PDL: Scaffolding Problem Solving in Programming Courses

Shu Lin* Na Meng† Dennis Kafura† Wenxin Li*
Peking University* Virginia Tech†
China* USA†
fzlinshu@pku.edu.cn,nm8247@cs.vt.edu,kafura@cs.vt.edu,lwx@pku.edu.cn

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*.

CCS CONCEPTS

• **Social and professional topics → Computational science and engineering education**.

KEYWORDS

problem-solving skill; problem description; problem understanding; program generation; student assessment

ACM Reference Format:
Shu Lin, Na Meng, Dennis Kafura, and Wenxin Li. 2021. PDL: Scaffolding Problem Solving in Programming Courses. In *26th ACM Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2021), June 26-July 1, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3430665.3456360

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 help 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 $\mathcal{A} = \{\alpha_1, \cdots, \alpha_M\}$ and their value ranges, (2) variables $\mathcal{V} = \{v_1, \cdots, v_N\}$, value ranges, and their value relations with $\mathcal{A}$: $\mathcal{R} = \{r_1(\mathcal{A}, \mathcal{V}), \cdots, r_L(\mathcal{A}, \mathcal{V})\}$, and (3) an objective function $f(\mathcal{A}, \mathcal{V})$. The program will find the optimal value assignments for $\{v_1, \cdots, v_N\}$ such that (i) $\mathcal{R}$ is satisfied, and (ii) the value of $f(\mathcal{A}, \mathcal{V})$ 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

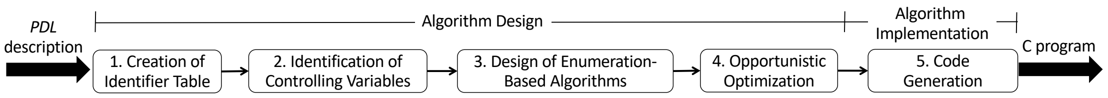
PDL description
Algorithm Design
Algorithm Implementation
1. Creation of Identifier Table
2. Identification of Controlling Variables
3. Design of Enumeration-Based Algorithms
4. Opportunistic Optimization
5. Code Generation
C program

Figure 1: The *PDL* compiler takes in any *PDL* description, goes through five phases, and generates a solution program in C

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., COPs) 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, Foshay and Kirkley 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., $\mathcal{A}$).
- **Required Section** declares variables (i.e., $\mathcal{V}$), value ranges, and the mathematical formulas that define relations or constraints (i.e., $\mathcal{R}$) between variables and arguments.
- **Objective Section** defines the objective function (i.e., $f(\mathcal{A}, \mathcal{V})$).
- **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*", ">=", and "^"), (2) two logical quantifiers: universal quantifier `forall` (i.e., ∀) and existential quantifier `exists` (i.e., ∃), (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.

Natural Language	**PDL**
	`#input`
Given the capacity C.	`    C of int in [1,1000];`
Given a set of N items,	`    N of int in [1,100];`
each with a weight w_i and	`    w of (int in [1,C])[1..N];`
a value v_i.	`    v of (int in [1,100])[1..N];`
Find a subset of items	`#required`
such that the weight sum	`    S <= (int in [1..N]) {};`
is no more than C,	`   summation [w[x] : forall x (x in S)] <= C;`
	`#objective`
while the value sum is	`    maximize`
maximized.	`      summation [v[x] : forall x (x in S)];`
	`#output`
Print the selected items.	`    S;`

Figure 2: 0/1 knapsack problem described in two ways

Please refer to *PDL*'s online manual [21] for more information of the syntax. Essentially, *PDL* is a declarative instead of imperative language; so with *PDL*, users specify the problem to solve rather than how to solve it. For example, Figure 2 presents both natural-language description and *PDL* description of the 0/1 knapsack problem. The input section declares two integer arguments: C and N, two array arguments w and v, and their value ranges. The required section declares a set variable S to record items chosen to fill the knapsack; it also defines a constraint on the weight sum of selected items. The objective section defines the optimization goal, while the output section declares the variable S to print.

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).

Algorithm 1: An algorithm skeleton for iterative search

```
Input: $\mathcal{A}$ /* input parameters */
Output: $best\_result$ and the value of variables in $\mathcal{V}$
1.1 $best\_result \leftarrow$ null;
    /* suppose the controlling variables are $\mathcal{I} = \{i_1, \cdots, i_k\}$ */
1.2 foreach $i_1 \in i_1$'s range do
        ⋱
1.3     foreach $i_k \in i_k$'s range do
1.4         calculate the values of variables in $\mathcal{V} \backslash \mathcal{I}$;
1.5         if $\mathcal{R}$ is satisfied $\wedge f(\mathcal{A}, \mathcal{V})$ is better than $best\_result$ then
1.6             update $best\_result$ and record the variables' value;
1.7 return $best\_result$ and the variables' value;
```

Algorithm 2: An algorithm skeleton for recursive search

```
Input: $\mathcal{A}$ /* input parameters */
Output: $best\_result$ and the value of variables in $\mathcal{V}$
2.1 $best\_result \leftarrow$ null;
2.2 recursive_enum_$v$(1);
2.3 function recursive_enum_$v$($step$)
2.4 if $step > |\mathcal{I}|$) then
2.5     calculate the values of variables in $\mathcal{V} \backslash \mathcal{I}$
2.6     if $\mathcal{R}$ is satisfied $\wedge f(\mathcal{A}, \mathcal{V})$ is better than $best\_result$ then
2.7         update $best\_result$ and record the variables' value;
2.8 else
2.9     foreach $v_{step} \in v_{step}$'s range do
2.10        recursive_enum_$v$($step$+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 $\mathcal{VS} \subseteq \mathcal{V}$ such that:

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

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 $S_1 \subset S_2$), 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 naïvely 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 $step_i (i \in [1, N])$, *PDL* adds an `if`-check to decide whether the weight sum so far $sumW_i$ exceeds C, i.e., $sumW_i > C$. This is because all items have positive weights (i.e., $w[j] \geq 1 (j \in [1, N])$). If $sumW_i$ exceeds C, even though none of the last $(N - step_i + 1)$ 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 solve 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:

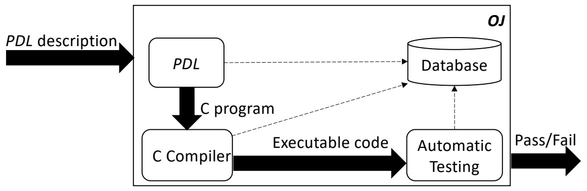

Figure 3: The online judge (OJ) system we built to evaluate the quality of *PDL* descriptions

- **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*.

Table 1: *PDL*'s usability evaluation based on 45 COPs

Algorithm	Total	Inexpressible	Partly Solvable	Solvable
Enumeration	17	2 (11.8%)	0 (0.0%)	15 (88.2%)
With Pruning	16	3 (18.8%)	3 (18.8%)	10 (62.5%)
DP & Pruning	12	2 (16.7%)	3 (25.0%)	7 (58.3%)
Total	45	7 (15.6%)	6 (13.3%)	32 (71.1%)

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.*

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 I, they solved three COPs either with or without *PDL*; then in Phase

Table 2: The six problems used in the second experiment

ID	Name	Algorithm	Description
P1	Duplicate Number	Enumeration	Given $N(1 \le N \le 1000)$ integers, find the only one duplicate number.
P2	Eight Queens	With Pruning	Place eight chess queens on the chessboard such that none of them is able to capture the others.
P3	Shortest Path	DP & Pruning	There are $N(1 \le N \le 100)$ cities. Given the distance between each pair of cities, find the shortest path from City 1 to City N.
P4	Sum Is K	Enumeration	Find two integers among $N(1 \le N \le 1000)$ given integers, such that the sum of them is equal to K.
P5	Messager Problem	With Pruning	There are $N(1 \le N \le 10)$ cities. Given the distance between each pair of cities, find the shortest route that visits each city exactly once without the need of returning to the start city.
P6	Teamwork	DP & Pruning	There are $N(1 \le N \le 100)$ candidates. Each candidate has a cooperation value v $(-50 \le v \le 50)$ and a working value w $(-50 \le w \le 50)$. Select any number of candidates to form a team, such that the summation of all the team members' cooperation values is positive and the summation of their working value is maximum.

Table 3: The tasks assigned to each group

Group ID	Designated Tasks to Fulfill
Group I	First P1–P3 with *PDL*, then P4–P6 with C
Group II	First P4–P6 with C, then P1–P3 with *PDL*
Group III	First P4–P6 with *PDL*, then P1–P3 with C
Group IV	First P1–P3 with C, then P4–P6 with *PDL*

Table 4: Students' status for solving the six COPs

	With *PDL*						Without *PDL*					
	P1	**P2**	**P3**	**P4**	**P5**	**P6**	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**
Quit Rate (%)	0.0	1.1	2.3	0.0	2.1	1.0	1.0	5.2	18.6	0.0	4.5	23.9
Avg. Solving Time (min)	5.5	8.3	9.2	5.0	8.1	6.4	5.9	14.6	22.6	7.1	15.4	26.7
Error Rate (%)	0.0	3.4	2.3	0.0	2.1	0.0	2.1	9.8	17.7	0.0	11.9	17.9

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.

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 hard, 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*

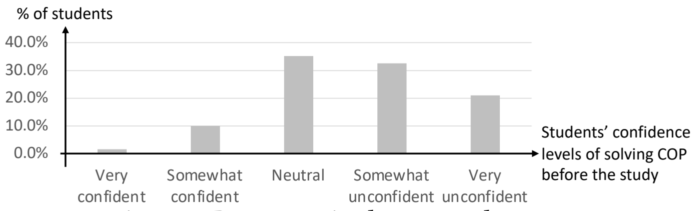

Figure 4: Responses in the pre-study survey

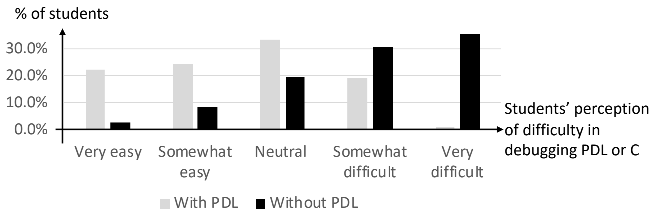

Figure 6: Responses to Q3

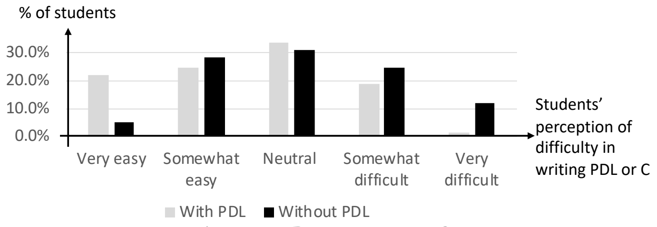

Figure 5: Responses to Q2

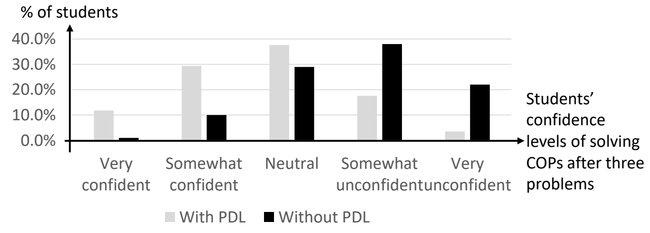

Figure 7: Responses to Q4

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* effectively reduced the complexity of solving COPs by generating solution code to COPs. When stuents 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=2193, 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

[1] I.E. Allen and C.A. Seaman. 2007. Likert scales and data analyses. *Quality Progress* 40 (07 2007), 64–65.

[2] A. Bart, J. Tibau, E. Tilevich, C. A. Shaffer, and D. Kafura. 2017. BlockPy: An Open Access Data-Science Environment for Introductory Programmers. *Computer* 50, 05 (may 2017), 18–26. https://doi.org/10.1109/MC.2017.132

[3] Theresa Beaubouef and John Mason. 2005. Why the High Attrition Rate for Computer Science Students: Some Thoughts and Observations. *SIGCSE Bull.* 37, 2 (June 2005), 103–106. https://doi.org/10.1145/1083431.1083474

[4] Pietro Belotti, Sonia Cafieri, Jon Lee, and Leo Liberti. 2010. Feasibility-based bounds tightening via fixed points. In *International Conference on Combinatorial Optimization and Applications*. Springer, 65–76.

[5] Matthew Butler and Michael Morgan. 2007. Learning challenges faced by novice programming students studying high level and low feedback concepts. In *Proceedings of ASCILITE - Australian Society for Computers in Learning in Tertiary Education Annual Conference 2007*. Australasian Society for Computers in Learning in Tertiary Education, 99–107. https://www.learntechlib.org/p/46043

[6] William J. Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver. 1998. *Combinatorial Optimization*. John Wiley & Sons, Inc., New York, NY, USA.

[7] Michael De Raadt. 2008. *Teaching programming strategies explicitly to novice programmers*. Ph.D. Dissertation.

[8] Christian DeLozier, Richard Eisenberg, Santosh Nagarakatte, Peter-Michael Osera, Milo Martin, and S. Zdancewic. 2013. Ironclad C++ A Library-Augmented Type-Safe Subset of C++. *ACM SIGPLAN Notices* 48, 287–304. https://doi.org/10.1145/2509136.2509550

[9] Hammond Ed. 2001. Scaffolding: Teaching and Learning in Language and Literacy Education. (01 2001).

[10] Francisco Buitrago Flórez, Rubby Casallas, Marcela Hernández, Alejandro Reyes, Silvia Restrepo, and Giovanna Danies. 2017. Changing a Generation's Way of Thinking: Teaching Computational Thinking Through Programming. *Review of Educational Research* 87, 4 (2017), 834–860. https://doi.org/10.3102/0034654317710096 arXiv:https://doi.org/10.3102/0034654317710096

[11] Wellesley Foshay and Jamie Kirkley. 1998. *Principles for Teaching Problem Solving*.

[12] Felienne Hermans and Efthimia Aivaloglou. 2017. To Scratch or Not to Scratch? A Controlled Experiment Comparing Plugged First and Unplugged First Programming Lessons. In *Proceedings of the 12th Workshop on Primary and Secondary Computing Education* (Nijmegen, Netherlands) *(WiPSCE '17)*. Association for Computing Machinery, New York, NY, USA, 49–56. https://doi.org/10.1145/3137065.3137072

[13] Mohd Ismail, Nor Ngah, and Irfan Umar. 2010. Instructional strategy in the teaching of computer programming: A need assessment analyses. *The Turkish Online J Edu Technol* 9 (04 2010).

[14] Mohd Nasir Ismail, Nor Azilah Binti Ngah, and Irfan Naufal Umar. 2010. INSTRUCTIONAL STRATEGY IN THE TEACHING OF COMPUTER PROGRAMMING: A NEED ASSESSMENT ANALYSES.

[15] Andrew J. Ko, Thomas D. LaToza, Stephen Hull, Ellen A. Ko, William Kwok, Jane Quichocho, Harshitha Akkaraju, and Rishin Pandit. 2019. Teaching Explicit Programming Strategies to Adolescents. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 7 pages. https://doi.org/10.1145/3287324.3287371

[16] Thomas D. LaToza, Maryam Arab, Dastyni Loksa, and Amy J. Ko. 2020. Explicit Programming Strategies. *Empirical Software Engineering* (2020), 1–34.

[17] Irene Lee, Fred Martin, Jill Denner, Bob Coulter, Walter Allan, Jeri Erickson, Joyce Malyn-Smith, and Linda Werner. 2011. Computational thinking for youth in practice. *ACM Inroads* 2 (02 2011), 32–37. https://doi.org/10.1145/1929887.1929902

[18] Dastyni Loksa, Andrew J. Ko, Will Jernigan, Alannah Oleson, and Margaret M. Burnett. 2016. Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. In *the 2016 CHI Conference*.

[19] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Trans. Comput. Educ.* 10, 4, Article 16 (Nov. 2010), 15 pages. https://doi.org/10.1145/1868358.1868363

[20] MIT App Inventor 2020. MIT App Inventor. https://appinventor.mit.edu.

[21] PDLManual 2020. PDL Manual. Retrieved January 7, 2020 from https://github.com/pdlteam/pdl-manual/blob/master/pdlmanual.pdf

[22] Frank Pfenning. 2010. C0 Reference. https://www.cs.cmu.edu/~fp/courses/15122-f10/misc/c0-reference.pdf.

[23] Alex Ruthmann, Jesse M. Heines, Gena R. Greher, Paul Laidler, and Charles Saulters. 2010. Teaching Computational Thinking through Musical Live Coding in Scratch. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (Milwaukee, Wisconsin, USA) *(SIGCSE '10)*. Association for Computing Machinery, New York, NY, USA, 351–355. https://doi.org/10.1145/1734263.1734384

[24] Shima Salehi, Karen D. Wang, Ruqayya Toorawa, and Carl Wieman. 2020. Can Majoring in Computer Science Improve General Problem-Solving Skills?. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 156–161. https://doi.org/10.1145/3328778.3366808

[25] Jho-Ju Tu and John R. Johnson. 1990. Can Computer Programming Improve Problem-Solving Ability? *SIGCSE Bull.* 22, 2 (June 1990), 30–33. https://doi.org/10.1145/126445.126451