoding method, AdaptGen precisely evaluates and optimizes templates against concrete quality criteria. This design enables fine-grained control over the generation process, allowing mutation rates and fitness weights to be tuned so that the resulting optimal solutions strictly align with specific task requirements. These solutions are then processed through a flexible core-code-hiding mechanism to produce the final templates that are expected to enable programmers to focus on their desired skill-specific practice while minimizing boilerplate. These templates could also serve as valuable code assets for broader applications, such as providing structural code sketches that shift the workload of LLMs from long-scope code generation to short-scope code infilling [5], [6], and advancing research within the neuro-symbolic paradigm by using the templates' logic skeletons to constrain and guide neural models [7], [8], [9], [10].

We evaluated AdaptGen using datasets from LeetCode and NowCoder, containing 841 and 156 problems, and over 46,000 and 6,400 solutions. Results showed that AdaptGen produced templates for 77.7% of solution categories on LeetCode and 83.8% on NowCoder. Around 80% of templates performing well in manual evaluations. In addition, AdaptGen achieves the best overall performance among evaluated LLMs, delivering superior template quality and consistency across diverse problem categories while maintaining an optimal balance between generation quality and response efficiency. We further conducted a user study to assess AdaptGen's practical utility. AdaptGen's generated templates are rated by participants with high logic accuracy and syntax correctness, and are considered useful in facilitating their programming training in real-world online programming platforms. The main contributions of this paper are as follows:

1) We introduce AdaptGen, a solution template generation technology that helps programmers focus on problem thinking and core coding by automatically generating flexible boilerplate solution templates to improve learning efficiency.
2) We present the use of genetic algorithms for generating solution templates, including efficient encoding methods, diversity-maintaining strategies, and a specialized fitness function.
3) We construct two Java language datasets from LeetCode and NowCoder to validate the effectiveness of AdaptGen. The experimental code and datasets are publicly available on the AdaptGen GitHub home page[5].
4) We conduct a user study that confirms the practical value of AdaptGen, highlighting its potential to facilitate programming learning of programmers with different expertise.

II. Methodology

In this paper, we propose AdaptGen, a problem-adaptive solution template generation method. AdaptGen automatically identifies template code from a set of solutions and constructs problem-specific templates. Inspired by genetic programming, we treat template generation as an optimization problem. The fitness function evaluates candidate templates, guiding the evolution process to find the best template. Fig. 1 illustrates the basic framework of AdaptGen, consisting of three stages: solution encoding, templated solution construction, and template finalization. The solution encoding stage lays the foundation for genetic algorithms by encoding solutions into hash sequences that preserve both lexical and abstract semantics. In the templated solution construction stage, initial templates are generated by randomly selecting encoded code statements, forming a population that evolves through selection, crossover, and mutation to improve fitness. The evolution is guided by a fitness function, designed to incorporate multiple constraints that reflect common problem-solving logic and coding practices. In the finalization stage, code statements within the templated solutions undergo semantic abstraction and flexible core code-hiding, producing a finalized template for programming practice.

A. Hash Sequence Encoding of Solutions

In genetic programming, encoding methods are critical as they affect both the algorithm's efficiency and the quality of results. While existing encoding solutions for code [11], [12], [13], [14], such as those based on ASTs, code element occurrence matrix, and feature vectors, are available, they are not optimally suited for our task: We focus on the basic framework for solution templates rather than detailed code information. The code implementation details of each code line captured by AST would not help much in template construction but adversely bring memory overhead and inefficient computation problems [15]; The code element occurrence matrix way, which represents code as a matrix by quantifying different types of code elements, such as statements, operators, and operands [16], fails to preserve the code's structural information; the

[5]https://github.com/excuse2020/AdaptGen/

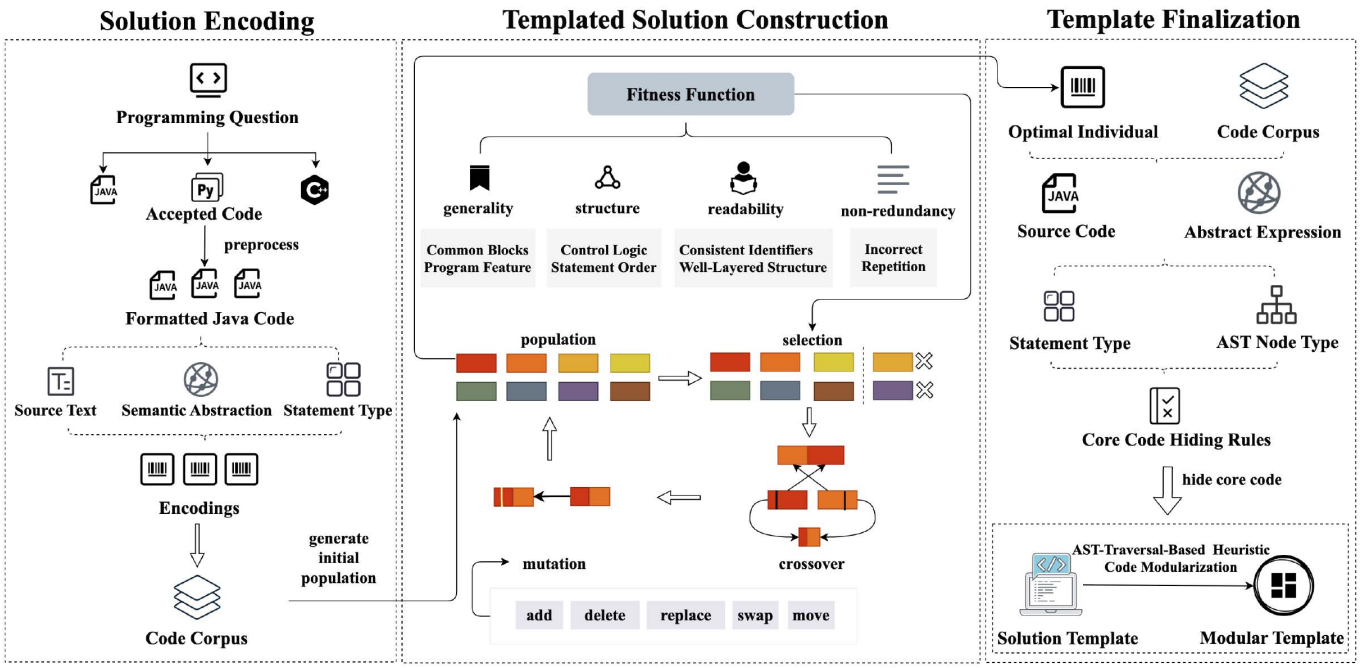


Fig. 1. The overview of AdaptGen framework.

feature vector way that extracts features like control flow, data flow, and AST to represent code as a feature vector, requires extensive computational resources [17], [18], [19].

Hence, we propose a tailored encoding scheme for our solution template generation task, namely *hash sequence encoding*, which preserves essential code structure while supporting efficient genetic evolution. Before encoding, we preprocess the raw code by removing comments, blank lines, and standardizing the code style which involves aligning indentation, formatting code blocks, and ensuring consistent expression formatting. We focus on solutions in Java, using the Java JDT library to convert code into AST, and using AST to complete the formatting of all Java code.

After preprocessing, we apply the hash sequence encoding method, which hashes each line of code statement twice: once for its lexical content and once for its abstract expression. A solution consisting of N lines of code statements is represented by two hash sequences, each containing N hash values. During the encoding phase, we additionally extracted code statement types that would be leveraged in later core code concealment in Section II-C. Several kinds of hashing mappings are constructed for efficient storage and solution-information retrieval. Details are as follows.

(1) **Hashing Lexical Content of a Solution.** A solution is represented as a sequence of hash values, where each value is derived by summing the ASCII values of all characters in a line of code statement. To mitigate hash collisions where different code statements may produce the same hash value, we check whether a newly computed hash has already been assigned to a different statement. If so, then a hash collision happens. In this case, we continuously increment the hash value by one until a unique value is found. This approach simplifies code comparison and management, enhancing the efficiency of fitness calculations in genetic algorithms.

(2) **Semantic Abstraction of Source Code.** We further encode the abstract semantics of the code to identify functionally equivalent code fragments with different textual forms. For instance, both `int res = 5` and `int ans = 5` are abstractly represented as `int SIMPLE_NAME = Num`. This is achieved by analyzing the types of AST nodes and employing corresponding JDT ASTNode APIs[6] to transform similar nodes into unified abstract identifiers.

(3) **Hashing Abstract Expressions.** After obtaining the abstract expressions, we hash them by summing the ASCII values of all characters in each expression, similar to the lexical content hashing. This approach focuses on abstract semantics, enabling efficient comparison of semantically similar code fragments.

(4) **Code Statement Type Extraction.** For each line of code statement, we first obtain its AST node type, e.g., `FOR_STATEMENT` for `for (...)` code statements. Then we categorize these AST node types into different statement types; for instance, AST node types like for_statement, enhanced_for_statement and while_statement are categorized as loops. Other types besides loops include loops, conditionals, declarations, assignments, method calls, returns, and expressions. More implementation details can be found in the GitHub source code of our shared AdaptGen.

(5) **Hash Mapping Storage and Management.** We use various mappings for efficient storage and retrieval of solution information: $codeToHash$ and $hashToCode$ map lexical

[6]https://help.eclipse.org/latest/index.jsp?/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/package-summary.html/

hash values to code, $codeToExp$ and $expToHash$ manage abstract expressions and their hash values, $hashToType$ tracks the statement types associated with each abstract hash value.

By comparing lexical and abstract expression hash values, we efficiently assess whether code statement share the same functionality, ensuring structural similarity while abstracting from specific details. Besides, we build a corpus for each category of each programming problem, preprocess and encode all solutions under each category, and then add each line of code to the corpus.

B. Templated Solution Construction via Genetic Evolution

Templated solution generation is the core phase of AdaptGen, aiming to produce a solution that reflects common practices across individual solutions. We propose using a genetic algorithm to construct these templated solutions. The key challenges include generating the initial population, selecting an appropriate fitness function, and applying genetic operations. The following section details these aspects of the genetic algorithm.

(1) **Generation of the Initial Population.** In genetic algorithms, the initial population consisting of a certain number of individuals (templated solution candidates in this study), kick-starts evolution through crossover and mutation. Ensuring its individual diversity is essential to avoid premature convergence and improve global search. To this end, we construct each individual by randomly selecting N code statements from the statement corpus, where N is also randomly chosen from the range defined by the minimum and maximum line counts of existing solutions. This randomness ensures diversity, while the length constraint prevents excessively short or long individuals that are unlikely to be templated solution candidates. Constructed individuals are then encoded using the hash encoding strategy (Section II-A) and represented by two sequences: a lexical hash sequence and an abstract expression hash sequence.

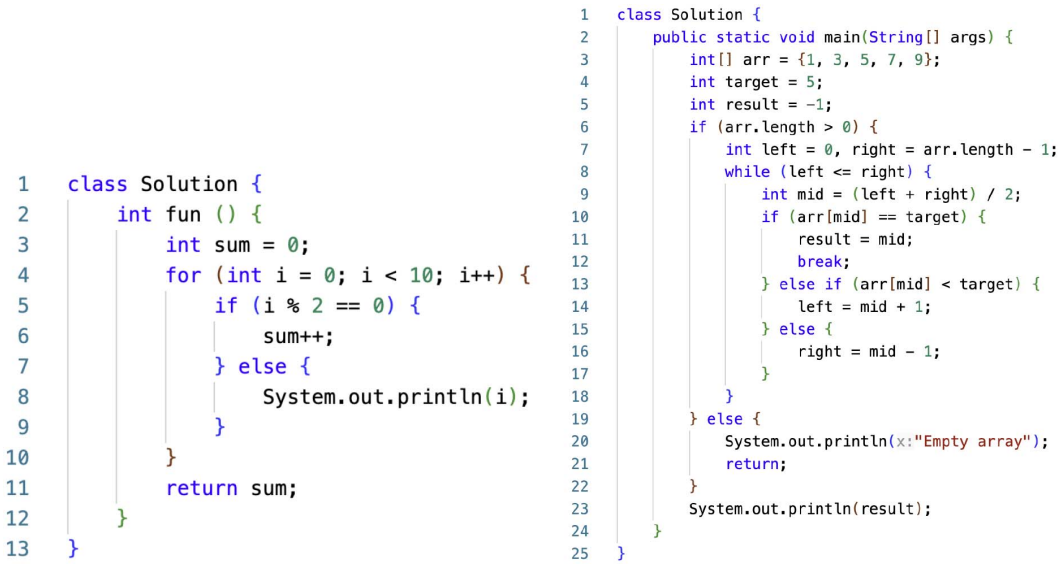
(2) **Evolutionary Strategies.** Evolution strategies are optimization algorithms inspired by natural evolution, designed to solve complex problems. The key challenge lies in how to design concrete *selection*, *crossover*, and *mutation* operations to maintain population diversity, avoid premature convergence, and continuously improve individual quality throughout the evolution process. Towards this, we propose the following strategies. For *selection*, we employ a non-redundancy strategy. That is, we begin by combining the parent and offspring populations into one candidate pool. Then, we filter out duplicates based on their code hash sequences. From the remaining candidates, we select the top L individuals with the highest fitness scores (L is the population size). With this strategy, we can ensure population uniqueness while preserving high-quality solutions. For *crossover*, we employ a stratified single-point strategy. Specifically, we first divide the parent population into high- and low-fitness halves based on their fitness scores. Then we randomly select one parent from each, and generate offspring by swapping their chromosome segments at a randomly chosen crossover point. This strategy promotes genetic exchange across different fitness levels. As for *mutation*, we apply a multi-operator strategy that includes five operators: addition (inserting a new code line at a random position), replacement (substituting a position with a new code line), swapping (exchanging two position's code), moving (relocating a position's code), and deletion (removing a position's code). These operators introduce new variations and expand the search space. The above strategies together drive the evolution process.

(3) **Design of the Fitness Function.** Designing the fitness function is crucial in genetic algorithms, as it evaluates an individual's performance and guides selection. A well-crafted fitness function ensures the evolutionary process produces optimal solution templates.

The design of our fitness function is guided by two general requirements: a templated solution should align with most solutions in code structure and follow common problem-solving logic or coding practices revealed in programmers' submitted solutions. Based on an extensive review of existing literature and analysis of programming data, we developed a fitness function that integrates seven concrete metrics to evaluate the goodness of each individual (i.e., templated solution candidate) during the evolutionary process.

(3.1) Including Common Code Blocks. The code blocks that frequently appear in AC code (i.e., Accepted code, in this study, we use `solution` and `AC code` interchangeably) for a programming problem are likely to reflect the essential part of a solution and, hence, are very likely to be part of a template. To what extent an individual contains those common blocks is one aspect the fitness function takes into account, about which the ratio of common blocks an individual contains is calculated. To calculate this ratio, we need to mine the common blocks among all AC code first, then it would be easy to compute the ratio value. We mainly adopted the algorithm mentioned in CP-Miner [20] to obtain the frequent or common code blocks. Frequent code blocks refer to code fragments or patterns that frequently occur in a software codebase. These blocks often repeat due to common coding habits, design patterns, or specific functional implementations. The core idea of the algorithm is that if a sequence is frequent, then all its subsequences are also frequent. By concatenating each frequent item with the shorter frequent subsequences obtained in previous iterations, longer frequent subsequences are recursively generated. We implemented this algorithm to extract all frequent code blocks from all AC code. Then for each generated individual, we calculate the proportion of frequent code blocks it contains.

(3.2) Preserving General Program Features. A templated solution generally reveals the common coding practice of programmers on online programming platforms. Hence, we measure the matching degree of program features between each generated individual with all AC code in the evolution process. Specifically, inspired by Sudhamani et al. [16], we leverage the program feature matrix to help measure the degree of a constructed individual in preserving the general program features of AC code. The program feature matrix is defined as a



```java
class Solution {
    int fun () {
        int sum = 0;
        for (int i = 0; i < 10; i++) {
            if (i % 2 == 0) {
                sum++;
            } else {
                System.out.println(i);
            }
        }
        return sum;
    }
}
```

```java
class Solution {
    public static void main(String[] args) {
        int[] arr = {1, 3, 5, 7, 9};
        int target = 5;
        int result = -1;
        if (arr.length > 0) {
            int left = 0, right = arr.length - 1;
            while (left <= right) {
                int mid = (left + right) / 2;
                if (arr[mid] == target) {
                    result = mid;
                    break;
                } else if (arr[mid] < target) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }
        } else {
            System.out.println(x:"Empty array");
            return;
        }
        System.out.println(result);
    }
}
```

Fig. 2. Example code for calculating program feature similarity.

one-dimensional matrix $\mathbf{M} \in \mathbb{N}^{1\times 13}$, where each element represents the occurrence count of a specific type of code statement; and 13 statement types in total are considered, with the order of these statement types being: loop statements, `else if` statements, `else` statements, `if` statements, `return` statements, `switch` statements, class declarations, assignment statements, variable declarations, method declarations, method call statements, expression statements, and other statements. Take the two code segments shown in Fig. 2 as an example, their corresponding program feature matrices would be:

$$\mathbf{m}^{(1)} = [1,0,1,1,1,0,1,0,1,1,1,1,0],$$
$$\mathbf{m}^{(2)} = [1,1,2,2,1,0,1,3,5,1,2,0,1].$$

By calculating the average program feature similarity between a templated solution and all AC code, we could assess whether the constructed individual effectively captures the general program features of the AC code of all solutions.

The similarity is calculated using the following formula:

$$pf(m1,m2) = \frac{\sum_{i=0}^{n-1}\left(1 - \frac{|m1[i]-m2[i]|}{m1[i]}\right)}{n} \tag{1}$$

The matrices $m1$ and $m2$ represent the program feature matrices corresponding to the code. The program feature similarity pf measures how similar $m2$ is to $m1$, with $m1$ typically being the feature matrix of the AC code and $m2$ the feature matrix of the generated individual. This similarity is used to assess how well an individual captures the program features of the AC code. The length of the feature matrix is denoted by n, and we assume that the indices of the feature matrix start at 0. If $m1[i]$ equals 0, the similarity at that position is taken as the inverse of $m2[i]$ (i.e., if $m2[i]$ is 0, it is considered 1; otherwise, it is considered 0).

(3.3) Capturing Common Control Logic. While the first two metrics ensure the inclusion of common template code, it does not guarantee that the code is appropriately placed, and hence revealing the correct control logic of most solutions. This metric aims to correctly structure the algorithm by grouping control statements (loops, conditionals) into an accurate algorithmic framework and assigning various types of code statements to the proper blocks (loop blocks, conditional blocks). Following the approach proposed by Sudhamani et al. [16], which involves calculating control statement feature similarity, we fill the control statement feature matrix with the types of control statements and the program features of the blocks within these statements. By calculating the similarity between the control statement feature matrices of two code segments, we can gauge their structural similarity. The average structural similarity between a constructed individual and all AC codes assesses whether the individual effectively captures the basic structure of the algorithm. Tables I and II illustrate the control statement feature matrices for the two code segments in Fig. 2, where L represents loop statements, C represents conditional statements, and the $ProgramFeature$ column denotes the program feature matrix of the current loop or conditional block (a detailed description of the construction process can be found in our supplemental material).

TABLE I
CONTROL STATEMENT FEATURE MATRIX CORRESPONDING TO CODE 1 IN FIG. 2

No	Type	Program Feature												
1	L	0	0	1	1	0	0	0	0	0	0	0	0	0
2	C	0	0	0	0	0	0	0	0	0	0	0	1	0
3	C	0	0	0	0	0	0	0	0	0	0	1	0	0

Assuming that the two matrices are of sizes n and m, respectively, if $n = m$, the similarity $csf(A,B)$ is calculated as follows:

$$csf(A,B) = \frac{1}{n}\sum_{i=0}^{n-1}\begin{cases} pf(A[i],B[i]), & A[i].type = B[i].type \\ 0, & otherwise \end{cases} \tag{2}$$

TABLE II
CONTROL STATEMENT FEATURE MATRIX CORRESPONDING TO CODE 2 IN FIG. 2

No	Type	Program Feature												
1	C	1	0	0	0	0	0	0	0	1	0	0	0	0
2	L	0	1	1	1	0	0	0	0	1	0	0	0	0
3	C	0	0	0	0	0	0	0	1	0	0	0	0	1
4	C	0	0	0	0	0	0	0	1	0	0	0	0	0
5	C	0	0	0	0	0	0	0	1	0	0	0	0	0
6	C	0	0	0	0	1	0	0	0	0	0	1	0	0

In this formula, csf represents the control statement feature similarity of code B relative to code A, where A and B are the control statement feature matrices corresponding to the code. Typically, A is the control statement feature matrix of the AC code, and B is the control statement feature matrix of a generated individual. The value n indicates the number of rows in the feature matrix, with $A[i]$ and $B[i]$ representing the i-th row of matrices A and B, respectively. The term $pf(A[i], B[i])$ is the program feature similarity between the one-dimensional matrices $A[i]$ and $B[i]$. The expression $A[i].type$ refers to the type of control statement corresponding to that row in the matrix (L/C).

If $n \neq m$, the smaller matrix is matched with all submatrixs of the larger one, transforming the comparison into a similarity calculation between matrices of equal size. The maximum similarity value $csf_{\max}$ is selected, and the final similarity is adjusted as follows:

$$csf(A, B) = \frac{2 \times \min(n, m)}{n + m} \times csf_{\max} \tag{3}$$

(3.4) Maintaining Code Statement Order. With the previous three metrics as guidance, an individual is expected to construct the algorithm framework and retrieve necessary code blocks. However, this is still not enough, as the exact sequence of these statements within the blocks remains indeterminate, and similar types of template code may be misplaced in incorrect blocks. For example, suppose both a loop block and a conditional block contain an assignment statement and a method call (till now, only the types of statements are considered, the order of these statements are not further examined); if we do not add further order constraints to these statements, the assignment statement supposed to belong to the loop block may be misplaced into the conditional block, and vice versa.

To resolve this issue, we introduce the concept of edit distance, which measures the minimum number of edit operations, such as insertion, deletion, or substitution, required to transform one string into another. A smaller edit distance indicates a smaller difference between two segments of code. Each line of code is represented by the hash value of an abstract expression, and dynamic programming is employed to efficiently compute the edit distance between two code segments. To obtain a scale-invariant similarity score, we compute the normalized edit distance between two code segments, denoted as A and B, using the following formula:

$$\mathrm{ED}(A, B) = 1 - \frac{\mathrm{edits}(A \to B)}{\max(A.\mathrm{size}(),\ B.\mathrm{size}())} \tag{4}$$

```
1  class Solution {
2      public int peakIndex (int [] arr) {
3          int left = 0, right = arr.length - 1;
4          while (left < right) {
5              int mid = l + (r - l) / 2;
6                  if (arr[mid] > arr[mid + 1]) {
7                      right = mid ;
8                  } else {
9                      l = mid + 1;
10                 }
11             }
12         return l;
13     }
14 }
```

Fig. 3. Example code for inconsistent identifiers.

Here, $\mathrm{edits}(A \to B)$ denotes the number of edit operations required to transform A into B, and $A.\mathrm{size}()$ and $B.\mathrm{size}()$ represent the number of code lines in the two segments. By calculating the average ED between an individual and all AC code, we can assess whether the fundamental statements are correctly ordered within the appropriate code blocks. A better template with the correct sequence and placement of code will exhibit a smaller average edit distance.

(3.5) Keeping Consistent Identifiers. Since the code of an individual is constructed through evolution operators like mutation and crossover over the corpus of all AC code, it is possible that identifiers like variables or method names supposed to be of the same names are instead, used with other names in relevant code statements. For example, in Fig. 3, the variables names $left$ and $right$, defined in line 3, are mistakenly used as l and r in lines 5, 9, and 12. In this study, we resort to the code coverage metric to help mitigate this issue, in that a template that maintains consistent naming usage will show higher code coverage by matching more lines with the AC code. Specifically, for each individual, we compute the proportion of matching lines between the individual and each piece of AC code, with the code lines represented by their lexical hash values. Among all computed code coverage values, the maximum value is chosen to indicate the degree to which an individual keeps a consistent naming and use of identifiers.

(3.6) Avoiding Incorrect Repetition. Based on our observations, we find that some generated individuals contain unnecessary redundancy, where the same code statement appears multiple times despite not being repeated in the original AC code. To address this, we propose using both edit distance (as described in section (4)) and code repetition rate to measure redundancy. The edit distance serves as a constraint, as individuals with redundant code tend to have a larger edit distance from the original AC code, indicating more edit operations. The code repetition rate evaluates the extent of repeated code by comparing the frequency of identical statements between the template and AC code, favoring smaller discrepancies. A higher frequency of repeated statements lowers the code repetition rate. By calculating the average repetition rate across all AC code, we can quantify redundant code in the generated solution. These two metrics work together to minimize incorrect repetition in the constructed solutions.

(3.7) Holding A Well-Layered Code Structure. After we obtain an individual that contains all necessary and ordered

templated code statements (mainly from the logic and functional perspective), we also need to ensure that these statements are organized in a well-layered code structure that has correct indentation and bracket usage. In this way, a well-formatted and easy-to-read individual would be produced and could be provided to programmers to directly work on (after the template finalizaiton step in Sect. II-C). Note that for brackets, we mainly consider the {} as: (1) They are closely tied to indentation, which enables us to evaluate the indentation structure based on their usage, and (2) Other brackets like [] and () generally appear within a single line of code, their correct usage would generally be guaranteed by our AdaptGen that working on individual lines granularity. In this study, we mainly employ a stack-based algorithm to evaluate whether an individual holds a well-layered code structure, using {} nesting to track indentation levels and returning false upon any mismatch.

The above seven aspects, on the whole, determine how good an individual is. After we obtain the associated seven evaluation values, we use different weights to sum them and take the final score as the fitness of an individual during the evolution process.

C. Template Finalization via Semantic Abstraction and Core Code Concealment

(1) Core Code Concealment. After obtaining the optimal individual, i.e., the templated solution, evolved by the genetic algorithm, further decoding is required to convert the encoded hash sequence into the final solution template. Specifically, the hash sequence corresponding to the templated solution is converted into source code and abstract expressions based on the hash-code mappings ($hashToCode$, $hashToExp$) stored during the encoding phase (Section II-A). Next, each line of code statements is categorized to identify its core and boilerplate parts based on its statement type (also using constructed $hashToType$ mappings in Section II-A). Then, the boilerplate part is retained, while the identified core part is transformed into template elements through semantic abstraction, resulting in the final solution template. Based on statistical analysis of solutions, we identified the following statement types: loops, conditionals, declarations, assignments, method calls, returns, and expressions. The detailed semantic abstraction and core code identification for each type are outlined below.

(1.1) Loop Statements. Loop control statements usually contain the logical reasoning required to solve the problem. AdaptGen identifies these as core statements. During the process of template finalization, instead of using the original loop statements, we use its corresponding abstract expression, with the loop control logic being masked with `...`. For instance, the `for (int i = 0; i <nums.length; i++)` is replaced by `for (...)`.

(1.2) Conditional Statements. Conditional judgments often contain the logical reasoning required to solve the problem. AdaptGen also identifies them as core statements. Like loop statements, during template finalization, the conditional statements are replaced with their abstract expressions, with the predicate judgment masked with `....` For example, `if (nums[mid] >target)` is replaced with `if (...)`.

(1.3) Variable Declaration Statements. For variable declaration statements (like `int sum;` or `int sum = 0;` or `int sum = a + b;`), AdaptGen would take the variable part (like `int sum`) as boilerplate code and keep it in the template and replace the possible initial value (like `0` or `a + b`) with an abstract expression determined by the AST node type. For instance, `int sum = 0;` would be changed to `int sum = Num;`, and `int sum = a + b;` would be changed to `int sum = Expression;`.

(1.4) Method or Class Declaration Statements. Method or Class declarations typically do not involve relevant logical reasoning or computations, so their original code (e.g., `class Solution` and `int search(int[] nums, int target)`) is used directly during the template finalization.

(1.5) Assignment Statements. Similar to variable declarations, during template finalization, the left-hand side of an assignment statement is preserved, while the right-hand side is replaced by an abstract expression based on its corresponding AST node type.

(1.6) Method Invocation Statements. For these statements, we will preserve the method name, and replace parameters with abstract expressions based on AST node types. For example, in `dfs(2, i + 1)`, `dfs` is retained, while `2` is replaced with `Num` and `i + 1` with `Expression` (i.e, `dfs(Num, Expression)`).

(1.7) Return Statements. During our template finalization process, the keyword `return` of a return statement is retained, and the return value is replaced with a corresponding abstract expression based on its AST node type. For example, the statement `return "abc";` would be transformed to `return Str;` in our final solution template.

(1.8) Expression Statements. These include prefix, infix, and postfix expressions, all of which are uniformly replaced with the abstract expression `Expression` during the finalization process. For example, `a + b`, `cnt++`, `--cnt`.

Note that AdaptGen is not limited to handling simple statements in isolation. When encountering advanced or more complex Java language features, such as lambda expressions, long method invocation chains, or nested classes, AdaptGen does not simply treat them as generic Expression nodes. Instead, it applies the above core-code hiding rules in a recursive manner by decomposing complex constructs into smaller AST nodes and abstracting each node according to the same rule set (a more detailed description of how these advanced features are handled can be found in our supplemental material).

In the template finalization phase, these rules are applied to each line of code based on its type, transforming the genetic algorithm's templated solution to a programming problem into a final solution template that provides boilerplate code and hides core code to better support programmers' programming practices.

(2) Flexible modular templates. To better accommodate the needs of programmers of different expertise, we introduce a *modular template construction* mechanism that organizes each finalized template into five semantic modules: data structure definition, variable declaration, control structure that generally captures the core problem-solving logic, input/output

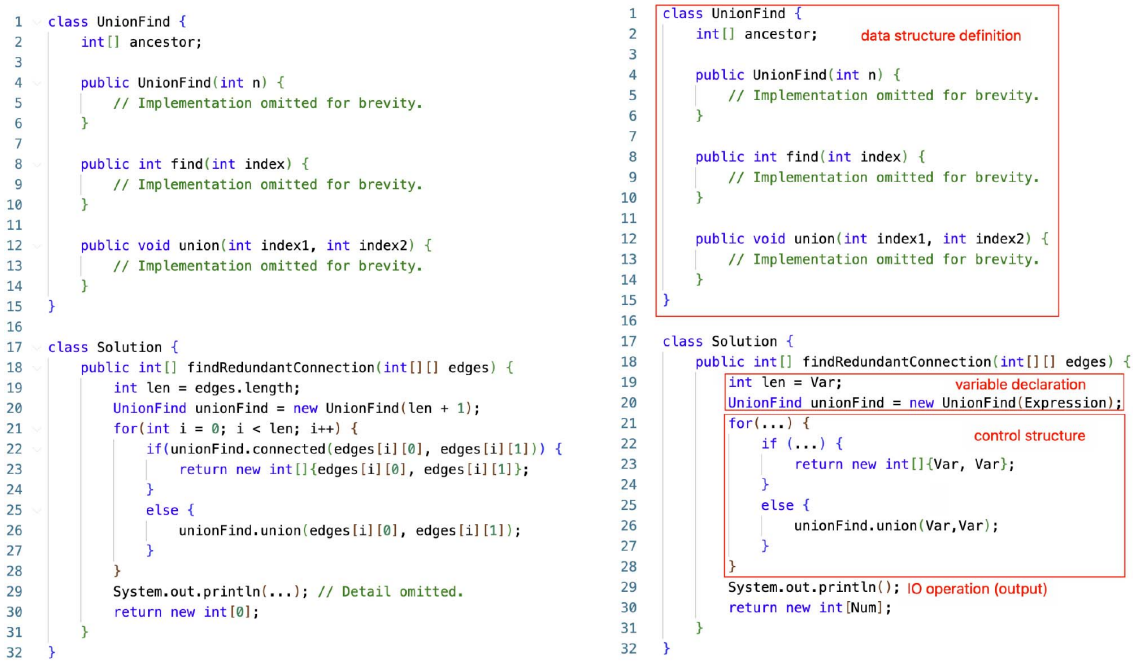


Fig. 4. Comparison of templates before (left) and after (right) finalization and modularization.

operations, and other statements not belonging to the previous four types. The modularization process is implemented via static analysis of AST, constructed using the Eclipse JDT library. Code elements are categorized into module types through AST traversal and rule-based heuristics. Specifically, class declarations not part of the main solution class (`class Solution`) are identified as data structure definitions. Variable declarations are identified through analyzing the node types of AST. Control structures, mainly include control statements like `if`, `for`, `while`, `switch` statements and their embedded body. To detect input/output operations, a predefined set of common Java I/O methods(e.g., `print`, `println`, `nextInt`, `readLine`) flags relevant method invocations. All remaining statements, such as class declarations, method declarations, and return statements, are classified as other statements. Each module's line range is recorded to support selective rendering during template generation. Fig. 4 provides a example showcasing the differences introduced by finalization and modularization.

This modularization lays the groundwork for flexible *template customization*, allowing different combinations of modules to be selectively included or omitted according to programmers' specific learning goals. For instance, to help programmers focus on data structure design, AdaptGen can generate customized templates that hide the data structure definition and variable declaration modules, leaving learners to complete them independently. By enabling module-level customization, this approach would provide learners with the flexibility to focus on specific areas, helping them develop their skills progressively without becoming over reliant on predefined templates.

III. Experimental Design

A. Experimental Dataset

We evaluated the practicality of AdaptGen using real-world programming data from two widely recognized online platforms: LeetCode and NowCoder. Both platforms offer programming exercises in multiple languages, and our study focused on generating Java solution templates. Data collection was carried out using a Python web crawler. The data from NowCoder was collected in July 2024, while the data from LeetCode was collected in August 2024. From NowCoder, we successfully retrieved 368 problems along with their accepted (AC) code. For the LeetCode platform, which contains a total of 3,279 problems, we obtained 1,081 problems and their corresponding AC code due to access restrictions or exceptions during the crawling process. The LeetCode dataset covers 58 different algorithm types, including depth-first search, greedy algorithms, and hash tables, while the NowCoder dataset includes 34 algorithm types, such as binary search, dynamic programming, trees, and stacks. This broad variety of algorithm types provides a strong foundation for evaluating AdaptGen's effectiveness.

After collecting the AC code, we preprocessed the data by removing comments, eliminating empty lines, and standardizing the code styles. We also filtered out programming problems with fewer than five pieces of AC code to ensure sufficient data for effective template generation. Considering that different solutions to the same programming problem may employ varying data structures and algorithms, it is crucial to classify these solutions before generating templates. Without distinguishing between different types, mixing them during template generation could result in poor outcomes due to the

TABLE III
BASIC STATISTICS OF LEETCODE (L) AND NOWCODER (N) DATASETS AFTER EACH PROCESSING STEP

Datasets (L/N)	#Questions	#AC Code	#Solution Categories
Original	1,081/368	74,093/30,640	-
Formatted Java Code	1,028/343	58,643/9,312	-
Filter Question	943/202	58,434/9,030	-
Classified	943/202	58,434/9,030	8,872/1,781
Filter Category	841/156	46,319/6,437	2,730/489

significant differences in code structures caused by different algorithms. To address this, we classify the AC code collection based on structural similarity, grouping together code that share similar structures. We use the structural similarity metric described in Section II-B (formulas 1 - 3) to perform this grouping. Code with a similarity score above s are grouped together, ensuring that any two pieces of code within the same category have a similarity greater than s. After preliminary experimenting with different values of s, we found that setting $s = 0.6$ provides a reasonable balance, ensuring both useful classification and template generalization. Following classification, we filter out categories containing fewer than five pieces of AC code to ensure sufficient data for supporting algorithm evolution and evaluation.

Table III details the datasets used. The original LeetCode dataset included 1,081 problems and 74,093 AC code. After filtering and classification, we retained 841 problems (Easy: 264, Medium: 448, Hard: 129), whose solutions are grouped into 2,730 categories, comprising 46,319 Java programs. The original NowCoder dataset included 368 problems and 30,640 AC code. After processing, we retained 156 problems (Easy: 55, Medium: 88, Hard: 13), 489 solution categories, and 6,437 Java programs.

B. Research Questions

To evaluate the effectiveness of AdaptGen, we seek to answer the following three research questions:

- **RQ1:** Can AdaptGen effectively generate high-fitness solution templates?
- **RQ2:** How well do the solution templates generated by AdaptGen meet the various template requirements?
- **RQ3:** What advantages does AdaptGen offer compared to advanced LLMs?

RQ1 aims to assess whether AdaptGen can efficiently produce individuals with high fitness values. As a genetic algorithm-based method for generating solution templates, one key aspect is whether AdaptGen can converge to an optimal solution within a limited number of iterations. The fitness value of the generated solution at convergence serves as a crucial indicator of AdaptGen's usability. The fitness function proposed in this paper defines the criteria that an ideal solution template should meet, and higher fitness values indicate that the template better satisfies these criteria.

While RQ1 offers a quantitative assessment of AdaptGen's efficiency and the fitness of its generated templates, RQ2 complements this with a qualitative analysis. This involves manually evaluating each solution template based on several key aspects, such as the capture of the basic algorithm structure, the presence of commonly used code, readability, etc. This approach offers a deeper insight into the strengths and limitations of AdaptGen in generating high-quality solution templates.

RQ3 investigates whether AdaptGen outperforms typical LLMs in generating high-quality solution templates. We designed an experiment to evaluate the ability of each LLM to generate solution templates based on multiple solution examples. Specifically, we compared AdaptGen against seven models: DeepSeek-R1, DeepSeek-V3, Qwen-Max, GPT-4o-Mini, Doubao-Seed-1.6-Thinking, Gemini-2.5-Flash and Claude- 3.5-Haiku.

C. AdaptGen Implementation

We developed AdaptGen using JDK 17 and JDT 3.24.0 as the development environment. For the genetic algorithm, we set the termination condition to when the best fitness value in the population remained unchanged for n consecutive generations. The configuration of n, the population size, and the mutation rate are key parameters that may significantly impact the performance of AdaptGen. To identify a suitable parameter set that effectively balances solution quality with evolutionary efficiency, we first conducted preliminary tests on a small-scale dataset by evaluating various combinations. Based on these results, the following parameters were selected for the final experiments: n was set to 500, the population size was configured as 100 individuals per generation, and a relatively high mutation rate of 0.7 was adopted. The weights used in the fitness function were also determined during these preliminary tests. That is, we checked the impact of different weight combinations on the generated templated solutions to assess whether they could consistently satisfy the proposed requirements. The concrete weight assignment was determined by considering both the relative importance of each metric to the overall objectives and the experimental observations (e.g., certain requirements could not be met when their corresponding weights were set too low). Finally, the combination of weights: 0.1, 0.1, 0.25, 0.25, 0.05, 0.2, and 0.05 was selected in our experiments.

IV. EXPERIMENTAL RESULTS

A. RQ1: Can AdaptGen Effectively Generate High-Fitness Solution Templates?

To evaluate whether AdaptGen can generate high-fitness templates, we analyzed its performance in terms of convergence, overall fitness, and qualification rate.

Convergence: We applied AdaptGen to generate solution templates for 2,730(489) solution categories in the LeetCode (NowCoder) dataset. For each category, we recorded the number of iterations required for convergence. The maximum evolution time was limited to 10 minutes. Our experiments showed that in the NowCoder dataset, 98.36% (481/489) of the solution

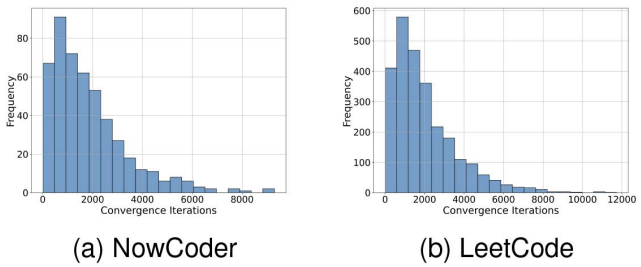


Fig. 5. The distribution of convergence iterations for NowCoder and LeetCode.

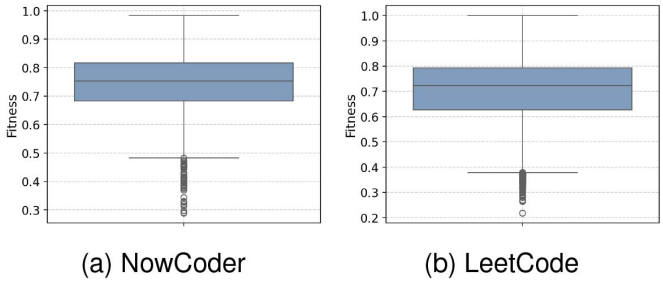


Fig. 6. The distribution of fitness values for solution templates after convergence.

categories converged to a stable solution within 10 minutes. In the LeetCode dataset, the corresponding value was 95.38% (2,604/2,730). Fig. 5 illustrates the distribution of convergence iterations across the NowCoder and LeetCode datasets, revealing a trend similar to a power-law distribution, with the majority of solution categories (64% and 61%) converging within 2,000 iterations. This finding indicates that in most cases, the algorithm can identify an optimal solution with relatively few iterations, a characteristic that largely supports AdaptGen's practical application.

Overall Fitness: Fig. 6 presents box plots of the fitness values after convergence in the NowCoder and LeetCode datasets. From the figure, we can find that the fitness values of generated templates for the NowCoder platform are concentrated between 0.7 and 0.8, with a median close to 0.75 and very few outliers. This means AdaptGen has a good performance in generating solution templates with good fitness. For the LeetCode platform, most fitness values are concentrated between 0.6 and 0.8, with a median of around 0.72. This also indicates an overall satisfactory generation performance of AdaptGen. Meanwhile, some outliers with fitness values below 0.4 show that quality fluctuations may still happen in specific programming solutions.

Qualification Rate: To evaluate the quality of the evolved solution template, we examined whether the fitness value of the optimal solution met the expected standard. Since it is difficult to set a unified fitness value across different programming problems, we calculated the fitness values of all AC code for each category and used their average as the benchmark for evaluating the generated template. The method for calculating the fitness of AC code is consistent with that used for templates. We employed the qualification rate (QR) to determine whether the genetic algorithm produced satisfactory solutions. The specific formula is shown in Equation 5.

$$\text{QR} = \frac{\sum_{i=1}^{n} \mathbf{1}\{f(T_i) > avg(f(C_i))\}}{n} \tag{5}$$

Here, n is the number of evolved templates; $f(T)$ is the fitness score of the template; $f(C)$ is the fitness score of the AC code; and $avg(f(C))$ is the average fitness score of all AC code.

For each solution category in the dataset, we calculated the average fitness value of all AC code within that category and compared it with the fitness value of the optimal solution generated by the genetic algorithm. As a result, 77.7% (2,120/2,730) of the templates in the LeetCode dataset met the qualification standard, meaning that for 77.7% of the solution categories, the fitness values after convergence exceeded the average fitness values of the AC code. In the NowCoder dataset, a higher qualification rate was observed, with 83.8% (410/489) of the templates exceeding the average fitness value of all AC code. This indicates that the genetic algorithm is generally able to evolve solutions that meet the expected requirements. (i.e., the seven criteria considered in the fitness function discussed in Section II-B).

RQ1 Results Summary: AdaptGen demonstrates good performance in generating high-fitness solution templates for most solution categories of LeetCode and NowCoder.

B. RQ2: How Well Do the Solution Templates Generated by AdaptGen Meet the Various Template Requirements?

To assess the practical applicability of templates generated by AdaptGen, we manually reviewed the generated templates by AdaptGen, with a focus on checking to what extent individual templates meet various template requirements, including capturing the basic structure of an algorithm, the ability to hide core code of a solution, etc. We established scoring criteria tailored to different template requirements based on our preliminary observations over the online programming data. Using these criteria, we manually checked the generated templates against the original AC code of each solution category to evaluate its quality. All 481 generated templates for the NowCoder dataset are manually checked. For the LeetCode dataset, due to the large number of solution categories and high manual inspection costs, we opted to sample a portion of the data for manual review. To ensure the representativeness and reliability of our findings on LeetCode, by following the guide of [21], we randomly selected 384 categories from the 2,604 successfully generated templates for evaluation, with a 95% confidence level and a 5% margin of error. The specific scoring criteria we referred to during manual evaluation are shown in Table IV. Detailed evaluation methods can be found in the RQ2 section of the AdaptGen Github home page.

Tables V and VI show the experimental results of manual evaluation over the NowCoder and LeetCode datasets. From the tables, we can find that the generated templates are generally effective in capturing common code statements (A: 87.7% / 82.8%, B: 10.2% / 15.1%), avoiding incorrect repetition (A:

TABLE IV
SCORING CRITERIA FOR TEMPLATE EVALUATION

Criterion	Description	Scoring Criteria
Inclusion of Common Code Statements	Evaluates missing parts of common code in the template.	A: $< 5\%$ B: $< 15\%$ C: $< 30\%$ D: $> 30\%$
Capture of the Algorithm's Basic Structure	Assesses the extent to which the structural code in the template aligns with the algorithm's basic structure.	A: $< 5\%$ B: $< 15\%$ C: $< 30\%$ D: $> 30\%$
Free from Incorrect Repetition	Evaluates the proportion of incorrect redundant code in the template.	A: $< 5\%$ B: $< 15\%$ C: $< 30\%$ D: $> 30\%$
Proper Hiding of Core Code Statements	Determines the proportion of core code in the template, ensuring room for creativity.	A: $< 5\%$ B: $< 15\%$ C: $< 30\%$ D: $> 30\%$
Good Readability with Well-Layered Structure	Checks that the structure is correct (Yes/No) and consistency of variable naming in the template.	A: Yes and $< 5\%$ B: Yes and $< 25\%$ C: Yes and $< 40\%$ D: No or $> 40\%$
Overall Quality	Evaluates the overall quality of the template, considering the required adjustments.	A: $< 5\%$ B: $< 25\%$ C: $< 40\%$ D: $> 40\%$

Note: Percentages represent the proportion of each item (e.g., redundant code, unhidden core code, inconsistent naming, or required adjustments) relative to the total lines of the template.

TABLE V
MANUAL ANALYSIS OF SOLUTION TEMPLATE ON REQUIREMENT SATISFACTION (NOWCODER)

Requirements \ Category	A	B	C	D
Common Code Inclusion	87.7%	10.2%	0.2%	1.6%
Basic Structure Capturing	79.6%	7.2%	7.0%	6.3%
Incorrect Repetition Avoidance	84.1%	13.3%	1.0%	1.6%
Readability	38.5%	39.5%	10.2%	11.5%
Core Code Hiding	96.1%	1.2%	0.8%	1.8%
Overall Quality	50.9%	29.9%	9.8%	9.4%

84.1 % / 95.1%, B: 13.3% / 4.7%), and hiding core code (A: 96.1% / 96.1%, B: 1.2% / 2.9%). These results verify that metrics like the ratio of common code blocks and program feature similarity (which are integrated within our fitness function to evolve templates) are indeed reliable indicators of template quality. In terms of readability, the templates performed slightly less consistently compared to the other aspects, with the proportion of A and B grades being around 78% for both NowCoder and LeetCode. Compared to structural-related problems like incorrect indentation, issues related to identifier naming are more common. This suggests that sometimes the coverage metric (embedded in the fitness function) may not fully ensure identifier naming consistency of templates. Incorporating code refactoring tools in future template optimization processes could help standardize variable naming and enhance overall code readability.

Through manual inspection, we found that templates with poorly organized structures consistently struggled to capture algorithmic frameworks effectively. In contrast, templates with well-structured formats reliably captured the underlying algorithm. This confirmed the importance of the control statement-related metrics integrated into our fitness function. Structure correctness had a significant impact on the overall quality of the generated templates. Poorly structured templates, which often scored zero in the structural validity metric (also embedded in the fitness function), indicated that the genetic algorithm had failed to identify suitable solutions. These poorly structured templates were typically longer, reinforcing our hypothesis. To improve the chances of generating structurally sound templates, we plan to explore more advanced genetic operators to enhance the diversity of individuals in the evolutionary process. This would expand the solution search space and increase the likelihood of generating well-structured templates.

TABLE VI
MANUAL ANALYSIS OF SOLUTION TEMPLATE ON REQUIREMENT SATISFACTION (LEETCODE)

Requirements \ Category	A	B	C	D
Common Code Inclusion	82.8%	15.1%	1.6%	0.5%
Basic Structure Capturing	71.6%	10.9%	8.6%	8.9%
Incorrect Repetition Avoidance	95.1%	4.7%	0.3%	0.0%
Readability	40.1%	38.3%	11.2%	10.4%
Core Code Hiding	96.1%	2.9%	0.3%	0.8%
Overall Quality	44.0%	35.9%	9.9%	10.2%

Regarding overall quality, 50.9% of the templates generated by AdaptGen for NowCoder and 44.0% for LeetCode were of high quality, requiring only minor adjustments ($<5\%$) to be transformed into ideal templates for practical use. With minimal code adjustments (A+B), approximately 80% of the generated templates could be refined into ideal templates, demonstrating their significant potential for real-world application in online programming platforms. Moreover, AdaptGen performed well in generating templates for highly structured algorithms, such as binary search and breadth-first search. These algorithms typically follow a standardized framework with few variations, which makes them easier to template.

RQ2 Results Summary: Our qualitative manual evaluations demonstrate that AdaptGen-generated templates, particularly for highly structured solutions, achieve good requirement satisfaction on two representative online programming platforms, though readability still has room for improvement.

C. RQ3: What Advantages Does AdaptGen Offer Compared to Advanced LLMs?

Given that advanced LLMs can also generate a template directly from a set of solution examples, it is essential to examine whether AdaptGen provides concrete advantages over LLMs in producing accurate and robust templates. Towards this, we designed an experiment to assess each model's generation ability from multiple K solutions, with five K values, i.e., 5, 10, 15, 20, 30, being tested separately. Seven typical/advanced LLMs

were compared, including DeepSeek-R1, DeepSeek-V3, Qwen-Max, GPT-4o-Mini, Doubao-Seed-1.6-Thinking, Gemini-2.5-Flash and Claude-3.5-Haiku, varying K could help capture individual LLMs' sensitivity to example counts and reveal AdaptGen's robustness under different sizes of the solution pool used by the GA.

Specifically, for each solution category with $\geq$ K solutions, we first randomly selected K solutions and then let each model (AdaptGen + seven LLMs) to generate a templated solution for this category; these templated solutions can subsequently be transformed into flexible modular templates through our core-code hiding strategy. The fitness scores of these templated solutions are then compared across AdaptGen and seven LLMs; the time each model took to generate templates is also recorded.

During experiments, for the LLMs, we carefully crafted a prompt through the prompt generation and optimization capabilities provided by the AutoPE platform[7]. The final prompt after optimization is shown in the Fig. 7 and is used to guide the LLM to generate templates. Further, to balance computational cost while maintaining sufficient statistical power [21], we applied stratified sampling: when the number of qualifying categories exceeded 384, we randomly sampled 384 categories; otherwise, all qualifying categories were used. Fig. 8 presents fitness distributions across multiple models on LeetCode and NowCoder datasets with varying numbers of input solutions. The specific counts for each sample size are shown in the x-axis labels (format: x/y/z) of Fig. 8, where x represents the number of categories actually used in experiments, y indicates the number of qualifying categories, and z shows the total available categories in the original dataset. To provide a comprehensive comparison, we analyze the performance of AdaptGen and LLMs from the following aspects.

Template Quality. From the fitness boxplots in Fig. 8, we can observe that AdaptGen achieves a higher overall template quality (as measured by fitness score) across both datasets. Specifically, based on our statistics, on LeetCode, the mean fitness of AdaptGen ranges from 0.751 to 0.777 across different sample sizes, remaining above all LLMs; and the median value ranges from 0.775 to 0.782, also above all LLMs except being slightly surpassed by DeepSeek-R1 at 30 samples (0.779 for AdaptGen vs. 0.780 for DeepSeek-R1). On NowCoder, the mean fitness of AdaptGen grows from 0.776 (5 samples) to 0.807 (30 samples), remaining above all LLMs except for Claude-3.5-Haiku at 20 samples (0.799 vs. 0.801); and the median value grows from 0.781 to 0.815, also higher than all LLMs except that Claude-3.5-Haiku slightly surpasses AdaptGen at 10, 20, and 30 samples (with differences within 0.01), and DeepSeek-R1 exceeds AdaptGen at 30 samples (0.815 vs. 0.822). Notably, at 5 samples, where the available data is sufficient, AdaptGen attains the highest mean and median fitness among all models.

The proportion of high-quality templates (with fitness scores > 0.7) generated by AdaptGen also substantially outperforms other models across varying sample sizes. AdaptGen achieves 77%–83% of high-quality templates on LeetCode and 83%–97% on NowCoder across all sample sizes. For LLMs, the proportions are: DeepSeek-R1 65%–79% (LeetCode), 71%–95% (NowCoder); DeepSeek-V3 61%–77%, 65%–95%; Doubao-Seed-1.6-Thinking 60%–74%, 64%–97%; Claude-3.5-Haiku 49%–66%, 74%–97%; Gemini-2.5-Flash 62%–70%, 72%–78%; GPT-4o-Mini 56%–67%, 56%–78%; Qwen-Max 56%–70%, 52%–81%.

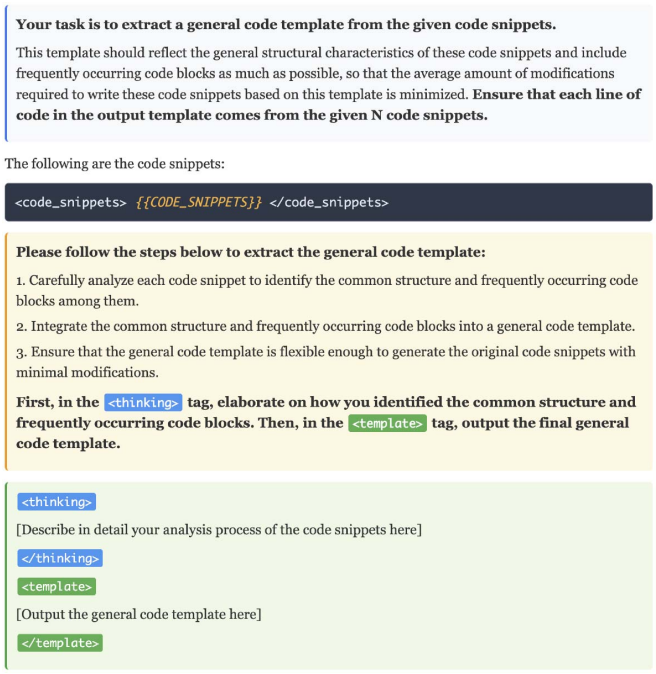


Fig. 7. The designed prompt for LLMs in the template generation task.

Generation Stability. From the fitness-score boxplots in Fig. 8, we can see that, on LeetCode, the interquartile range of AdaptGen remains compact across all sample sizes, with outliers mainly concentrated in the high-score region. On NowCoder, AdaptGen's fitness boxplots become progressively tighter as sample size increases, and low-quality outliers nearly disappear at 30 samples. When jointly considering the fitness distributions of LLMs, we see that AdaptGen consistently produces a tighter distribution with fewer outliers, indicating AdaptGen's greater stability in generating templates.

Sensitivity to Sample Size. From Fig. 8, we can find that, as the number of input solutions increases, AdaptGen's median fitness remains stable on the LeetCode dataset, while showing a steady upward trend on the NowCoder dataset. In contrast, LLMs exhibit gradual performance improvements on both datasets, with DeepSeek-R1 and DeepSeek-V3 exhibiting faster improvements and other LLMs showing slower gains. This indicates that LLMs are more sensitive to sample size than AdaptGen, highlighting AdaptGen's stronger robustness and stability under varying data availability conditions. This makes AdaptGen more attractive in light of the observation that AdaptGen consistently generates higher-fitness templates than LLMs (as shown in the Template Quality analysis part), while LLMs needs to bear substantially higher reasoning overhead

[7]https://console.volcengine.com/ark/region:ark+cn-beijing/autope/startup/

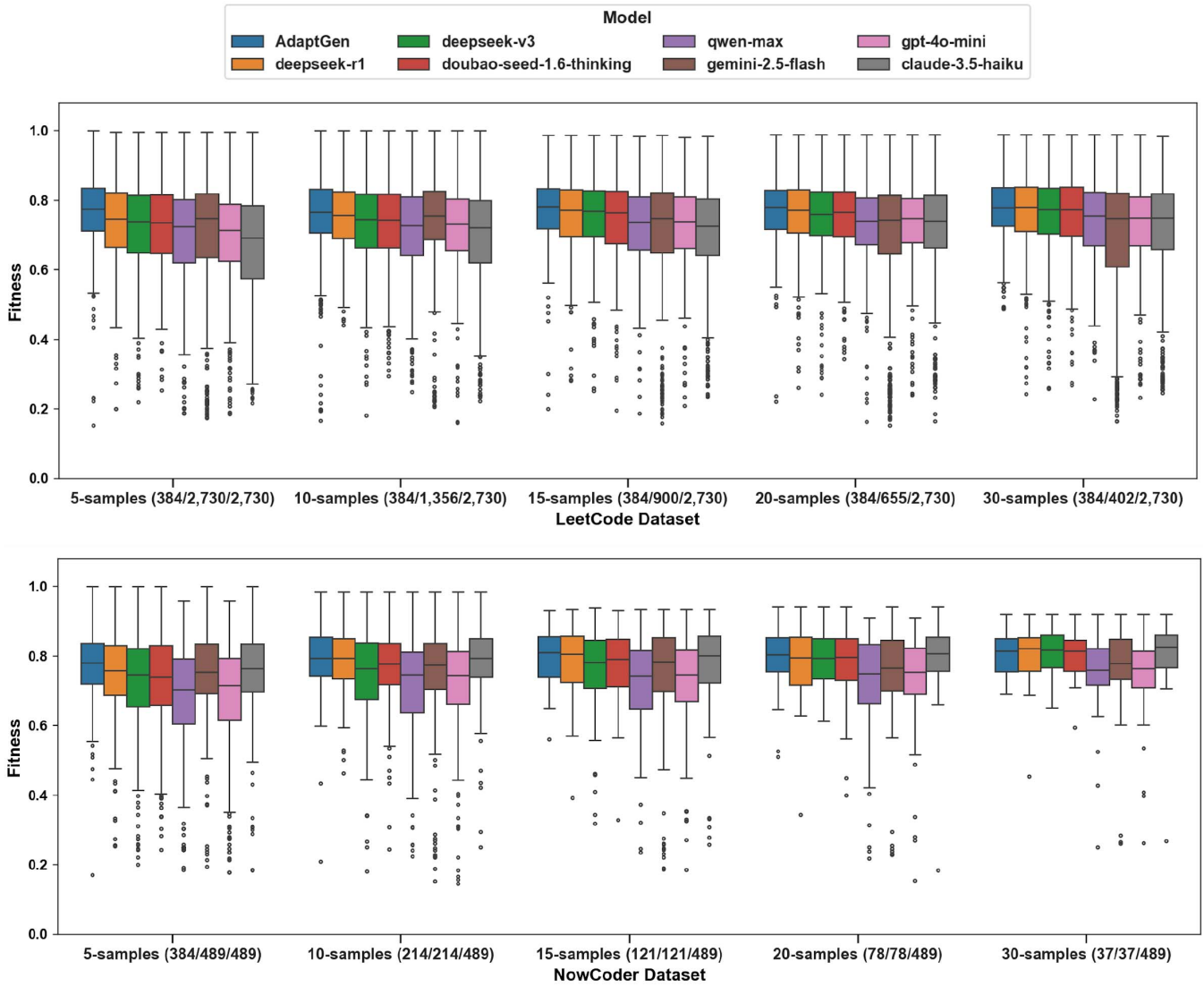


Fig. 8. Box plots comparison of template quality evaluated by fitness scores across different models on LeetCode and NowCoder datasets with varying numbers of solutions.

when relying on increased input samples to achieve comparable quality in a possibly unstable generation way.

Response Efficiency. We mainly compute the median response time for each model across all sample sizes and datasets to measure their response efficiency. Based on our statistics, the fastest model is GPT-4o-Mini, with medians ranging from 6.4s to 12.3s. Following GPT-4o-Mini are Claude-3.5-Haiku (14.4s–30.5s), Qwen-Max (19.9s–26.3s), and DeepSeek-V3 (22.0s–37.4s). Our AdaptGen shows moderate efficiency, with median response time increasing from 21.7s to 60.7s as the sample size grows from 5 to 30. Two reasoning models, i.e., Doubao-seed-1.6-thinking and DeepSeek-R1, are the slowest, whose median response time are from 1min 45.9s to 5min 51.2s, and 1min 50.2s to 6min 5.3s, respectively.

RQ3 Results Summary: AdaptGen demonstrates a robust combination of high-quality, stable solution generation and competitive response efficiency, making it a highly effective approach for templated solution generation across varying datasets and sample sizes.

V. User Study

To validate the practical utility of AdaptGen in real programming scenarios, we conducted a user study evaluating its impact on coding efficiency and learning experiences. We developed a browser extension[8] to integrate AdaptGen's template generation capabilities directly into LeetCode platforms, enabling seamless template usage during problem-solving activities.

A. Participants and Programming Tasks

1) Participants: We recruited 12 programmers with diverse backgrounds for our user study. All participants had computer science backgrounds. As detailed in Table VII, we collected comprehensive demographic and technical profiles including gender, professional roles, educational backgrounds, years of development experience, Java proficiency, commonly used programming languages, number of solved problems on programming platforms, and primary purposes for programming

[8]https://github.com/excuse2020/AdaptGen-crx/

TABLE VII
PARTICIPANT PROFILES: DEMOGRAPHICS AND TECHNICAL EXPERTISE

Attribute	Values (P1–P12)
Gender	male, male, male, male, male, male, male, male, male, male, female, female
Role	student, student, developer, developer, student, student, student, student, project leader, project leader, student, product manager
Education Level	graduate student, graduate student, bachelor, bachelor, graduate student, graduate student, undergraduate, graduate student, bachelor, master, graduate student, master
Development Experience	4 years, 3 years, 5 years, 5 years, 4 years, 5 years, 2 years, 3 years, 5 years, 16 years, 5 years, 15 years
Java Proficiency	2 years, 2 years, 3 years, 5 years, 3 years, 4 years, 1 year, 2 years, 1 year, 13 years, 3 years, 3 years
Programming Languages	python, java, cpp, java, java, java, java, cpp, java, java, java, python, R
#Solving Problems	0-50, 0-50, 200-300, 300-400, 600-700, 700-800, 0-50, 300-600, 1000+, 100-200, 100-200, 800-900
Purpose	learning, learning, learning, learning, job, job, learning, job, competition, learning, learning, job

practice. The diverse representation across multiple dimensions shown in Table VII covers a wide spectrum of potential user groups, enabling validation of AdaptGen's applicability across different user backgrounds.

2) Programming Tasks: We carefully curated our evaluation dataset, i.e., the programming tasks that participants need to finish, by following a systematic problem selection process. In detail, our approach involved selecting multiple distinct algorithmic categories that represent the most common data structures and algorithms encountered in programming practice. For each category, we planned to include multiple problems to address the potential randomness: We aimed to identify problems with high similarity-sharing the same fundamental solution framework and comparable difficulty levels, while differing only in the specific content that needs to be filled into the template based on individual problem contexts. This could help us obtain more generalizable results and verify the template's adaptability across different problems within the same algorithmic domain. Additionally, we needed to ensure that all selected problems possessed high-quality reference templates for meaningful assessment, enabling us to evaluate whether good solution templates can truly play an effective role in programmers' learning and problem-solving processes. Besides, the difficulty distribution of selected programming problems were designed to align with that in the LeetCode platform, on which the ratio of easy:medium:hard problems approximates 1:2:1.

Through this systematic approach, we successfully identified 8 distinct algorithmic categories that met our quality and similarity criteria: Hash Table, Depth-First Search (DFS), Breadth-First Search (BFS), Backtracking, Dynamic Programming, Sliding Window, Stack, and Monotonic Queue. For each category, we selected two problems that met all our criteria, including similar solution strategies, acceptance rates, difficulty levels, and code structures. Additionally, both problems had high-quality templates available for use. This systematic approach resulted in a total of 16 programming tasks across 8 problem sets (2 sets of easy problems, 4 sets of medium problems, and 2 sets of hard problems, with a difficulty distribution being 1:2:1 just as mentioned before), providing a robust foundation for evaluating template generation methods across diverse algorithmic scenarios while ensuring meaningful assessment results.

B. Data Collection

Each participant was required to complete all 16 programming problems using the AdaptGen browser extension. For each task, participants followed the standardized workflow outlined in the user manual: generating candidate templates, selecting the appropriate one, copying and pasting the template into the code editor, completing the implementation, submitting the solution, and finally providing feedback on the template. To evaluate the effectiveness of AdaptGen, we adopted a two-pronged data collection strategy that integrated both quantitative logging and qualitative feedback.

1) Quantitative Logging of User Behavior: During the experiment, the browser extension automatically recorded detailed user interactions, which enabled us to extract several key behavioral metrics, including template selection order, copy-paste content, and final submitted code. For **template selection order**, the extension provided K candidate templates (with $K = 10$ by default), sorted in descending order of fitness scores. By analyzing the position of the selected template in the ranked list, we assessed whether users preferred the top-ranked templates and whether the fitness scores align with human preferences. For **copy-paste content**, we logged the specific code segments copied from the template and pasted into the editor to study how users adapted or reused the generated content. For **final submitted code**, when users submitted their solutions on the LeetCode platform, the extension prompted them to report the submission result (success or failure) and submit their written code for analysis. We evaluated the correctness and usability of the templates by comparing the final submitted code with the selected templates and analyzing the differences between them.

2) Qualitative User Feedback: In addition to behavioral data, we collected structured user feedback through a post-task survey that assessed both the correctness and the usefulness of the solution templates.

To evaluate **template correctness**, users were asked to rate the template on two dimensions: logic correctness and structural and syntactic correctness, both measured on a 1–5 Likert scale. *Logic correctness* refers to whether the template accurately reflects the logical structure and solution strategy of the problem (5 = perfectly aligned, 1 = completely misaligned). *Structural and syntactic correctness* refers to whether the template contains structural or syntactic errors (5 = no errors, 1 = multiple serious errors). To figure out exactly what structure/syntax errors a template tends to have, we provided a checklist of common template issues for participants to select from, including structural issues (e.g., indentation errors, unmatched brackets), incorrect statement ordering, variable naming issues (e.g., meaningless or undefined variables), function naming

issues (e.g., inconsistent or unclear names, mismatched usage), and other user-identified problems.

To evaluate **template usefulness**, users were asked to rate the template from four aspects, including learning facilitation, efficiency improvement, the value of code style, and overall usefulness using 1–5 Likert scales. *Learning facilitation* measures whether the template helped users understand the problem-solving approach for this type of problem (5 = very helpful, 1 = not helpful). *Efficiency improvement* evaluates whether the template improved problem-solving efficiency (5 = significantly improved, 1 = significantly hindered). For this aspect, users could also indicate potential reasons for improvement or hindrance, such as whether the template provided a good starting point, helped them focus on the core logic, inspired or reinforced their solution ideas, contained incorrect logic or code errors, diverged from their intended solution, used an unfamiliar or undesirable coding style, or other user-specified reasons. *The value of code style* assesses whether the coding style reflected in the template is worth learning (5 = very worthwhile, 1 = not worthwhile). *Overall usefulness* captures participants' overall perception of the template, considering all aspects (5 = very useful, 1 = not useful at all).

We also included an optional open-ended question to encourage participants to provide explanations for their rating scores, describe their usage experience, outline the specific scenarios in which they used the templates, and comment on the benefits and limitations of the templates.

This combined evaluation offers both objective and subjective perspectives: quantitative logs capture detailed usage behaviors, while qualitative feedback reflects users' perceived template quality and usefulness, together enabling a comprehensive understanding of template effectiveness across user types and problem categories.

C. Result Analysis

In total, we collected 192 task records covering different users and problem types. Each piece of task record includes both template-using behavior of programming participants and their feedback. Table VIII present the overall statistical results related to the usefulness/potential of our AdaptGen in real-world online programming platforms.

1) Template Usage Behavior: In this part, we mainly checked three aspects. We first checked the rationality of our fitness function particularly designed for our template-generated task, by recording the selection ranks of several templates ordered by fitness scores. Then we studied whether the coders copied the whole template or part of it to help them finish programming. Last, we explored whether the programmers modified our provided templates during their coding by comparing the submitted accepted code with our provided template. The results are as follows.

In the user study, we provided 10 templates with largest fitness scores in an descend order (i.e., rank 1 to 10) to the programmer, he/she could select one template to use in their problem-solving practice. Based on our statistics, we found that **the average/median rank** of selected templates is 1.93/1. This means participants generally selected the templates in the top 1 or 2 positions. Some lower-ranked templates were chosen mainly when they happened to contain more of the code lines needed by the programmers. Considering that the templates are ranked based on their fitness scores, this largely supports that the design of our fitness function aligns reasonably well with user preferences, providing validation of its rationality.

TABLE VIII
SUMMARY OF USER BEHAVIOR AND FEEDBACK

Item	Result
Total Records	192 ($12 \times 8 \times 2$)
Template Usage	
Average/Median Template Rank	1.93 / 1
Copy Behavior	Full Template (174 / 192)
Template Modification	0.83 / 1 (3.7)
Template Correctness	
Logic Correctness	4.36 / 5
Structure and Syntax Correctness	4.68 / 5
Structure Errors	0 / 192
Incorrect Statement Ordering	0 / 192
Variable Name Issues	23 / 192
Function Name Issues	0 / 192
Other user-identified problems	0 / 192
Template Usefulness	
Learning Facilitation	4.23 / 5
Efficiency Improvement	4.31 / 5
Good Start Point	181 / 192
Help Focus on Core Logic	178 / 192
Idea Boosting	176 / 192
Quality Issues	15 / 192
Logic Mismatch	5 / 192
Unfamiliar Coding Style	5 / 192
Code Style Learning Value	4.33 / 5
#Other User Specified Comments	123 / 192
Overall Rating	4.30 / 5

After participants selected templates, we further recorded whether they directly used the whole template or they used just part of the selected template. We found that, in most cases (174 out of 192 records), participants **copied the entire template,** indicating strong trust in the templates as starting points. In few cases (18), participants copied only specific code blocks from the selected template, this happened due to factors like encountering mismatched styles or components they preferred to implement independently.

To get deeper insights into the use of our provided templates, we further studied to what extent the original templates got modified by participants during their programming. Specifically, for each accepted code solution submitted by participants, we transformed the code into a template by using the same template finalization steps in II-C to get its corresponding template. Then we compare the obtained template with our provided templates by computing their edit distance and similarity (by using formula 4). The results yielded an average edit distance of 3.7 and a normalized similarity score of 0.83, indicating that users made minimal **modifications** to the original template structure. Few cases with a low normalized similarity score

(< 0.5) tend to be accompanied by user-reported Quality Issues (recorded in the Template Usefulness below). This suggests that most templates were both structurally correct and practically effective in supporting the development of the final solution.

2) Template Correctness: As designed, we adopted a 1-5 Likert Scale for participants to rate the correctness of our templates related to problem-solving logic, code syntax, code structure, statement ordering, variable and function naming. As a result, we obtained an average Likert rating of 4.36 for logic correctness, which means our provided templates correctly reflect the problem-solving logic for their programming tasks at hand. A high Likert value of 4.68 for structure/syntax correctness means our template code rarely contains structure and syntax errors; actually, among 192 collected records, none reflected structural errors. Meanwhile, no statement ordering errors were found. Variable naming issues appeared in 23 out of 192 cases[9]. Function naming issues and other potential issues were not observed at this stage. The above-mentioned results demonstrate the reliability of our templates in maintaining correctness.

3) Template Usefulness: According to our 1–5 Likert-scale results, participants rated the templates highly in terms of learning facilitation (4.23), with the reported value primarily encompassing the acquisition of problem-solving patterns and coding styles. Regarding their workflow, participants perceived the templates as helpful aids, giving a high rating of 4.31 for their feeling of improved efficiency. This rating reflects the participants' subjective experience (given the absence of a no-template baseline for objective comparison) that the templates facilitated their process and made them feel more productive. Among 192 evaluations submitted by our participants, the vast majority of responses (181/192) agreed that the templates were helpful in providing a good starting point for their task; 178 out of 192 mentioned that the templates could help programmers focus on core problem-solving logic, while 176 highlighted its role in enhancing/inspiring ideas. Other potential usefulness mentioned by participants include the ability of templates to make problem-solving more structured, reduce redundant work through framework reuse, and help users identify overlooked edge cases by comparing the template structure with their own approach, thereby preventing incorrect submissions. Among 192 records, 15 mentioned template quality issues, these issues were primarily related to naming inconsistencies. Five reported logical correctness issues and another five noted discomfort with the coding style. Additional factors that may reduce efficiency include confusion among beginners due to the perceived excessive use of placeholders, which can make the templates harder to understand initially. Templates derived from formatted accepted solutions received an average Likert rating of 4.33 in coding style, suggesting that they align well with Java programming conventions and provide a familiar structure for users.

4) Overall Satisfaction: Based on the collected 192 Likert-Scale (1-5 levels) rating results on the overall satisfaction of our provided templates, we obtained an average rating of 4.30 (the ideal score is 5), reflecting broad user satisfaction. This indicates that participants generally found the templates helpful and effective in supporting their programming tasks. Further, according to participants' feedback, we found AdaptGen provided differentiated support for programmers at different skill levels; to make it even more practical, users across experience levels contributed varied suggestions for refinement. Specifically, **Beginners** found templates helpful for understanding solution structure and reducing confusion. They requested inline comments and more informative placeholders (e.g., hints inside `if(...)`). **Mid-level users** particularly valued the structural organization of the templates, as it helped stimulate their problem analysis and solution design. They also recommended including brief explanations of key steps to further enhance the learning experience. **Advanced users** viewed templates as tools to improve efficiency and focus on core problem-solving logic. They suggested offering advanced configuration options, such as allowing users to customize placeholder names or adjust the level of detail presented in placeholders.

5) Qualitative Feedback and Template Case Analysis: To help readers gain a clearer understanding of AdaptGen's strengths and limitations, we have made the detailed user feedback available on our website[10]; some representative template cases, including both highly rated examples and failure or edge cases (e.g., templates misaligned with problem logic) are also accessible on the website[11] for reference.

VI. Threats To Validity

AdaptGen is primarily designed for Java programs. Although we cannot guarantee that our findings are directly applicable to other languages, our design allows AdaptGen to be adapted for use with them by, for example, replacing the Java-specific JDT parser with the corresponding parser built within other languages, or using a code translation tool to transform code written in other languages to Java. Actually, we have implemented a C/C++ version of AdaptGen (mainly by replacing JDT with CDT, also released in our AdaptGen website) and tested it on C/C++ programming tasks (41 categories in total) from LeetCode. We obtained a median fitness score of 0.795, which, to some extent, demonstrates the generalizability of our AdaptGen across languages.

AdaptGen derives templates from solution groups obtained via structural-similarity-based clustering, thus the choice of the similarity threshold s is non-trivial. In our experiments, we set $s = 0.6$. As a post hoc check, we also examined 0.5 and 0.7 by randomly sampling 100 clustered categories for each s setting from the combined LeetCode and NowCoder datasets, stratified by problem difficulty. For $s = 0.5$, 0.6, and 0.7, we observed 6, 3, and 3 cases, respectively, where different solution patterns were grouped into the same cluster. Although higher s values may further reduce mis-grouping, they can also lead to overly fine-grained groups with only minor differences between them, thereby reducing intra-cluster diversity and limiting template

[9]To make AdaptGen more practical for online programming platforms, we have added a new feature into our shared AdaptGen plugin that integrates LLMs to automatically detect and correct inconsistent variable names produced by our genetic algorithm.

[10]https://zzzzzzgw.top/feedback/

[11]https://zzzzzzgw.top/cases-display/

generality. We regard $s = 0.6$ as a reasonable balance between cluster purity and template generality.

For the fitness function, the metrics used to capture the template requirements and their weights were determined mainly based on our practical experience, literature review, and pilot studies. Despite their rationale having been verified through both our quantitative and qualitative analysis (in three RQs and our user study), it remains worthwhile to further investigate whether alternative metrics or weight configurations could yield additional improvements.

Three key parameters of the genetic algorithm were mainly determined through our small-scale pilot study. Although our supplemental sensitivity analysis confirms that the chosen configuration performs well on the LeetCode and NowCoder datasets in terms of generation quality and evolutionary efficiency, it is still valuable to explore additional parameters and broader value ranges.

In RQ2, establishing the scoring criteria was a critical step. Due to the absence of directly relevant references, the determination of these values was primarily based on the authors' research on programming data and the experience accumulated from their practical work and research. In the future, we plan to develop a plugin for programming platforms and implement it in practice. By collecting programmer feedback, we aim to derive more realistic and widely applicable experimental results, which will help in refining the scoring standards.

In RQ3, we selected representative and high-performing LLMs available at the time and conducted multiple rounds of prompt optimization to maximize the quality of model outputs. Nevertheless, we acknowledge that the rapid evolution of LLMs may lead to even better-performing models in the near future. Additionally, prompt design remains an open problem. There may exist more effective prompts that have not yet been explored, which could potentially affect the evaluation results.

In the user study, several factors may also threaten the validity of our study. The limited sample size of 12 participants may not fully represent the broader programming community, and subjective template quality assessments introduce potential bias from individual programming preferences. The selected problem sets, while systematically chosen, may not cover the complete spectrum of real-world programming challenges, and the controlled experimental environment may differ from actual programming scenarios. Additionally, since participants solved all problems with the AdaptGen templates available, our evaluation of AdaptGen's usefulness is primarily based on participants' subjective perceptions, such as whether they felt the templates were helpful and whether they perceived improvements in efficiency. While these results reflect users' perceived benefits, they do not directly quantify performance gains compared to a workflow without templates.

VII. Related Work

A. Code Generation

Research on generating solution templates for online programming platforms is limited, though it intersects with the broader field of code generation, including tasks like code completion, program synthesis, and code repair [13], [22], [23], [24], [25]. Our work aligns with program synthesis, which generates code from specifications to achieve specific functionality. Traditional program synthesis approaches include deductive reasoning, grammar-guided, and code search-based methods. Manna et al. [26], [27] proposed a deductive reasoning approach that uses constructive proofs to transform specifications into propositions, enabling automatic program generation. Huang et al. [28] introduced grammar-guided synthesis, combining semantic specifications with structural constraints through decision trees and counterexample-guided inductive synthesis. Swim [29] developed a code search-based method to retrieve relevant API names and generate code based on API patterns. Recent advancements in deep learning have led to the use of large models for program synthesis [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40]. DeepCoder [31] generates programs from input-output examples using a domain-specific language and neural networks. AlphaCode [32] generates competitive solutions based on problem descriptions, outperforming human competitors in programming contests. Mathews et al. [35] integrate Test-Driven Development (TDD) into LLM-based code generation, showing that test cases significantly improve code correctness. Chen et al. [36] proposed MANGO, which leverages code comments as logical pivots to enhance small and medium-scale code LLMs, significantly improving generation accuracy and robustness. Yan et al. [37] conducted a large-scale study on ChatGPT's zero-shot performance in competition-level code generation, showing 25.4% test case success and potential improvement with feedback. Austin et al. [40] explored LLMs for generating short Python programs from natural language descriptions, showing promising results with few-shot learning and fine-tuning.

LLM-based code generation methods produce complete, executable solutions based on the problem description, focusing on correctness, i.e., passing test cases. In contrast, AdaptGen provides programmers with a structured starting point rather than a full solution. It leverages AC code shared by real programmers and employs a genetic algorithm guided by a tailored fitness function to generate templated solutions that capture common solution patterns, maintain good structure, and possess high readability. These templated solutions emphasize generality, ensuring applicability across programmers and alignment with common coding practices. LLM-generated code, in contrast, may be overly or insufficiently optimized and lacks generality. Providing complete code can also lead to over-guidance. To mitigate this, AdaptGen uses a core-code hiding strategy to conceal essential logic, producing solution templates that balance guidance with independent thinking. A more detailed side-by-side comparison is provided in our supplemental material.

B. Genetic Programming

Our method employs a genetic algorithm inspired by Genetic Programming (GP) principles. GP is a stochastic search technique that evolves computer programs toward specific functional or quality goals and has shown effectiveness in code

generation and repair tasks. For example, Illanes et al. [12] used GP to generate code for object-oriented and dynamically typed languages by representing programs as ASTs and performing crossovers by swapping subtrees, with fitness determined by the number of test cases passed. Similarly, Weimer et al. [13] applied GP for code repair, representing programs through ASTs and weighted execution paths. Le Goues et al. [15] optimized this approach with an efficient encoding method using an edit sequence of AST node operations to reduce space requirements, while Oliveira et al. [41] improved diversity in generated individuals. Unlike previous GP approaches for code generation and repair, which rely on AST-based or edit-sequence representations, AdaptGen adopts a linear hash sequence encoding. This approach selectively preserves only the features relevant to the evolutionary process, rather than all syntactic details. It enables direct computation of fitness metrics without converting sequences back to full code, as required by traditional methods. Genetic operations are also performed linearly, resulting in high time and space efficiency for our template generation. A detailed comparison is provided in our supplemental material.

C. Code Similarity Detection

Code similarity detection is a key component of AdaptGen, used for classifying AC code, calculating fitness, and evolving templates. Several methods have been developed to measure code similarity, including text, token, syntax tree, and control flow graph-based approaches. Early methods, such as Johnson et al. [42], employed text-based fingerprinting to identify duplicate text segments, while Kamiya et al. [43] introduced a token-based approach through the CCFinder tool, comparing code tokens sequentially. Tree-based methods, like those from Sager et al. [44], use common subtree isomorphisms and tree edit distance to measure similarity. GPLAG [17] detects code similarity using Program Dependence Graphs for plagiarism detection. More recently, machine learning techniques have been used for detecting code similarity. Alon et al. [18] developed Code2Vec, which represents code fragments as fixed-length vectors from AST paths to predict semantic properties. Liu et al. [19] proposed Tailor, using deep graph-structured code features and Code Property Graphs to detect functional similarity. Ahn et al. [45] introduced BinShot, a BERT-based [46] framework for Binary Code Similarity Detection.

While these advanced methods are effective in capturing fine-grained similarities, they are computationally expensive and may not be ideal for our needs. Since generating general-purpose solution templates does not require detailed similarity matching, we utilize a method that integrates lexical, semantic, programmatic, and structural features. This approach offers a balance between matching accuracy and computational efficiency, enhancing the effectiveness of AdaptGen.

VIII. Conclusion

In this paper, we introduced AdaptGen, a novel solution template generation technique based on genetic algorithms. AdaptGen maps the requirements for solution templates onto the fitness function within the genetic algorithm, enabling the adaptive generation of solution templates for various programming challenges. We validated its effectiveness using datasets from LeetCode and NowCoder, showing that AdaptGen efficiently generates high-quality solution templates within a limited number of iterations. The comparisons of AdapteGen over typical LLMs and our conducted user study further demonstrated the necessity and usefulness of our AdaptGen in constructing solution templates for programmers on online programming platforms.

As future work, we aim to enhance AdaptGen's capability in handling programming tasks without available solutions. Currently, AdaptGen is primarily designed to generate solution templates for programming tasks with available accepted code. For new tasks lacking accepted solutions, AdaptGen would face challenges in generating appropriate templates. To overcome this limitation, future versions of AdaptGen could explore integrating LLMs, leveraging their strengths in code generation while mitigating their generation instability by guiding them with high-quality templates from other programming tasks generated by AdaptGen. This integration could significantly enhance AdaptGen's capacity to handle a broader range of programming problems, improving both the accuracy and adaptability of its generated templates. In addition, it is also interesting and valuable to exploit the potential of our hash sequence encoding strategy in other challenging tasks involving program semantic representation, such as clone detection and code generation with various structural complexity.

References

[1] M. Resnick et al., "Scratch: programming for all," *Commun. ACM*, vol. 52, no. 11, pp. 60–67, 2009.
[2] A. A. DiSessa, *Changing Minds: Computers, Learning, and Literacy.* Cambridge, MA, USA: MIT Press, 2000.
[3] J. Wing, "Computational thinking," *Commun. ACM*, vol. 49, no. 3, pp. 33–35, 2006.
[4] I. D. Baxter et al., "Clone detection using abstract syntax trees," in *Proc. Int. Conf. Softw. Maintenance*, 1998, pp. 368–377.
[5] J. Li, Y. Li, G. Li, Z. Jin, Y. Hao, and X. Hu, "SkCoder: A sketch-based approach for automatic code generation," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.,* 2023, pp. 2124–2135.
[6] C. Zhu-Tian et al., "Sketch then generate: Providing incremental user feedback and guiding LLM code generation through language-oriented code sketches," 2024, *arXiv:2405.03998*.
[7] X. W. Yang et al., "Neuro-symbolic artificial intelligence: Towards improving the reasoning abilities of large language models," 2025, *arXiv:2508.13678*.
[8] B. P. Bhuyan et al., "Neuro-symbolic artificial intelligence: A survey," *Neural Comput. Appl.*, vol. 36, no. 21, pp. 12809–12844, 2024.
[9] A. Solar-Lezama, "Program sketching," *Int. J. Softw. Tools Technol. Trans.*, vol. 15, no. 5, pp. 475–495, 2013.
[10] V. Murali et al., "Neural sketch learning for conditional program generation," 2017, *arXiv:1703.05698*.
[11] M. Brameier, *On Linear Genetic Programming.* Dortmund, Germany: Tech. Univ. of Dortmund, 2004.
[12] V. Illanes and A. Bergel, "Generating object-oriented source code using genetic programming," in *Proc. Genetic Improvement Workshop*, 2021, pp. 45–50.
[13] W. Weimer, T. V. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proc. IEEE Int. Conf. Softw. Eng.*, 2009, pp. 364–374.
[14] Z. Chen and M. A. Monperrus, "Literature study of embeddings on source code," 2019, *arXiv:1904.03061*.

[15] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair," in *Proc. IEEE Int. Conf. Softw. Eng.*, 2012, pp. 3–13.
[16] M. Sudhamani and L. Rangarajan, "Code similarity detection through control statement and program features," *Expert Syst. Appl.*, vol. 132, pp. 63–75, Oct. 2019.
[17] C. Liu et al., "GPLAG: detection of software plagiarism by program dependence graph analysis," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2006, pp. 872–881.
[18] U. Alon et al., "code2vec: Learning distributed representations of code," *ACM Program. Lang.*, vol. 3, no. POPL, pp. 1–29, 2019.
[19] J. Liu et al., "Learning graph-based code representations for source-level functional similarity detection," in *Proc. Int. Conf. Softw. Eng.,* 2023, pp. 345–357.
[20] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 176–192, Mar. 2006.
[21] R. V. Krejcie and D. W. Morgan, "Determining sample size for research activities," *Educ. Psychological Meas.*, vol. 30, no. 3, pp. 607–610, 1970.
[22] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proc. Eur. Softw. Eng. Conf. ACM SIGSOFT Symp.*, 2009, pp. 213–222.
[23] A. Hindle et al., "On the naturalness of software," *Commun. ACM*, vol. 59, no. 5, pp. 122–131, 2016.
[24] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.,* 2023, pp. 1482–1494.
[25] A. Svyatkovskiy et al., "IntelliCode Compose: Code generation using transformer," in *Proc. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 1433–1443.
[26] Z. Manna and R. Waldinger, "Synthesis: Dreams→ programs," *IEEE Trans. Softw. Eng.*, vol. SE-5, no. 4, pp. 294–328, Jul. 1979.
[27] Z. Manna and R. Waldinger, "A deductive approach to program synthesis," *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 1, pp. 90–121, 1980.
[28] K. Huang et al., "Reconciling enumerative and symbolic search in syntax-guided synthesis," 2018, *arXiv:1802.04428*.
[29] M. Raghothaman, Y. Wei, and Y. Hamadi, "SWIM: Synthesizing what I mean: code search and idiomatic snippet synthesis," in *Proc. Int. Conf. Softw. Eng.,* 2016, pp. 357–367.
[30] M. Chen, et al., "Evaluating large language models trained on code," 2021, *arXiv:2107.03374*.
[31] M. Balog et al., "DeepCoder: Learning to write programs," 2016, *arXiv:1611.01989*.
[32] Y. Li et al., "Competition-level code generation with AlphaCode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.
[33] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. Findings Assoc. Comput. Linguistics, EMNLP*, 2020, pp. 1536–1547.
[34] C. B. Clement et al., "PyMT5: Multi-mode translation of natural language and Python code with transformers," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2020, pp. 9052–9065.
[35] N. S. Mathews and M. Nagappan, "Test-driven development and LLM-based code generation," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2024, pp. 1583–1594.
[36] Y. Chen et al., "Comments as natural logic pivots: Improve code generation via comment perspective," in *Proc. Findings Assoc. Comput. Linguistics (ACL)*, 2024, pp. 7040–7051.
[37] D. Yan, Z. Gao, and Z. Liu, "A closer look at different difficulty levels code generation abilities of ChatGPT," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2023, pp. 1887–1898.
[38] X. Yu et al., "DroidCoder: Enhanced Android code completion with context-enriched retrieval-augmented generation," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2024, pp. 681–693.
[39] S. Dou et al., "StepCoder: Improving code generation with reinforcement learning from compiler feedback," in *Proc. Annu. Meeting Assoc. Comput. Linguistics*, 2024, pp. 4571–4585.
[40] J. Austin et al., "Program synthesis with large language models," 2021, *arXiv:2108.07732*.
[41] V. P. L. Oliveira et al., "Improved crossover operators for genetic programming for program repair," in *Proc. Int. Symp. Search Based Softw. Eng*., 2016, pp. 112–127.
[42] J. H. Johnson, "Identifying redundancy in source code using fingerprints," in *Proc. Centre Adv. Studies Collaborative Res.,* 1993, pp. 171–183.
[43] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002.
[44] T. Sager et al., "Detecting similar Java classes using tree algorithms," in *Proc. Int. Workshop Mining Softw. Repositories*, 2006, pp. 65–71.
[45] S. Ahn et al., "Practical binary code similarity detection with BERT-based transferable similarity learning," in *Proc. Annu. Comput. Secur. Appl. Conf.,* 2021, pp. 361–374.
[46] J. Devlin et al., "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. North Amer. Chapter Assoc. Comput. Linguistics, Human Lang. Technol.*, 2019, pp. 4171–4186.