Mapping the Problem-Solving Strategies of Novice Programmers to Polya's Framework: SWOT Analysis as a Bottleneck Identification Tool

Pakiso J. Khomokhoana and Liezel Nel(✉)

Department of Computer Science and Informatics, University of the Free State, Bloemfontein, South Africa
{khomokhoanap,nell}@ufs.ac.za

Abstract. The development of problem-solving skills continues to be a challenge in various disciplines including Computer Science. In this study, we used the principles of the Decoding the Disciplines (DtDs) paradigm to better understand the mental processes that novice programmers follow when answering source code comprehension (SCC) related questions. This understanding can be fundamental in helping novices to overcome problem-solving related challenges. While focusing on step 1 of the DtDs paradigm, the aim of this study was threefold. Firstly, we explored the problem-solving strategies utilised by novice programmers while they were attempting to answer SCC related questions. Secondly, the identified problem-solving strategies were mapped onto Polya's four problem-solving steps. Finally, we utilised a SWOT analysis as a tool to identify problem-solving related learning bottlenecks. This study utilised an integrated methodological approach where data was collected by means of asking questions, observations, and artefact analysis. Thematic analysis of the collected data revealed a range of problem-solving strategies that these novice programmers utilised while performing various SCC tasks. These strategies were then mapped onto Polya's problem-solving steps. Based on a SWOT analysis of these strategies, we identified six problem-solving bottlenecks that point to difficulties that are not sufficiently addressed in introductory CS courses.

Keywords: Decoding the disciplines · Problem-solving · Source code comprehension · Novice programmers · Computer Science education · Polya's framework · SWOT analysis

1 Introduction

High dropout rates in introductory computer programming courses remain a major concern for higher education educators across the world [1, 20, 22]. Computer programming is a complex and multi-faceted task as it requires not only conceptual and procedural knowledge but also skills to create, modify and comprehend computer code [13, 35, 38] in order to solve programming problems. Numerous studies [2, 19, 34] have identified a

© Springer Nature Switzerland AG 2022
W. S. Leung et al. (Eds.): SACLA 2021, CCIS 1461, pp. 132–148, 2022.
https://doi.org/10.1007/978-3-030-95003-3_9

lack of problem-solving skills as one of the biggest challenges that novice programmers experienced. Another major challenge (which has been researched extensively) relates to how novice programmers comprehend (or interpret) pieces of source code [10, 18]. Lister et al. [18] regard the skill of source code comprehension (SCC) as a prerequisite to problem solving. While Conn and McLean [9] argue that novice programmers' lack of problem-solving skills stems from how problem solving is taught in schools, Belski [3] blames university educators for failing to properly develop this skill in students. This failure has been attributed to educators' tendency to focus on the teaching of syntactical and conceptual programming knowledge, and paying less (or no) attention to the strategic knowledge needed to appropriately and effectively solve programming problems [7, 19, 41]. This tendency is not exclusive to the Computer Science (CS) discipline. Middendorf and Shopkow [24] (p. 2) note that while educators (as practitioners) are familiar with the "approaches, techniques, and applications" of their specific academic disciplines, "they tend to organise their courses around specific contents rather than around the mental moves they want students to make".

The Decoding the Disciplines (DtDs) paradigm – formulated by Middendorf and Pace [23] – recognises that each discipline has its own unique ways of thinking that students need to master to succeed in their higher-level studies. As part of the initial step in the seven-step DtDs process, educators are encouraged to identify specific points where their students' learning are interrupted [11]. These points (referred to as bottlenecks) are likely to prevent students from mastering the basic disciplinary ways of thinking. Only once these bottlenecks have been identified can educators continue with the remaining DtDs steps of designing, implementing, and evaluating specific strategies to address the bottlenecks that their students are experiencing [26]. There are several examples of approaches that educators used to identify bottlenecks. These include educators' personal experiences in teaching a specific course [24, 28], problems discovered while grading student assignments [26], problems observed while watching students' attempts to solve a given problem [17], and clarification questions asked by students regarding assignment specifications [39]. A tool that could potentially also be used in this regard is a SWOT analysis [16]. Although mostly used in business environments, this strategic planning tool can help to analyse and position any situation in four regions: strengths, weaknesses, opportunities, and threats [36]. By analysing the mental moves novice programmers make while solving SCC problems, a SWOT analysis could potentially be used to, not only identify the problem-solving steps that students are performing well, but also the areas (potential bottlenecks) where students are not following the correct problem-solving steps.

The foundation of most modern problem-solving strategies (both general and in the field of CS) can be traced back to George Polya's [30] four basic problem-solving steps: understand the problem, devise a plan, carry out the plan, and evaluate the effectiveness of the plan. Since problem solving is a skill that is used daily by all human beings, it can be argued that the natural (unenhanced) problem-solving strategies followed by novice programmers should in theory show some resemblance to the original steps of Polya's framework. By mapping the problem-solving strategies followed by novice programmers while trying to answer an SCC question, it should be possible to identify shortcomings

in their strategies. This, in turn, could point to potential learning bottlenecks that CS educators must specifically address.

This paper therefore attempts to answer the following three questions:

1. What are the problem-solving strategies utilised by novice programmers during SCC?
2. How do these strategies relate to Polya's four basic problem-solving steps?
3. How can a SWOT analysis of the SCC problem-solving strategies followed by novice programmers be used as a learning bottleneck identification tool?

In the remainder of this paper, a theoretical framework guiding this study together with a review of relevant background literature are presented in Sect. 2. This is followed by a discussion of the research design and methods in Sect. 3, and a presentation and interpretation of the results in Sect. 4. A SWOT analysis of the identified SCC strategies is presented in Sect. 5, followed by bottleneck identification in Sect. 6. Conclusions and recommendations for future research are presented in Sect. 7.

2 Theoretical Framework

Polya's [30] problem-solving framework provides the theoretical framework for this study. In the following sub-sections, the four basic steps of Polya's framework, together with examples of how these steps are typically executed by computer programmers during SCC tasks, are discussed in more detail.

2.1 Understand the Problem

It is impossible to solve a problem if the problem is not understood first. Polya [30] provides several questions that can be asked by the problem solver while trying to understand or comprehend a problem: Do you understand all the words used in the problem statement? What are you asked to find or show? Can you restate the problem in your own words? Can you think of a picture or diagram that might help you understand the problem? Is there enough information to enable you to find a solution?

As part of the understanding process, Chi et al. [8] suggest organising information around important concepts. As part of understanding a given problem, programmers typically apply strategies that include reading or re-reading problem requirements and/or related lines of code, and reasoning aloud [14, 25]. Other strategies such as highlighting or colouring some lines of code or text [31], writing comments [37], and making drawings or annotations (doodles) [18] are often utilised by programmers to gain a better understanding of the problem at hand. For SCC tasks, expert programmers will often scan the code from top to bottom to get a quick overview of the problem [40], while also making notes on important information. They can then easily refer to these notes (if needed) at a later stage [33].

2.2 Devise a Plan

Polya [30] suggests several of strategies for devising a problem-solving plan. These include guess and check, look for a pattern, make an orderly list, draw a picture, eliminate possibilities, solve a simpler problem, use symmetry, use a model, consider special cases, work backwards, use direct reasoning, use a formula, solve an equation, and be ingenious. In resonance with Polya [30], the Mathematics students in the study by Malloy and Jones [21] exhibited planning strategies that included drawing a picture or diagram, using patterns, making lists or charts, guessing and checking, working backward, using logical deduction, disregarding extra (unnecessary) data, as well as using multiple or a combination of strategies and logical deduction.

The problem-solving plan helps to arrange ideas together, which in turn helps a problem solver to gain an even better understanding of the problem [5]. With regard to SCC, Fitzgerald et al.'s [14] respondents recognised that one SCC question resembled other questions seen previously (pattern recognition) and, therefore, made assumptions based on previous answers. During the planning stage of SCC, programmers tend to identify possible test cases [25] to use in the execution of their plan. This helps them to avoid carrying out infinite tests. Similar to understanding a problem, programmers also make notes (annotations/doodles) on how their plan will be executed [18]. Ultimately, devising a plan will typically involve working through different scenarios and gauging the good as well as the bad points of each alternative [33]. During this process, a problem solver justifies all the decisions he/she arrives at [15]. This also enables problem solvers to remember important information that may be useful during the remainder of the problem-solving process [5].

2.3 Carry Out the Plan

When it comes to carrying out the plan, Polya [30] suggests that the chosen plan should be followed. If the plan does not work, it should be discarded, and another alternative plan should be selected. The decision to either discard or continue with the selected plan can only be made if the problem solver is continuously monitoring the plan – a task that requires metacognitive abilities [5]. Inherently, a lot of thinking and analysing are in operation during this problem-solving step [4]. In addition to applying surface thinking, the problem solver also needs to think critically and creatively while integrating prior knowledge as part of the process [12]. In carrying out a plan, the problem solver usually considers limited details that are only relevant to the selected plan [33]. Interestingly, Fitzgerald et al. [14] note that some of their participants would start from scratch with an SCC task whenever they were becoming confused.

2.4 Evaluate the Effectiveness of the Plan

To evaluate the effectiveness of the selected plan, Polya [30] suggests that the problem solver should take the time to reflect and look back at what has been done, what worked, and what did not work. This step emphasises the importance of assessing and possibly even re-assessing the identified solution to any programming or SCC related problem [12]. This will typically be an iterative process during which the programmer may

discover even more knowledge about the problem and/or solution [19]. It is also not uncommon for programmers to document their reflections on a completed SCC task [37].

3 Research Methods

3.1 Design

The design of this study was narrative in nature and followed an integrated-methods research approach based on Plowright's [29] Frameworks for an Integrated Methodology (FraIM). Within this framework, both narrative and numeric data were collected by means of observations, asking questions, and artefact analysis. The study population consisted of final-year undergraduate CS students from a selected South African university. After participating in an SCC activity (as part of an earlier phase of this multi-phase research project), students who incorrectly answered the three most difficult questions (as determined during the earlier phase) were invited to take part in this phase of the study. The purposeful and convenient sample [27] consisted of the 10 students who agreed to partake in this part of the study. The purposefulness of the sample was based on the fact that the students had already completed four programming modules. However, they could still be regarded as novice programmers since they did not have any professional programming experience. Based on their underperformance in the mentioned SCC activity, it was highly likely that they were experiencing some bottlenecks in their learning. The sample was also convenient since the students were affiliated with the same department as the researchers.

3.2 Data Collection

The research activity consisted of individual sessions during which each participant had to use a think-aloud technique [32] to verbally explain his/her thinking processes while answering three selected SCC questions. These questions – C# versions of question 3 (Q3), question 6 (Q6) (see Appendix), and question 8 (Q8) from Lister et al.'s [18] study – were the same three questions that the selected participants were unable to answer correctly during an earlier phase of this research project. This data collection strategy can be regarded as a means of "asking questions" [29]. Time slots of 45 min were scheduled for each of the individual sessions. The first author (principal researcher) played the role of the interviewer by asking probing questions when required (i.e. in case of no progress or silence). Where deemed necessary, the interviewer also recorded some observations as an additional data collection strategy. The proceedings of each session were audio recorded with permission from the relevant participant.

3.3 Data Analysis

Since the participants had to verbalise their thoughts as part of the think-aloud process, the transcripts contained numerous illogical and repeated statements. We therefore cleansed the data, after which we familiarised ourselves with the data [6]. This was achieved

by listening to the audio records numerous times as well as intensively and repeatedly reading the transcripts. This helped us to decide on a coding plan where the analysis would be guided by the data as it relates to our research questions. At this stage, we imported the 10 validated transcripts into NVivo 12 Professional for Microsoft Windows. After this, we developed the necessary codes (by creating several nodes) according to our research questions. We coded the data by highlighting and/or underlining text within the domain of the stated units of analysis. We then populated the created codes by moving the necessary text into them. We continuously revised the names of the codes and the relevant themes until recurrent themes were emerging. For each theme developed, the NVivo-generated frequencies of occurrence were used.

4 Results and Interpretation

The discussion in this section focuses on the problem-solving strategies identified from the data collected during the 10 think-aloud sessions. The discussion is grouped according to Polya's four problem-solving steps (see Sect. 2).

4.1 Understand the Problem

Analysis of both the think-aloud data and the interviewer's observations revealed that the participants utilised two main strategies to help them understand the problem.

Read Instructions and Identify Important Concepts or Terms. The typical first step in understanding the problem is to read the instructions or the problem statement [30]. This strategy was utilised by all the participants with a total of 35 occurrences observed. It is also key that the problem solver, while reading through the instructions, identifies important concepts and/or terms [8]. In the 75 occurrences of this strategy, the participants mentioned and/or underlined specific programming concepts and terms that they would have to understand in order to answer the given questions. These included Boolean values, parameters, initialisation, nested loops, arrays, array indexes, and the use of post- and pre-increment operators. By identifying all these concepts, participants were trying to form an initial understanding of crucial parts of the provided programming code.

Interpret the Problem Details. As part of understanding the problem [30], all the participants (29 occurrences) also tried to interpret more specific details from the provided SCC questions. As part of this interpretation process, they conducted a more thorough examination of specific pieces of source code and tried to explain the meaning thereof in an attempt to increase their level of understanding. At this stage, it also became apparent that the flawed interpretations made by some of the participants were likely to have a negative impact on their ability to ultimately solve the problem. Due to his misinterpretation of the combined index in the statement `b[x[i]]= true;` from Q3, P10 incorrectly concluded that "*the second `for` loop resets everything from the first `for` loop back to `true`*". By failing to recognise from the start that only the first three values in array `b` would be set to `true`, he was eventually unable to identify option B as the correct answer.

4.2 Devise a Plan

It was noted that none of the participants in this study attempted to formulate a definitive plan for solving the various SCC questions. They did, however, follow some of the strategies outlined in Sect. 2.2.

Eliminate Possibilities. After a quick look at the source code of Q6, P1 eliminated option C as the possible missing piece of source code. She said: "*Let me understand option C… if the index of an array at position* `i` *is bigger than the index of* `i + 1`*, I must return* `false`*, else return* `b` *which is already* `false`*. There is no way of returning* `true`*, so it is totally wrong, it can't be the missing source code*". This meant that she was left with only four options to consider, which simplified the remainder of the problem-solving process for her.

Make Notes. As part of devising a plan during the problem-solving process [30], problem solvers typically make various types of notes while they work through the problem [31, 37]. Seven participants utilised this strategy (with 24 occurrences). As part of his attempt to answer Q3, P5 made notes on the answer sheet to keep track of the values assigned to array `b`. While working through the second `for` loop he said: "*Set that value to* `true`*. So that is going to mean that* `1`*,* `2` *and* `3` *is going to be* `true`". He then scratched out the three relevant `false` values in the notes he originally made when he was working through the first `for` loop. Although he did not make any further written notes, he continued to reference the written down values while he was working through the third `for` loop to determine the final value of the variable `count`. It should, however, be noted that note-making instances were also observed with the participants during the three other steps of the problem-solving process.

Use Test Cases. In devising a plan during problem solving, a problem solver can identify some test cases [25] to avoid conducting infinite tests. Four occurrences of this strategy were observed with three of the participants. As evidence that P6 used test cases, she said: "*Let me just eliminate this one for now, so that I don't waste time on irrelevant things … [Interviewer: You have created a long array!] … yeah it was quite long*". In formulating test cases for Q3, P2 decided to draw a trace table as part of his planning and reasoning. He later referred to this table when he had to determine his final answer, where he said: "*Since* `i` *is equal to* `4`*, our value is* `5`*. So, our answer here should be* `5` *according to this table*". P10 also created his own array and started adding values as he went through his reasoning for Q6: "*Well, I made my array and I just changed the values so that I can better understand what I was sorting out*". It is evident that these participants not only identified possible test cases, but specifically used these values during the execution of their problem-solving plans.

Work Backwards. While devising a plan, problem solvers can also work backwards to check or confirm some of their initial interpretations [30], and then take corrective actions if needed. Ten occurrences of this strategy were identified with six of the participants. While attempting to interpret the second `for` loop from Q3, P6 exhibited several signs of confirming initial understanding: "*… meaning* `zero` *is less than* `5`*, yeah* `5`*, wait… `5` … yeah, it's* `5`*… I think my answer is going to be B, wait… `x[i]` is* `1`*, then* `2`*… okay*

wait that's `zero`*, that's* `1`*. So now let's see,* `i` *is* `0` *... eish, what do we call this step? Initialising? Yeah, I think so ... [Interviewer: Which one?] ... the one with* `b[x[i]]`*... [Interviewer: You are assigning another value.], Yeah, assigning it to* `true`". It is evident from this excerpt that while P6 attempted to confirm his initial understanding, he also resorted to seeking confirmation or clarification from the interviewer. He specifically wanted to confirm whether he was using the correct terminology while attempting to interpret the statement `b[x[i]]= true;` which is an example of an assignment and not an initialisation statement. It can therefore be deduced that although he was attempting to link whatever he was doing to known disciplinary concepts [8], there were still some gaps in his knowledge of basic programming terminology.

Pattern Recognition. Another strategy that problem solvers often utilise during the devising-a-plan step is pattern recognition. During SCC, this happens when the problem solver recognises similarities between different pieces of source code that allow for code sections to be considered collectively [14]. The pattern recognition strategy was observed with eight participants (20 occurrences). While looking at the initialisation of array `x` in Q3, P8 said: "*I am looking at the pattern for this, what was declared of* `x` *... so now by judging from the pattern that I see here, it's* `1, 2` *and then it becomes constant*". By "constant", P8 was referring to the value of `3` that was assigned to the last three elements of the `x` array. P6 also noticed a pattern in the headers of the first two `for` loops in Q3: "*The second* `for` *loop is still the same as the first one, but instead of taking the* `b` *length we take the* `x` *length, so all these are the same*". While P2 recognised the same pattern as P6, he also noted that there were actually similarities between the headers of all three `for` loops in Q3: "*All these loop statements look a little bit the same because all of them start at* `i = 0` *... and* `i++`*, so they are all correct here"*. Identification of a pattern or range of patterns [21, 30] helps to reduce cognitive load as the problem solver is left with fewer unique possibilities to concentrate on [5].

4.3 Carry Out the Plan

In carrying out their plans, participants corrected their earlier misinterpretations and acknowledged areas where they experienced difficulties.

Correct Earlier Misinterpretations. In carrying out a plan, problem solvers can correct the misinterpretations they might have made in the previous problem-solving steps. This strategy was observed with three participants (five occurrences). After reviewing the pencil notes she made earlier in the problem-solving process, P9 realised that she had labelled the array indexes incorrectly: "*Coming back to A again,* `int`*, I have... (erasing on the answer sheet)... I am just writing down the numbers again as well as their corresponding position on top because I forget that they are not numbered* `1, 2, 3`*. They are numbered* `0, 1, 2, 3`". Through a critical review [12] of her initial notes, this participant was able to recognise her earlier mistake and take the necessary corrective action.

Acknowledge When Lost. While Polya [30] encourages problem solvers to discard their original plans in favour of alternative plans (if the original plans are not working), the

participants in this study could not always devise an alternative. Eight of the participants (25 occurrences) acknowledged at some point during this problem-solving step that they were either lost or confused. While tracing through the second `for` loop of Q3, P8 said: "*That's the part where I am really stuck—the second loop*", while P6 said: "*So this second `for` loop is the one that is freaking me out*". Both participants got lost at this point because they were unable to interpret the combined index in the statement: `b[x[i]]= true;`. As another example of this strategy, P4 correctly identified option C as the answer to Q3, but when asked why he thought the answer was C, he was neither sure of himself nor of his answer. Instead, he was quick to acknowledge the limitations of his general programming skills: "*Like I said, I am not the best at these programming subjects*". He then attempted to explain his reasoning which consequently resulted in him selecting option A (which was incorrect) as his final answer. Although this participant was able to identify the correct answer, he could not explain his reasoning. This could imply that his original selection of option C was a guess rather than a definitive answer.

4.4 Evaluate a Solution

At this stage, the participants had already identified answers to the questions. Instances of personal reflections were observed when the participants attempted to justify these answers.

Defend an Incorrect Answer. During the evaluation of their final answers, six participants (21 occurrences) remained confident while attempting to defend their incorrect answers. While P9 completely ruled out option C (the correct answer) as a possible answer to Q3 ("*so option C does not work"*), P2 remarked as follows on his answer for Q3: "*So the answer here, I think it should be `4` [option D] because this statement says, if `i` is less than `b.Length`, and `b.Length` is `5`, we will increment it*". Although P3 performed a thorough analysis and followed a reasoning strategy to answer Q6, he ended up saying: "*I believe, I am taking option A"* while option B was the correct answer. The interviewer did ask him to further explain why he thought option A was the correct option, but he still ended up giving the same incorrect answer. Although all the participants mentioned here attempted to assess or re-assess their answers, they just repeated the same flawed reasoning to arrive at the same incorrect answers. This serves as a further indication that several of the participants lacked some basic programming knowledge.

Verify Answer. While reflecting on their final answers, six participants (18 occurrences) attempted to verify or justify their selected answers while some also elaborated on their reasons for not selecting any of the other answer options. As part of the process to verify his selection of option B as the correct answer for Q6, P3 also provided justification for why he thought some of the other options were incorrect: "*Option D is totally wrong because it says if this number, in my example, `2` is greater than `3`, therefore it is ascending, which is `false`. Then I have this one [pointing at the values on his written down example], so it's `true` and it returns `false`. Then it will go and take the second value, check and increment and check the other one"*. P3 followed a similar strategy

while reflecting on choosing option B as the correct answer for Q3: *"Okay, this one [scribbling and looking at his written down example] will be wrong because* `j` *starts from* `zero`*, and here in this example, you don't check the numbers that are behind - you only check the numbers that are after that position. So, options A and B in this particular example will be out"*. Both of these excerpts illustrate the level of critical thinking [12] that P3 applied during the problem-solving process. These excerpts also illustrate how problem solvers can reuse some of the strategies from earlier steps (e.g., make notes, use test cases, and eliminate possibilities) while evaluating their solutions.

5 SWOT Analysis

Our next step was to perform a SWOT analysis of the SCC problem-solving strategies identified in Sect. 4 (see Table 1). Strengths refer to strategies that the participants performed well during their problem-solving processes, while the weaknesses are aspects that were lacking from their skills sets and need to be improved. The opportunities are strategies that educators can include when modelling a more ideal SCC problem-solving strategy to their students. The threats are unfavourable strategies practised by the participants that could lead to an overwhelmingly flawed problem-solving process.

Table 1. SWOT analysis of problem-solving strategies.

Strengths	Weaknesses
• Read through instructions • Identify important concepts and/or terms • Acknowledge when lost • Eliminate incorrect answer options at an early stage • Make notes for future reference • Utilise test cases • Confirm initial understanding • Recognise coding patterns • Express confidence in devised solution • Verify final answer	• Self-doubt in own programming abilities • Misinterpret basic code syntax • Fail to understand the working of basic programming structures • Reluctant to change line of thinking when lost • Struggle to interpret nested concepts • Fail to recognise all coding patterns • Incorrect use of programming terminology
Opportunities	**Threats**
• Realise value of devising an actual problem-solving plan • Emphasise importance of using correct programming terminology • Identify specific gaps in programming knowledge	• Disregard earlier interpretations or notes • Content to provide answers to all the questions (regardless of correctness) • Look at sections of code in isolation • Ignore "difficult" pieces of code completely • Resort to guessing when reasoning fails

6 Bottleneck Identification

By considering the identified weaknesses and threats (see Table 1), we identified six main learning bottlenecks that could prevent novice programmers from mastering problem solving in the CS discipline.

Bottleneck 1: Students Ignore Previously Understood Information at Later Stages of the Problem-Solving Process. Many of the examples illustrated in the students' excerpts (see Sect. 4) indicate the inconsistencies they exhibited while answering the given questions. It seems as if students were forgetting the correct understanding they previously had, unless they were, in the first place, not even convinced that their original understanding was correct. In this way, the students' original thinking and analysis [4] were not necessarily helping them during the entire problem-solving process.

Bottleneck 2: Students Regard Their Working Notes as a Non-crucial Component of the Problem-Solving Process. Although many of the students made conscious efforts to demonstrate how they were going about answering the questions and making their thinking logic as explicit as possible, many of the markings, comments, notes, and underlinings [18, 31, 37] they made were either overly rough or unreadable. Some of the participants also ignored their initial notes in later stages of the problem-solving process.

Bottleneck 3: Students Doubt Their Own Programming Abilities. From many of the excerpts presented in Sect. 4, it was evident that the students were not always confident of themselves. A level of doubt was observed not only with concepts they struggled to understand, but also with concepts they seemed to find fairly easy to comprehend. Consequently, they resorted to gathering fragmented pieces of information in their minds, hence making it difficult to make usable mental connections. This type of reasoning makes it difficult for one to gain a proper understanding of the question [5], and often results in guessing the answers.

Bottleneck 4: Students Are Unable to Comprehend Nested Concepts. Although all the students identified the crucial terms or concepts of computer programming, some of them were unable to fully comprehend even the most basic of these concepts and/or terms when these were combined within other programming structures. Consequently, some students were unable to organise their continuous understanding of the question around these basic concepts [8], and therefore resorted to viewing pieces of code in isolation.

Bottleneck 5: Students Are Unable to Easily Change Their Viewpoints During the Problem-Solving Process. Although some students were able to change their initial understanding, others struggled to find alternative avenues of reasoning when they got lost or stuck. Although these students attempted to make logical deductions [21], they still were unable to correctly interpret the given source code.

Bottleneck 6: Students Are More Focused on the Answer Than the Process of Arriving at the Answer. The ultimate goal of most of the students was to get to the final answer. Consequently, many of their problem-solving steps were unclear, and the

expected direct reasoning [30] steps did not come to the fore. It can therefore be deduced that they regarded their final answers as more important than the actual problem-solving process. There were also instances where some students arrived at the correct answer either using flawed reasoning or displaying a lack of comprehension of the relevant programming concepts.

7 Conclusions and Future Work

The lack of problem-solving skills remains a challenge to undergraduate CS students. Comprehension of the mental moves made by students during problem solving can be essential in helping them to overcome related challenges. By focusing on Step 1 of the seven-step DtDs framework (identifying places in a course where many students consistently fail to master crucial material), this study aimed to (1) explore the problem-solving strategies utilised by novice programmers during SCC; (2) relate these strategies to Polya's four basic problem-solving steps; and (3) utilise a SWOT analysis of these strategies for the identification of problem-solving bottlenecks experienced by novice programmers. Thematic analysis of data collected by means of asking questions, observations, and artefact analysis revealed that the novice programmers in this study did not necessarily follow a well-defined problem-solving process. We were, however, able to link the specific strategies they employed to all four of Polya's basic problem-solving steps. While some of the identified strategies could be mapped to a single step, several strategies were utilised repeatedly in different stages of the problem-solving process. A SWOT analysis of the identified strategies provided further insight regarding strategies that the novices performed well (strengths), and the aspects that were lacking from their skills sets (weaknesses). We also identified several undesirable strategies (threats) that could hamper their problem-solving attempts. The listed observations point to strategies that educators could utilise to assist students in improving their problem-solving skills.

This study also illustrated a novel approach in utilising a SWOT analysis as a learning bottleneck identification tool. By considering the identified strengths and weaknesses, we formulated six bottlenecks that could prevent novice programmers from mastering problem solving in the CS discipline. The fact that these bottlenecks were identified from the problem-solving attempts of final-year undergraduate students further highlights the problem-solving difficulties that are currently not effectively addressed in introductory CS courses. Further research is needed to investigate how the identified problem-solving bottlenecks can be addressed through the application of the remaining six steps of the DtDs framework [23].

Appendix

Question 3
Consider the following source code fragment:

```
int[] x = {1, 2, 3, 3, 3};
bool[] b = new bool[x.Length];

for (int i = 0; i < b.Length; ++i)
  b[i] = false;

for (int i = 0; i < x.Length; ++i)
  b[x[i]] = true;

int count = 0;

for (int i = 0; i < b.Length; ++i)
{
if (b[i] == true)
    ++count;
}
```

After this source code is executed, `count` contains:

a) 1
b) 2
c) 3
d) 4
e) 5

Question 6
The following method `isSorted` should return `true` if the array is sorted in ascending order. Otherwise, the method should return `false`:

```
public static bool isSorted (int[] x)
{
    //missing source code goes here
}
```

Which of the following is the missing source code from the method `isSorted`?

a)
```
bool b = true;
for (int i = 0; i < x.Length - 1; i++)
{
  if (x[i] > x[i + 1])
    b = false;
  else
    b = true;
}
return b;
```

b)
```
for (int i = 0; i < x.Length - 1; i++)
{
  if (x[i] > x[i + 1])
    return false;
}
return true;
```

c)
```
bool b = false;
for (int i = 0; i < x.Length - 1; i++)
{
  if (x[i] > x[i + 1])
    b = false;
}
return b;
```

d)
```
bool b = false;
for (int i = 0; i < x.Length - 1; i++)
{
  if (x[i] > x[i + 1])
    b = true;
}
return b;
```

e)
```
for (int i = 0; i < x.Length - 1;i++)
{
  if (x[i] > x[i + 1])
    return true;
}
return false;
```

References

1. Alturki, R.A.: Measuring and improving student performance in an introductory programming course. Inf. Educ. **15**(2), 183–204 (2016). https://doi.org/10.15388/infedu.2016.10
2. Anyango, J.T., Suleman, H.: Teaching programming in Kenya and South Africa: what is difficult and is it universal ? In: Proceedings of the 18th Koli Calling International Conference on Computing Education Research, pp. 1–2. ACM, New York (2018). https://doi.org/10.1145/3279720.3279744
3. Belski, I.: Teaching thinking and problem solving at university: a course on TRIZ. Creativity Innov. Manag. **18**(2), 101–108 (2009). https://doi.org/10.1111/j.1467-8691.2009.00518.x
4. Bransford, J., Brown, A., Cocking, R.: How People Learn: Brain, Mind, Experience, and School (Expanded). National Academy Press, Washington, D.C. (2000). https://doi.org/10.4135/9781483387772.n2
5. Bransford, J.D., Stein, B.S.: The Ideal Problem Solver: A Guide for Improving Thinking, Learning, and Creativity, 2nd edn. W.H. Freeman, New York (1993)
6. Braun, V., Clarke, V.: Using thematic analysis in psychology. Qual. Res. Psychol. **3**, 77–101 (2006). https://doi.org/10.1191/1478088706qp063oa
7. Cheah, C.S.: Factors contributing to the difficulties in teaching and learning of computer programming: a literature review. Contemp. Educ. Tech. **12**(2), 1–14 (2020). https://doi.org/10.30935/cedtech/8247
8. Chi, M., Glaser, R., Rees, E.: Expertise in problem solving. In: Sternberg, R. (ed.) Advances in the Psychology of Human Intelligence, vol. 1, pp. 7–75. Lawrence Erlbaum Associates Inc, New Jersey (1982)
9. Conn, C., McLean, R.: Bulletproof Problem Solving: The One Skill That Changes Everything. Wiley, Hoboken (2019)
10. Cunningham, K., Blanchard, S., Ericson, B., Guzdial, M.: Using tracing and sketching to solve programming problems: Replicating and Extending an Analysis of What Students Draw. In: Proceedings of the 2017 ACM Conference on International Computing Education Research, pp. 164–172. ACM, New York (2017). https://doi.org/10.1145/3105726.3106190
11. Diaz, A., Middendorf, J., Pace, D., Shopkow, L.: The history learning project: a department "decodes" its students. J. Am. Hist. **94**(4), 1211–1224 (2008). https://doi.org/10.2307/25095328
12. Egbert, J.: Methods of Education Technology: Principles, Practice, and Tools. Pearson, New Jersey (2017). https://opentext.wsu.edu/tchlrn445/
13. Fedorenko, E., Ivanova, A., Dhamala, R., Bers, M.U.: The Language of programming: a cognitive perspective. Trends Cogn. Sci. **23**(7), 525–528 (2019). https://doi.org/10.1016/j.tics.2019.04.010
14. Fitzgerald, S., Simon, B., Thomas, L.: Strategies that students use to trace code: an analysis based in grounded theory. In: Proceedings of the 1st International Workshop on Computing Education Research, pp. 69–80. ACM, New York (2005). https://doi.org/10.1145/1089786.1089793
15. Herrmann, J.W.: Rational decision making. In: Balakrishnan, N., Colton, T., Everitt, B., Piegorsch, W., Ruggeri, F., Teugels, J.L. (eds.) Wiley StatsRef: Statistics Reference Online, pp. 1–9. Wiley, New York (2017). https://doi.org/10.1002/9781118445112.stat07928
16. Humphrey, A.S.: SWOT analysis. Long Range Plan. **30**, 46–52 (2005)
17. Khomokhoana, P.J., Nel, L.: Decoding source code comprehension: bottlenecks experienced by senior computer science students. In: Tait, B., Kroeze, J., Gruner, S. (eds.) SACLA 2019. CCIS, vol. 1136, pp. 17–32. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-35629-3_2

18. Lister, R., et al.: A multi-national study of reading and tracing skills in novice programmers. ACM SIGCSE Bull. **36**(4), 119–150 (2004). https://doi.org/10.1145/1041624.1041673
19. Loksa, D., Ko, A.J., Jernigan, W., Oleson, A., Mendez, C.J., Burnett, M.M.: Programming, problem solving, and self-awareness: effects of explicit guidance. In: Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, pp. 1449–1461. ACM, New York (2016). https://doi.org/10.1145/2858036.2858252
20. Malik, S.I., Coldwell-Neilson, J.: Impact of a new teaching and learning approach in an introductory programming course. J. Educ. Comput. Res. **55**(6), 789–819 (2017). https://doi.org/10.1177/0735633116685852
21. Malloy, C.E., Jones, M.G.: An investigation of African American students' Mathematical problem solving. J. Res. Math. Educ. **29**(2), 143–163 (1998)
22. Margulieux, L.E., Morrison, B.B., Decker, A.: Reducing withdrawal and failure rates in introductory programming with subgoal labeled worked examples. Int. J. STEM Educ. **7**(1), 1–16 (2020). https://doi.org/10.1186/s40594-020-00222-7
23. Middendorf, J.K., Pace, D.: Decoding the disciplines: a model for helping students learn disciplinary ways of thinking. New Dir. Teach. Learn. **98**, 1–12 (2004). https://doi.org/10.1002/tl.142
24. Middendorf, J., Shopkow, L.: Overcoming Student Learning Bottlenecks: Decode Your Disciplinary Critical Thinking. Stylus Publishing LLC, Sterling (2018)
25. Moore, D., Zabrucky, K., Commander, N.E.: Validation of the metacomprehension scale. Contemp. Educ. Psychol. **22**(4), 457–471 (1997). https://doi.org/10.1006/ceps.1997.0946
26. Pace, D.: The Decoding the Disciplines Paradigm: Seven Steps to Increased Student Learning. Indiana University Press, Bloomington (2017)
27. Patton, M.Q.: Qualitative Research & Evaluation Methods: Integrating Theory and Practice, 4th edn. SAGE, Thousand Oaks (2015)
28. Pinnow, E.: Decoding the disciplines: an approach to scientific thinking. Psychol. Learn. Teach. **15**(1), 94–101 (2016). https://doi.org/10.1177/1475725716637484
29. Plowright, D.: Using Mixed Methods: Frameworks for An Integrated Methodology. SAGE, Thousand Oaks (2011)
30. Polya, G.: How to Solve It: A New Aspect of Mathematical Method. Doubleday, University of Michigan (1957)
31. Powell, N., Moore, D., Gray, J., Finlay, J., Reaney, J.: Dyslexia and learning computer programming. ACM SIGCSE Bull. **36**(3), 242 (2004). https://doi.org/10.1145/1026487.1008072
32. Praveen, A.: Program comprehension and analysis. Int. J. Eng. Appl. Comput. Sci. **1**(01), 17–21 (2016). https://doi.org/10.24032/ijeacs/0101/04
33. Preece, J., Rogers, Y., Sharp, H.: Interaction Design: Beyond Human-Computer Interaction, 4th edn. Wiley, New York (2015)
34. Rahmat, M., Shahrani, S., Latih, R., Yatim, N.F.M., Zainal, N.F.A., Rahman, R.A.: Major problems in basic programming that influence student performance. Proc. Soc. Behav. Sci. **59**, 287–296 (2012). https://doi.org/10.1016/j.sbspro.2012.09.277
35. Renumol, V., Jayaprakash, S., Janakiram, D.: Classification of cognitive difficulties of students to learn computer programming. Indian Institute of Technology (2009). http://dos.iitm.ac.in/publications/LabPapers/techRep2009-01.pdf
36. Samejima, M., Shimizu, Y., Akiyoshi, M., Komoda, N.: SWOT analysis support tool for verification of business strategy. In: IEEE International Conference on Computational Cybernetics, pp. 631–635. IEEE, Talinn (2006). https://doi.org/10.1109/ICCCYB.2006.305700
37. Scalabrino, S., et al.: Improving code readability models with textual features. In: Proceedings of the 24th IEEE International Conference on Program Comprehension, pp. 1–10. IEEE, Austin (2016). https://doi.org/10.1109/ICPC.2016.7503707

38. Scherer, R., Siddiq, F., Sánchez Viveros, B.: A meta-analysis of teaching and learning computer programming: effective instructional approaches and conditions. Comput. Hum. Behav. **109**(106349), 1–18 (2020). https://doi.org/10.1016/j.chb.2020.106349
39. Shopkow, L.: How many sources do I need? Hist. Teach. **50**(2), 169–200 (2017)
40. Uwano, H., Nakamura, M., Monden, A., Matsumoto, K.I.: Analyzing individual performance of source code review using reviewers' eye movement. In: Proceedings of the 2006 Symposium on Eye Tracking Research & Applications, pp. 133–140. ACM, New York (2006). https://doi.org/10.1145/1117309.1117357
41. Yurdugül, H., Aşkar, P.: Learning programming, problem solving and gender: a longitudinal study. Proc. Soc. Behav. Sci. **83**, 605–610 (2013). https://doi.org/10.1016/j.sbspro.2013.06.115