PDL: Scaffolding Problem Solving in Programming Courses

Shu Lin∗ Na Meng† Dennis Kafura† Wenxin Li∗
Peking University∗ Virginia Tech†
China∗ USA†

Abstract
Programming tasks provide an opportunity for students to improve their problem-solving skills (PSS). However, when programming tasks are challenging, students could become demotivated and lose the opportunity to improve PSS in the process. To scaffold the difficulty of programming tasks and better motivate students to enhance PSS via coding, this paper introduces PDL (Problem Description Language). Given the natural-language description of a combinatorial optimization problem (COP), PDL requires students to describe (i) inputs, (ii) constraints, (iii) the optimization objective, and (iv) outputs, based on their problem comprehension. PDL then validates each problem description by (1) compiling a solution program from the description and (2) executing the generated program with predefined test cases. Based on the compiling and testing results, PDL provides feedback to students, and assists students to adjust their problem comprehension and improve problem descriptions.

To evaluate PDL’s effectiveness in motivating students to fulfill challenging programming tasks, we conducted a user study with 185 undergraduates and asked the students to solve COPs with or without PDL. We found that the students with PDL were less likely to give up than students without PDL. By using PDL, students solved more COPs and spent less time on each problem; they became more confident and motivated in handling COPs after using PDL.

1 Introduction
Problem solving in Computer Science mainly involves two parts: problem comprehension and solution development. Problem-solving skills (PSS) are important for students to succeed in programming courses. Prior studies show that the lack of such skills in novice developers helps explain the tremendously high failure rate in computer science [3, 5, 14]. Tu and Johnson observed that students could improve PSS by coding for various programming tasks [25]. However, when the tasks are very difficult and daunting, students are demotivated to fulfill those tasks, lose the opportunity to hone PSS, or even gain negative attitudes towards the computing field [13].

According to our experience of teaching programming courses, students found combinatorial optimization problems (COP) [6] to be especially hard. A typical COP requires a search for the optimal solution in a finite solution space. A programming problem is COP, if given (1) input parameters \( A = \{a_1, \ldots, a_M\} \) and their value ranges, (2) variables \( V = \{v_1, \ldots, v_N\} \), value ranges, and their value relations with \( A: R = \{r_1(A, V), \ldots, r_L(A, V)\} \), and (3) an objective function \( f(A, V) \). The program will find the optimal value assignments for \( \{v_1, \ldots, v_N\} \) such that (i) \( R \) is satisfied, and (ii) the value of \( f(A, V) \) is optimal. Exemplar COPs include 0/1 knapsack and shortest path problems. Based on interactions with students, we learnt that a COP is difficult for two reasons:

- **Problem Comprehension.** Some students could not interpret problems correctly and thus built incorrect programs.
- **Solution Development.** Some students interpreted problems correctly, but developed incorrect programs.

To help improve students’ PSS while they program for challenging COPs, we developed a novel tool—PDL—that scaffolds problem solving by decoupling problem comprehension and solution development. As a scaffolding technique, PDL suppresses overly complex coding issues that students are not initially ready to encounter. It helps students analyse problems, formulate problem descriptions, learn about the resulting code for formulated problems, and gain the confidence as well as abilities before they independently program for COPs. Our user study shows that with PDL’s detailed feedback on students’ problem descriptions, students were better motivated to solve COPs.

2 Background
The related work of our research includes studies on the relationship between PSS and CS education, and existing scaffolding techniques.

2.1 Problem-Solving Skills and CS Education
Researchers found that PSS are important for students to succeed in CS Education [3, 5, 14]. For instance, Beaubouef and Mason [3] reported that in many institutions, 30–40% of CS undergraduates...
dropped out for reasons like (1) poor math skills and problem solving abilities, (2) poorly designed CS1 lab courses, and (3) lack of practice/feedback. Prior studies show that students can improve PSS and computational thinking via programming [10, 24, 25]. For example, Salehi et al. [24] observed that when solving problems irrelevant to their majors, CS students performed significantly better than students in other majors; Salehi et al. also found CS programming assignments to be highly effective in helping students develop PSS. However, based on our teaching experience, when programming tasks (e.g., COP) are very difficult, students can become less motivated to solve problems, lose opportunities to further improve PSS, and even drop out due to the negative experience.

When introducing principles for teaching problem solving, Foschay and Kirckley recommended instructors to emphasize both the declarative and procedural knowledge components of any “real-world” job or task [11]. To address the procedural component, some researchers proposed explicit instructions on programming strategies [7, 15, 16, 18]. For instance, Ko et al. [15] and LaToza et al. [16] invented the teaching methods of using a domain-specific language Roboto to explicitly describe the programming strategy (i.e., a sequence of actions) for accomplishing any task (e.g., debugging). Our research complements existing work by focusing on the declarative component; with PDL, students could learn to declare the constraints that a solution program should satisfy.

2.2 Scaffolding Techniques

"Scaffolding" is a metaphor to capture the nature of support and guidance in learning [9]. Scaffolding techniques are temporary assistance that teachers provide for students. The techniques assist students to complete a task or develop new understandings, so that students can later complete similar tasks alone. One form of scaffolding (e.g., C0 [22] and Ironclad C++ [8]) defines "safe" subsets of general-purpose programming languages (e.g., C/C++). With these domain-specific languages, instructors can teach students basic programming concepts while hiding complicated issues (e.g., memory management). Another form of scaffolding includes visual programming languages to reduce coding complexity [2, 12, 17, 19, 20, 23]. For instance, Scratch is a block-based and object-oriented programming language [19]. It represents program constructs (i.e., if-statement) with distinctly shaped blocks, and users can create programs by dragging and dropping blocks. However, none of these techniques automate algorithm design or thoroughly factor out coding concerns to scaffold problem solving.

3 OUR APPROACH: PDL

As shown in Figure 1, given a COP, students are supposed to describe the problem with PDL, a domain-specific language (DSL). Based on such a PDL description, the PDL compiler analyses the description, automatically designs a solution algorithm, optimizes the design when possible, and generates a C program accordingly. In this section, we will introduce PDL (Section 3.1), explain the implementation of PDL compiler (Section 3.2), and describe PDL’s feedback on any erroneous problem description (Section 3.3). Our program and some PDL description examples are available at http://doi.org/10.5281/zenodo.3961672.

3.1 Language Design

To facilitate problem description, PDL has four sections to mathematically describe a COP:

- **Input Section** declares input arguments (i.e., \( A \)).
- **Required Section** declares variables (i.e., \( V \)), value ranges, and the mathematical formulas that define relations or constraints (i.e., \( R \)) between variables and arguments.
- **Objective Section** defines the objective function (i.e., \( f(A, V) \)).
- **Output Section** defines variables or expressions whose values should be printed to the console.

Arguments and variables can be declared with primitive or composite types. PDL supports four primitive types (i.e., integer, real number, character, and boolean) as well as three composite types (i.e., array, set, and tuple). Users can define formulas using: (1) arithmetic, logical, relational, or exponent operators (e.g., "+", "\not\"", "\ge\", and "\^\""), (2) two logical quantifiers: universal quantifier for all (i.e., \( \forall \)) and existential quantifier exists (i.e., \( \exists \)), (3) aggregate operators: summation, product, min, max, and count, and (4) a predicate alldifferent to declare that all elements in an array are all different.

3.2 Language Implementation

We developed a compiler that takes in PDL descriptions, and goes through five phases before generating C programs (see Figure 1).
3.2.1 Creation of Identifier Table. Given a PDL description, this phase tokenizes the description and conducts both syntactic and semantic analysis to build an identifier table—a table recording arguments, variables, their types, and value ranges. This table is important for PDL to later decide (1) what variables to use in the algorithm-to-design (AOD) to control loop iterations or recursive function calls and (2) how many iterations or recursions are involved in the search procedure for optimality. To minimize the number of iterations or recursions, this step tentatively tightens the value range of each variable using the feasibility-based bounds tightening (FBBT) algorithm [4]. Intuitively, given \( a \in [0, 0], b \in [0, 1], c \in [0, 1], a = b + c \), FBBT first converts the formula to \( b = a - c \), and then shrinks the range for \( b \) as \( D'_b = D_b \cap ([0, 0] - [0, 1]) = [0, 1] \cap [-1, 0] = [0, 0] \). PDL applies FBBT to variables iteratively until all value ranges become stabilized, and records the resulting ranges in the identifier table.

3.2.2 Identification of Controlling Variables. Generally speaking, any COP-solution algorithm enumerates value combinations between variables to search for the optima. As shown by Algorithms 1 and 2, the algorithm-to-design (AOD) is based on either iterations or recursions. Thus, this phase decides the controlling variables for either loop iterations or function recursions in the AOD, by identifying a variable subset \( VS \subseteq V \) such that:

- The values of all other variables (i.e., \( v \in (V - VS) \)) can be uniquely determined by the value assignments of \( VS \).
- The size of \( VS \) (i.e., the Cartesian product of all included variables’ ranges) is minimal. Here, when \( VS = \{ v_1, v_2, \ldots \} \), its size is \( \text{range}(v_1) \times \text{range}(v_2) \times \text{range}(\ldots) \).

Intuitively, PDL conducts brute-force search to investigate all variable subsets and to determine the controlling variables. We found such brute-force search often done efficiently because when variable subsets overlap (e.g., two subsets \( S_1 \) and \( S_2 \) where \( S_1 \subseteq S_2 \)), our approach quickly skips the exploration of unpromising ones (e.g., skip \( S_2 \) when \( S_1 \) is selected as a candidate for controlling variables).

3.2.3 Design of Enumeration-Based Algorithms. After identifying controlling variables, PDL generates a basic algorithm design for the naively exhaustive search. This algorithm enumerates the values of each controlling variable, calculates the values of non-controlling variables, checks whether all constraints are satisfied, and evaluates the objective function if constraints are satisfied. In particular, if all controlling variables have primitive types, PDL generates an iteration-based search algorithm similar to Algorithm 1. Otherwise, if any controlling variable has a composite type, PDL creates a recursion-based search algorithm similar to Algorithm 2. For the 0/1 knapsack problem shown in Figure 2, PDL designs a recursion-based search algorithm because the only variable \( S \) is a set.

3.2.4 Opportunistic Optimization. When a recursion-based algorithm is generated, PDL opportunistically applies two optimization strategies to reduce unnecessary computation: branch pruning and dynamic programming.

Branch Pruning. This optimization adds or moves if statements, to remove unnecessary enumeration of value combinations. Take the 0/1 knapsack problem as an example. With branch pruning, before any step of recursion \( \text{step}(i \in [1, N]) \), PDL adds an if check to decide whether the weight sum so far \( \sum W_i \geq C \), i.e., \( \text{sum}_W > C \). This is because all items have positive weights (i.e., \( w[j] \geq 1 \)). If \( \text{sum}_W \leq C \), even though none of the last \((N - \text{step} + 1)\) items is put into the knapsack, the overall weight sum can never meet the constraint. Consequently, there is no need to involve more steps of recursions, and PDL prunes the search subtree when Equation (1) is satisfied.

Dynamic Programming (DP). For certain COPs, DP algorithms can break down each given problem into simpler subproblems, and compute the optimal solution to the overall problem based on optimal solutions to the subproblems. Compared with a naïve enumeration of value combinations, DP algorithms effectively eliminate unnecessary enumerations. Given a COP, to tentatively refactor a basic algorithm into a DP algorithm, this step automatically characterizes the problem and decides whether a DP algorithm can be generated. In the scenarios when DP is feasible, PDL generates an algorithm that first initializes a table to store optimal solutions for subproblems, and then searches for optimal solutions in a top-down manner. Intuitively, the generated algorithm tries to iterate the overall problem by breaking it into smaller ones recursively, solving the smallest subproblems first, recording the optimal solutions in the table, and gradually solving larger problems by reusing optimal solutions to smaller ones.

3.2.5 Code Generation. To automate the C implementation of each algorithm design, PDL has six major types of predefined code templates to generate different parts of the implementation.

3.3 Feedback Generation

For any incorrect PDL description, our approach provides four major types of feedback, which are similar to errors or warnings generated by a traditional compiler:
• Parsing Errors describe the grammatical or spelling errors located in problem descriptions.
• Type Errors reveal any type conflicts between expressions. PDL reports such errors by showing the minimum erroneous subexpressions together with the inferred types of operands.
• Unbounded Variable Errors are about variables whose upper or lower value boundaries are unspecified.
• Unused Variable Warnings report the variables that are defined but never used in any constraint.

Additionally, PDL also presents the resulting C code generated for any problem description as its feedback. When problem descriptions are incorrect, the generated code together with any test case it fails can help students adjust their problem comprehension and improve descriptions accordingly; when problem descriptions are correct, students can still read the generated code to learn about program implementation and code optimization strategies. We designed such feedback mechanism in PDL for two purposes. First, by decoupling problem comprehension and solution development, we suppress coding issues and help students focus their practices on problem comprehension and description—essential components of PSS. Second, by demonstrating solution development for the problems they described, PDL helps students map problem characteristics to the solution space. Such mappings can prepare students to independently solve COPs later.

4 EVALUATION
To evaluate PDL, we conducted two experiments. The first one explores PDL’s usability by applying PDL to 45 COPs (Section 4.1), while the second one investigates PDL’s helpfulness in motivating students to solve COPs via a user study (Section 4.2).

4.1 Evaluation of PDL’s Usability
Usability indicates in how many scenarios, we can leverage PDL to solve COPs. Intuitively, the more COPs are solvable with PDL, the more usable our tool is to train problem-solving skills in students. Thus, we collected 45 COPs from the exercises and homework assignments of 4 programming courses for CS freshmen and sophomores. The first author then tried to write a PDL description for each COP and use PDL to create the solution program. To automatically assess the quality of PDL descriptions, we built an online judge (OJ) system as shown in Figure 3. After taking in a PDL description, OJ first uses PDL to identify any lexical or syntactic error in the description; if none, OJ then compiles the C code and conducts automatic testing to execute the compiled code with prescribed test cases. Additionally, OJ has a database to record all data of PDL description submissions and related feedback. Based on the feedback or output of OJ, the first author could debug PDL descriptions if they were incorrect, count the number of COPs solvable by PDL, and identify the COPs unsolvable by PDL.

As shown in Table 1, 17 of the 45 COPs are solvable with basic enumeration algorithms; 16 problems can be solved by enumeration with pruning; and 12 problems are solvable by DP as well as pruning. According to the first author’s experience, PDL successfully solved 32 problems (71.1%) after taking in correct problem descriptions. It means that PDL has great usability, because it could solve the majority of COPs. Additionally, PDL partially solved 6 problems (13.3%), because it generated inefficient code with correct program logic. By examining these problems and the answer keys, we found the problems to be very challenging. To efficiently solve the problems, students need to be creative and apply more advanced optimization techniques (e.g., domain-specific search or pruning). There are 7 problems (15.6%) that PDL could not solve, because they involve complex program logic to manipulate graphs or strings. Currently, PDL does not support the problem description or solution generation for such COPs.

**Finding 1:** PDL is quite usable as it generated correct programs for 71.1% of the explored COPs. PDL is also reliable because given a correct problem description, it generated no erroneous program.

### Table 1: PDL’s usability evaluation based on 45 COPs

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Total</th>
<th>Inexpressible</th>
<th>Partly Solvable</th>
<th>Solvable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enumeration</td>
<td>17</td>
<td>2 (11.8%)</td>
<td>0 (0.0%)</td>
<td>15 (88.2%)</td>
</tr>
<tr>
<td>With Pruning</td>
<td>16</td>
<td>3 (18.8%)</td>
<td>3 (18.8%)</td>
<td>10 (62.5%)</td>
</tr>
<tr>
<td>DP &amp; Pruning</td>
<td>12</td>
<td>2 (16.7%)</td>
<td>3 (25.0%)</td>
<td>7 (58.3%)</td>
</tr>
<tr>
<td>DP &amp; Pruning</td>
<td>12</td>
<td>2 (16.7%)</td>
<td>3 (25.0%)</td>
<td>7 (58.3%)</td>
</tr>
<tr>
<td>Total</td>
<td>45</td>
<td>7 (15.6%)</td>
<td>6 (13.3%)</td>
<td>32 (71.1%)</td>
</tr>
</tbody>
</table>

**4.2 Evaluation of PDL’s Helpfulness**

We integrated PDL into the CS course *Introduction to Artificial Intelligence*—a course covering C programming and algorithm design. After students became familiar with C and algorithm design strategies (e.g., enumeration, pruning, and DP), we introduced PDL as a tool that may facilitate COP programming and asked all students to participate in a PDL study as part of the course requirement.

#### 4.2.1 Study Design
Before the study, we gave a nine-page PDL manual [21] to all students. The manual introduces PDL and presents two exemplar COPs (i.e., cuboid and 0/1 knapsack problems) as well as related PDL descriptions. Students were supposed to read the tutorial and learn to use PDL before the study. To conduct a controlled experiment during the study, we instructed all students to independently work on six COPs. As shown in Table 2, the six COPs include two problems solvable with enumeration algorithms, two problems solvable with enumeration and pruning, and two problems solvable with DP and pruning. Generally speaking, the complexity comparison between different algorithms is *Enumeration < With Pruning < DP & Pruning*. The problems are similar to exemplar COPs in the lecture notes but different.

All 185 involved students are undergraduates who took a CS1 programming and algorithm course as the prerequisite. We ranked students based on their grades in CS1; we then divided students into four groups using the serpentine system in order to reduce bias between groups. Table 3 shows the task assignments to different groups. Every student went through two phases. In Phase 1, they solved three COPs either with or without PDL; then in Phase
II, they switched the programming approaches to solve another three problems. As we obtained roughly equal numbers of students working on each problem with or without PDL, such balanced data distribution ensures the fairness of our empirical comparison.

At the beginning of the study, we asked every student to fill a pre-study form to describe their confidence levels in solving COPs. For the study, we extended the OJ system shown in Figure 3 to also take in C program submissions. The system can assess the quality of both PDL descriptions and C programs via compilation and testing. When developing software artifacts (i.e., C code or PDL descriptions), students could access OJ via the Internet, submit artifacts as many times as they like, and receive feedback by OJ. Students were given 30 minutes to solve each problem. After solving three COPs with one method M (with or without PDL), students filled a survey form of four questions:

- Q1. How many minutes did you spend in solving each COP?
- Q2. How difficult or easy was it for you to create software artifacts with method M?
- Q3. How difficult or easy was it for you to debug the artifacts created with method M?
- Q4. How confident are you to solve COPs?

While solving COPs, students recorded the actual time they spent on each problem and answered Q1 based on those records. They answered Q2–Q4 in a five-level Likert scale [1]. Based on students’ response and the collected information in OJ’s database, we explored the following research questions:

RQ1. How well did students solve COPs with or without PDL? To answer this question, we compared the time spent by students on each problem (based on Q1) and the quality of resulting artifacts (based on OJ’s database).

RQ2. What is the complexity comparison between defining PDL descriptions and building C programs? For each COP, we clustered and compared students’ responses to Q2.

RQ3. What is the complexity comparison between debugging PDL descriptions and debugging C code? For each COP, we clustered and compared students’ responses to Q3.

RQ4. How does PDL help improve students’ confidence in solving COPs? We compared the responses by students for Q4 against their responses in the pre-study form.

### Table 2: The six problems used in the second experiment

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Eight Queens</td>
<td>Enumeration</td>
<td>Given $N(1 \leq N \leq 1000)$ integers, find the only one duplicate number.</td>
</tr>
<tr>
<td>P2</td>
<td>Shortest Path</td>
<td>DP &amp; Pruning</td>
<td>Find the shortest path from City 1 to City N.</td>
</tr>
<tr>
<td>P3</td>
<td>Sum Is K</td>
<td>Enumeration</td>
<td>Find two integers among $N(1 \leq N \leq 1000)$ given integers, such that the sum of them is equal to K.</td>
</tr>
<tr>
<td>P4</td>
<td>Messenger Problem</td>
<td>Pruning</td>
<td>There are $N(1 \leq N \leq 10)$ cities.</td>
</tr>
<tr>
<td>P5</td>
<td>Teamwork</td>
<td>DP &amp; Pruning</td>
<td>There are $N(1 \leq N \leq 100)$ candidates. Each candidate has a cooperation value $v$ ($50 \leq v \leq 50$) and a working value $w$ ($-50 \leq w \leq 50$). Select any number of candidates to form a team, such that the sum of all the team members’ cooperation values is positive and the summing of their working value is maximum.</td>
</tr>
<tr>
<td>P6</td>
<td>Mathematical Problem</td>
<td>Enumeration</td>
<td>Given $N(1 \leq N \leq 100)$ integers, such that the sum of them is equal to K.</td>
</tr>
</tbody>
</table>

### Table 3: The tasks assigned to each group

<table>
<thead>
<tr>
<th>Group ID</th>
<th>Designated Tasks to Fulfill</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group I</td>
<td>P1–P3 with PDL, then P4–P6 with C</td>
</tr>
<tr>
<td>Group II</td>
<td>P4–P6 with C, then P1–P3 with PDL</td>
</tr>
<tr>
<td>Group III</td>
<td>P4–P6 with PDL, then P1–P3 with C</td>
</tr>
<tr>
<td>Group IV</td>
<td>P1–P3 with PDL, then P4–P6 with PDL</td>
</tr>
</tbody>
</table>

### Table 4: Students’ status for solving the six COPs

<table>
<thead>
<tr>
<th>COP</th>
<th>With PDL</th>
<th>Without PDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>P2</td>
<td>1.2</td>
<td>0.0</td>
</tr>
<tr>
<td>P3</td>
<td>2.1</td>
<td>0.0</td>
</tr>
<tr>
<td>P4</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>P5</td>
<td>1.0</td>
<td>0.5</td>
</tr>
<tr>
<td>P6</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

4.2.2 Experiment Results. Figure 4 presents students’ responses in the pre-study form. According to Figure 4, 53.5% of students were unconfident to solve COPs, while only 11.4% of students had the confidence. The lack of confidence in many students reflects the difficulty of solving COPs. Table 4 presents students’ problem-solving status in our study. Quit Rate shows the percentages of students who gave up a COP without submitting any artifact. Avg. Solving Time describes the average problem-solving time for each COP. Error Rate shows among the latest submissions by students who did not quit, what percentage of artifacts are incorrect.

According to Table 4, students were less likely to quit when using PDL. In particular, for the most complex two problems P3 and P6, only 2.3% and 1.0% of students quit while using PDL; however, 18.6% and 23.9% of students gave up either task while coding in C. This implies that with PDL, students were more encouraged to solve COPs. Additionally, students with PDL usually spent less time than students without it. We further conducted Mann-Whitney U test to check whether the solving time is significantly different between the two approaches. For the solving time of P1 and P4, we observed no significant difference between the students who used PDL and those who did not use it. However, when solving other problems, students with PDL did spend significantly less time than those without (p=1e-5). The major reason is that P1 and P4 are much easier than other problems. As the problem complexity increases, the solving-time gap between the two approaches increases as well. Finally, there are fewer errors in submitted PDL descriptions than in C code. This is because when using PDL, students did not need to design or implement any algorithm. The tool usage eliminates the opportunity for students to commit coding errors.

Finding 2: PDL effectively encouraged students to solve COPs instead of giving up; it also helped students successfully solve more problems with less time spent.

Figure 5 presents students’ perception of the difficulty in writing PDL descriptions or C code. Interestingly, writing PDL descriptions seems to be easy for 46.5% of students, and seems hard for only 20.0% of students. On the other hand, writing C code seems easy for only 33.0% of students, but hard for 36.2% of students. According to Mann-Whitney U test, students perceived writing PDL descriptions to be significantly easier than writing C code (U=12574, p=1e-5). We observed similar contrasts in Figure 6. When debugging PDL
descriptions. 46.5% of students found it easy and 20.0% of students found it hard. However, only 11.4% of students considered it easy to debug C code but 69.2% of students considered it hard. Our Mann-Whitney U test shows that the students who debugged PDL descriptions sensed significantly less difficulty than those who debugged C code (U=8111.5, p<1e-5).

These observations help explain the above-mentioned phenomenon that students with PDL could solve more COPs with less time spent. PDL's effectiveness reduced the complexity of solving COPs by generating solution code to COPs. When students focused their efforts on problem comprehension and description, the feedback PDL provides can reveal flaws in students' descriptions, imply the relationship between problem characteristics and solution algorithms, and equip students with experience of solving COPs.

**Finding 3:** Compared with C coding or debugging, more students found it easier to write or debug PDL descriptions. The observations help explain why PDL encouraged students to solve COPs.

Figure 7 illustrates students' confidence levels after they solved three COPs in Phase I with or without PDL. By comparing this figure against Figure 4, we observed two interesting phenomena. First, after solving COPs without PDL, fewer students were neutral (29.0% vs. 35.1%). Some originally neutral students became either more or less confident in solving COPs, probably due to their positive or negative coding experience with the problems. In comparison, after solving COPs with PDL, a lot more students reported confidence in handling such problems (41.2% vs. 11.4%), and a lot fewer students claimed lacking confidence (21.1% vs. 53.5%). The Mann-Whitney U test shows that the confidence growth in students with PDL is significant (U=1913, p<1e-5), while the growth for students without PDL is not significant (U=4940.5, p=0.44433>0.05). With the positive problem-solving experience and PDL's constructive feedback, students became more optimistic in taking challenges.

**Finding 4:** The experience of using PDL considerably increased students' confidence in solving COPs, probably because (1) the experience is more positive and (2) the feedback is more detailed.

5 THREATS TO VALIDITY

In our user study, students' self reports may be subjective to human bias. To mitigate the problem, we conducted the user study with a large number of students (e.g. 185); during the study, we answered all students' questions to clarify expectations and reduce bias. To measure PDL's effectiveness in helping students improve PSS, we compared the quit rates, problem-solving time, error rates, and confidence levels between students with PDL and students coding in C. However, we did not measure the improvement in students' problem-solving capabilities, which we plan to explore in the future.

Although it seems unsurprising that describing problem is always easier than coding the solution, we could not assume PDL to be easier to use than C. Thus, we compared the data collected from students with PDL and students with C. The comparison indicates two things. First, PDL is usually easier to use, so students can turn to PDL when they are unable to code C solutions directly. Second, students had their confidence levels significantly increase after using PDL, so PDL actually reduced the technical barrier for students to code in C and can help retain students in the CS major.

6 CONCLUSION

When students try to improve PSS through programming, paradoxically, they can only benefit from the coding experience if they are able to understand problems well, quickly develop promising solutions, and successfully digest and resolve the coding issues encountered. To lower the technical barriers for students to better PSS through coding, we introduced PDL—a scaffolding approach that enable students to work on challenging COPs. Our evaluation shows that PDL effectively reduced the complexity of solving COPs and better motivated students to improve PSS via solving COPs. In the future, we will conduct larger-scale studies to explore how PDL helps different kinds of novice developers (e.g., K-12 students), and improve PDL by generating more optimization strategies.

ACKNOWLEDGMENTS

This work was supported by Science and Technology Innovation 2030 No.2018AAA0100901 of China and NSF-1845446 of US. We thank reviewers and Dr. Yingfei Xiong for their valuable feedback.
REFERENCES


