Difficulty and Complexity of Introductory Programming Problems

Tomáš Effenberger, Jaroslav Čechák, and Radek Pelánek

Masaryk University Brno
{xeffenb1, xcechak1, pelanek}@fi.muni.cz

Abstract. The paper is submitted as a *Research Paper*. Difficulty and complexity measures are used for problem sequencing, authoring, and detecting problematic problems. We review measures of observed difficulty and intrinsic complexity for introductory programming exercises and explore the relationship between these measures using correlation analysis and projections onto a plane. Even though the exercises are similar, only a few trends hold for all of them, stressing the necessity for validating previous results about complexity and difficulty measures that were based on data from a single exercise.

1 Introduction

Although the terms difficulty and complexity are often used as synonyms, they can be used to distinguish two different concepts. We use the term *complexity* for intrinsic characteristics of problems which influence performance, but which are independent of the context such as the people solving the problem [12, 5]. The complexity of programming problems can be computed from the problem statement and solution, for example, as the number of lines, number of flow-of-control structures, or number of unique programming concepts involved. The *difficulty*, in contrast, relates to the observed performance, such as failure rate, median solving time, or the number of attempts.

Both complexity and difficulty can be useful for problem sequencing, i.e., deciding in which order to present problems to the student [4]. It is a common practice to present the problems from the least complex to the most complex, or from the least difficult to the most difficult. Occasional drops in one of the complexity or difficulty dimensions might, nevertheless, make the experience more enjoyable [11]. When used for problem sequencing, difficulty measures suffer from the cold start problem. Complexity measures can be computed just from the problem statement and solution, and so they can be used to estimate the difficulty of a new problem before enough performance data are collected [16].

In addition to the problem sequencing, understanding the relationship between various complexity and difficulty measures can be useful for problem authoring, indicating, for example, which complexity dimensions to change to obtain a problem of the desired difficulty. On the other hand, the discrepancy between the estimated complexity and observed difficulty can serve as a simple heuristic for detecting suspicious problems that should be investigated.

In this work, we explore the relationship between complexity and difficulty measures for problems from four exercises in two systems for learning introductory programming, in which students create programs in either block-based or textual programming interface. We use a correlation analysis for comparing pairs of measures, and a projection of the measures onto a plane for a macroscopic view of all the relationships at once. Most trends are specific to individual exercises; however, there were a few observations that held for all four exercises.

2 Difficulty and Complexity Measures

Both complexity and difficulty can be measured in many ways. Some measures are domain-independent (e.g., failure rate), while others are specific to programming (e.g., number of flow-of-control structures). Some complexity measures even require tailoring for a given type of programming exercise; for instance, specification of possible programming concepts.

2.1 Complexity for Programming Problems

In the context of introductory programming, the problem solution, which is often code, is more amenable to automatic processing than problem statement, which is usually a text in natural language. Examples of complexity measures computed from a code include lines of code, number of flow-of-control structures [1], cyclomatic complexity [18], Halstead complexity measures based on the number of operators and operands [10], and number of unique programming concepts involved [10].

Even if the solution code stays the same, the complexity of a problem can be significantly shifted by the change in the problem statement. One reason is the linguistic complexity of the problem statement (e.g., how long is the text), and reference to external domains [16]. In micro-world programming exercises, a corresponding property is a complexity of the world description; for example, the size of the grid, or the number of distinct objects. Problem statement also determines Bloom's level of complexity for involved programming concepts [9]. The Bloom's level is also influenced by scaffolding, such as hints, starter code, a toolbox of programming blocks, or even the name of the problem.

2.2 Difficulty for Programming Problems

In addition to domain-independent failure rate and response time, the difficulty of programming problems can also be measured in the number of code edits, executions, and submits. These simple measures are straightforward to compute; however, they are affected by several biases such as learning, problem ordering, attrition bias, and self-selection bias [14]. In the context of testing, Item Response Theory models offer a group invariant estimates of difficulty using correctness of answers [7]. These models can be extended to incorporate learning [13], and to estimate problem solving times [15]. Difficulty can also be estimated by asking students for feedback [17].

Table 1. Programming exercises and data used for analysis.

Exercise	Interface	Problems	Students	Attempts
RoboMission	blocks	85	3,800	62,500
Turtle Blockly	blocks	77	11,000	63,600
Turtle Python	text	51	2,400	11,900
Python	text	73	2,000	10,700

3 Data Analysis

We use data from two systems for the practice of introductory programming (`robomise.cz` and `umimeprogramovat.cz`) and analyze the relationship between several complexity and difficulty measures.

3.1 Exercises and Measures

Table 1 presents an overview of four programming exercises that we use for analysis. In the RoboMission exercise [8], students build programs using a block-based programming interface [2] to guide a robot on a grid. In Turtle Blockly and Turtle Python, students command a turtle to draw assigned geometric objects [6]; the difference between the two is in the programming interface (block-based vs. textual). In the fourth exercise, students write Python code to solve problems with numbers, strings, and lists, and their code is evaluated on hidden test cases.

To assess the complexity of a problem we used the following measures: instruction length, code length, and the number of unique concepts; each with varying granularity. The instruction length was computed as a number of words or characters in the problem statement. The code length was computed as a number of lines or characters in the author's solution. The concepts vary across the exercises and they include flow-of-control statements, logical and mathematical operators, turtle commands, and robot commands. There were three levels of granularity: every concept independently, small groups of closely related concepts (e.g., logical operators), and coarse groups (e.g., all flow-of-control statements). Only the occurrence of the concept is measured; the frequency is not.

The difficulty was measured by failure rate, median solving time, and the median number of attempts. The failure rate is the ratio of unsuccessful students to all students than encountered the given problem. The median solving time is computed only from times of successful students. An attempt is an act of student submitting the solution for an automated assessment. Each measure also has six other variants. They were obtained by filtering out students with less than k visited problems or taking only attempts from kth encountered problem onward in the student's solving sequence. The selected values of k were 3, 5, and 10.

3.2 Relations among Measures

When presented with many measures, it is not obvious which one to use. Some may measure the same aspect of the complexity or difficulty, and some may

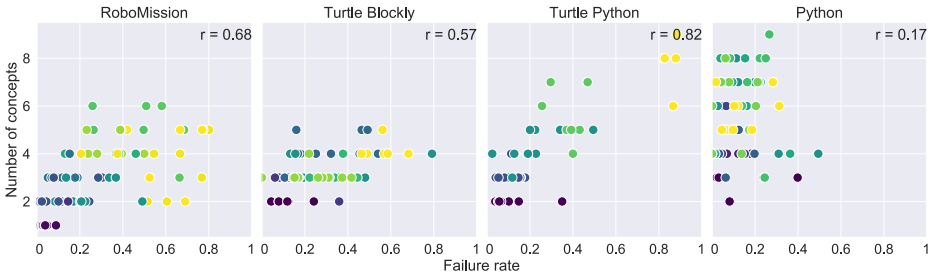

Fig. 1. Scatterplots and Spearman's correlation coefficients (r in the top right corner) for number of concepts and failure rate across the studied exercises. Colors denote problems assigned to the same problem set within the exercise.

vastly differ from others. Furthermore, the same measure can have multiple variants based on the granularity (e.g., concepts) or the used filtering (e.g., failure rate). To gain insight we propose using Spearman's correlation and Principal component analysis (PCA) projection to quantify the measure similarities.

Spearman's correlation coefficient is a great way of comparing a pair of measures. It is better suited for an in-depth comparison of a smaller number of measures. PCA, on the other hand, is well suited for providing a broader overview of measures and their similarities. PCA tries to project the data-points (measures) onto a smaller dimensional space while retaining as much variation in the data as possible. Both tools can be used to asses redundancy of measures.

The sample of results of our correlation analysis is presented in Fig. 1. The figure shows only a single pair of measures, number of concepts (variant with small groups) and failure rate (without any filtering), as an example. It illustrates that a conclusion drawn from a single type of exercise may not generalize well to other exercises. Here the correlation is fairly high for Turtle Python while abysmal for Python. Different variants of the same measure generally correlated well (r of 0.7 or higher) across all exercises. Although, there were some exceptions to this rule, e.g., variants of concepts and failure rate for Python exercise.

The results from our PCA analysis is presented in Fig. 2. It illustrates the variability of measure variants across exercises. It is worth noting that variants of the same measure form a tight cluster which is consistent with our findings in the correlation analysis. The failure rate in Python is, once again, an exception and its cluster is less compact than clusters for other exercises and measures. The analysis also shows that the same type of measures tends to be closer together. The strange behavior of the median number of attempts in Python can be attributed to different data gathering methodology.

3.3 Using Complexity for Difficulty Estimation

Complexity measures can be used to estimate the difficulty of new problems before any performance data are collected. We analyzed data from four programming exercises described in section 3.1 to find if there is a simple universal

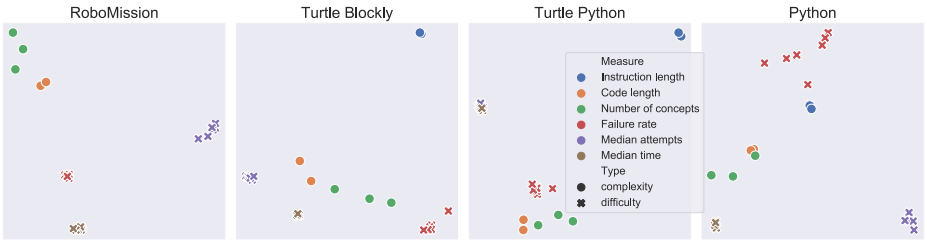

Fig. 2. PCA projections of measures across the studied exercises.

complexity measure that would be a reasonable predictor of difficulty. We compared two complexity measures: lines of code, and the number of concepts. For each complexity measure, the order of the problems in each exercise according to this measure was compared to the order according to the observed difficulty. As a proxy for difficulty, we used an average order according to solving time, the number of attempts, and failure rate. We summarized the discrepancies between orderings using Spearman's correlation coefficient.

For a complexity measure to be a robust predictor of the difficulty, high global correlation is not enough. Confounding factors can lead to a positive trend on the global level, even if there is no trend within subgroups [3]. For example, in Turtle Python exercise, there is a high global correlation between failure rate and the number of concepts (0.82), but Fig. 1 suggests that the correlation is much lower within problem sets. To quantify whether a given complexity measure would be successful in ordering problems within the problem sets, we computed correlations between orderings within individual problem sets, and compared the average within-problem-set correlations to the global within-exercise correlations.

The results are shown in Table 2. Neither of the complexity measures had high correlations with difficulty in all exercises. The within-exercise correlations ranged between 0.50 and 0.59 for lines of code, and between 0.53 and 0.83 for the number of concepts, and they further dropped by 0.07–0.34 when computed within individual problem sets of the exercise. While the number of concepts was more accurate in predicting the global trend, the lines of code had usually higher local correlations. This indicates that the number of concepts is a preferable measure for the division of problems into problem sets, while the lines of code might be more useful to consider for ordering the problems within the problem sets—though not as a single criterion.

4 Discussion

We have explored the relationship between complexity and difficulty measures in four introductory programming exercises. Even though these exercises were closely related, only a few trends held for all of them. We should be, therefore, cautious not to make general conclusions about the difficulty or complexity of

Table 2. Spearman's correlation coefficients between observed difficulty and measured complexity (lines of code, number of concepts) in 4 programming exercises. Global correlations are computed across all problems in the exercise, while local correlations are average correlations within individual problem sets. The last column shows the difference between global and local correlations.

Exercise	Complexity	Global correlation	Local correlation	Difference
Robomission	lines	0.53	0.46	0.07
Robomission	concepts	0.66	0.44	0.22
Turtle Blockly	lines	0.50	0.41	0.09
Turtle Blockly	concepts	0.64	0.31	0.34
Turtle Python	lines	0.51	0.32	0.20
Turtle Python	concepts	0.83	0.53	0.30
Python	lines	0.59	0.41	0.18
Python	concepts	0.53	0.29	0.24

programming problems using data from just a single exercise. Instead of proposing a single universal measure of complexity or difficulty for all programming exercises, we focus on a methodology for clarifying what the suitable measures for given exercise are.

When there are many measures to choose from, PCA provides an overview of their similarity. Although the proximity of projected measures does not directly translate to correlation, it gives at least a rough estimate. From our PCA analysis, it is clear that instruction lengths, in our case, are unrelated to other measures. This suggests some more sophisticated language analysis or an entirely different approach would be required for these measures to be useful. The various filtering methods resulted in highly similar difficulty measures both in terms of correlation and position in a projected plane. These results hint that filtering does not bring any tangible benefits.

Neither lines of code, nor the number of concepts are an accurate predictor of difficulty. Even though the number of concepts has a high global correlation with the difficulty for some exercises, the local within-problem-set correlations are lower. One approach to increase the precision is to further refine the information obtained from the problem statement and solution; for example, by detecting finer-grained and subtler concepts. Furthermore, different concepts should be weighted differently, especially if they are included in scaffolding.

Another approach to increase the precision of difficulty estimates is to extend a complexity measure to account for the context in which the problem is presented. For example, if a problem on while loops is a part of a problem set focusing on while loops, then the problem requires understanding *how* to use while loop, but not necessarily *when* to apply it. If the problem was moved into a mixed problem set practicing various control structures, the contextualized complexity would increase, as Bloom's level of complexity increased from *Understand* to *Apply* [9].

References

1. Andres Alvarez and Terry A Scott. Using student surveys in determining the difficulty of programming assignments. *Journal of Computing Sciences in Colleges*, 26(2):157–163, 2010.
2. David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. Learnable programming: blocks and beyond. *Communications of the ACM*, 60(6):72–80, May 2017.
3. Colin R Blyth. On simpson's paradox and the sure-thing principle. *Journal of the American Statistical Association*, 67(338):364–366, 1972.
4. Peter L Brusilovsky. A framework for intelligent knowledge sequencing and task sequencing. In *International Conference on Intelligent Tutoring Systems*, pages 499–506. Springer, 1992.
5. Donald J Campbell. Task complexity: A review and analysis. *Academy of management review*, 13(1):40–52, 1988.
6. Michael E Caspersen and Henrik Bærbak Christensen. Here, there and everywhere - on the recurring use of turtle graphics in cs 1. In *ACM International Conference Proceeding Series*, volume 8, pages 34–40, 2000.
7. R.J. De Ayala. *The theory and practice of item response theory*. The Guilford Press, 2008.
8. Tomáš Effenberger and Radek Pelánek. Towards making block-based programming activities adaptive. In *Proc. of Learning at Scale*, page 13. ACM, 2018.
9. Richard Gluga, Judy Kay, Raymond Lister, Sabina Kleitman, and Tim Lever. Coming to terms with bloom: an online tutorial for teachers of programming fundamentals. In *Proceedings of the Fourteenth Australasian Computing Education Conference-Volume 123*, pages 147–156. Australian Computer Society, Inc., 2012.
10. Petri Ihantola and Andrew Petersen. Code complexity in introductory programming courses. In *Proceedings of the 52nd Hawaii International Conference on System Sciences*, 2019.
11. Conor Linehan, George Bellord, Ben Kirman, Zachary H Morford, and Bryan Roche. Learning curves: analysing pace and challenge in four successful puzzle games. In *Proceedings of the first ACM SIGCHI annual symposium on Computer-human interaction in play*, pages 181–190. ACM, 2014.
12. Peng Liu and Zhizhong Li. Task complexity: A review and conceptualization framework. *International Journal of Industrial Ergonomics*, 42(6):553–568, 2012.
13. Radek Pelánek. Applications of the elo rating system in adaptive educational systems. *Computers & Education*, 98:169–179, 2016.
14. Radek Pelánek. The details matter: methodological nuances in the evaluation of student models. *User Modeling and User-Adapted Interaction*, 28:207–235, 2018.
15. Radek Pelánek and Petr Jarušek. Student modeling based on problem solving times. *International Journal of Artificial Intelligence in Education*, 25(4):493–519, 2015.
16. Judy Sheard, Angela Carbone, Donald Chinn, Tony Clear, Malcolm Corney, Daryl D'Souza, Joel Fenwick, James Harland, Mikko-Jussi Laakso, Donna Teague, et al. How difficult are exams?: a framework for assessing the complexity of introductory programming exams. In *Proceedings of the Fifteenth Australasian Computing Education Conference-Volume 136*, pages 145–154. Australian Computer Society, Inc., 2013.
17. Kelly Wauters, Piet Desmet, and Wim Van Den Noortgate. Item difficulty estimation: An auspicious collaboration between data and judgment. *Computers & Education*, 58(4):1183–1193, 2012.

18. Jacqueline Whalley and Nadia Kasto. How difficult are novice code writing tasks?: A software metrics approach. In *Proceedings of the Sixteenth Australasian Computing Education Conference-Volume 148*, pages 105–112. Australian Computer Society, Inc., 2014.