Towards Making Block-based Programming Activities Adaptive

Tomáš Effenberger
Masaryk University
Brno, Czech Republic
tomas.effenberger@mail.muni.cz

Radek Pelánek
Masaryk University
Brno, Czech Republic
pelanek@fi.muni.cz

ABSTRACT

Block-based environments are today commonly used for introductory programming activities like those that are part of the Hour of Code campaign, which reaches millions of students. These activities typically consist of a static series of problems. Our aim is to make this type of activities more efficient by incorporating adaptive behavior. In this work, we discuss steps towards this goal, specifically a proposal and implementation of a programming game that supports both elementary problems and interesting programming challenges and thus provides an environment for meaningful adaptation. We also discuss methods of adaptivity and the issue of evaluating student performance while solving a problem.

INTRODUCTION

Today there is a growing trend to get students acquainted with programming at least on the introductory level. A typical example of this trend is the Hour of Code campaign, which provides online introductory programming activities that reach millions of students [9] – exemplary learning at scale. The Hour of Code activities are typically based on restricted mini-languages [1], which allow students to get experience with programming in simplified environments. These activities often use block-based programming, often based on the popular, extensible Blockly implementation [4].

Most current block-based programming activities utilize a static series of tasks of increasing difficulty. To make these activities more efficient, it would be useful to make the progress adaptive. Even students with no programming knowledge differ in their predispositions for the task (experience with technology, logical reasoning capabilities). Our goal is thus to develop efficient, adaptive block-based programming activities.

Towards this goal, we propose a novel programming task that can support a wide range of problem difficulties. The task is a variation on the classical "robot in a grid world" setting, with the main novel feature of "continuous movement forward". We provide open source implementation of the task with 70 example problems. We also discuss a suitable approach to adaptivity for this task and we discuss the issue of measuring student performance in block-based programming.

L@S'18, London

ISBN .

DOI:

PROGRAMMING GAME

For adaptation to be really useful we need to have a suitable programming game that can support a wide range of problems to solve – we need a continuous gradation of problems from elementary problems to interesting programming challenges and a support for all typical aspects of introductory programming (sequential composition, repetition, conditional execution, nesting).

Commonly used environments for block-based programming are turtle graphics and a robot in a grid world. Turtle graphics [2] has several advantages. Even with simple programs, it can produce an attractive graphical output, which is motivating for students. It also provides a natural connection with mathematics, particularly concerning reasoning about angles. A disadvantage is that in the basic form it does not naturally support conditional commands, so it can be used to practice only a limited range of programming concepts.

The second popular approach is based on a virtual robot in a grid world, which was first introduced as "Karel the robot" [5]. Many current Hour of Code activities are variations on this theme, often keeping the basic principle and just using a popular graphics like Star Wars or Angry Birds. This time-tested approach works well. A slight disadvantage is that the robot movement requires many commands (e.g., letting the robot go around an obstacle requires four steps and four turns) and thus programs for more complex problem become lengthy and the block-based programming may become cumbersome.

We propose a novel variation on the "robot in a grid world" theme, which alleviates this problem by including a "default movement forward". Specifically, the game is presented as navigating a spaceship in a space. The goal is to get the spaceship to a final destination while avoiding obstacles and satisfying other conditions like collecting diamonds. Actions like left or right are automatically connected with movement (flying) forward. This allows compact specification of even complex spaceship trajectories.

Figure 1 provides several examples of the game world. Each field of the grid has a color and objects. Background colors are used for decisions; blue color is reserved for the final row (a destination). In addition to the spaceship, controlled by the

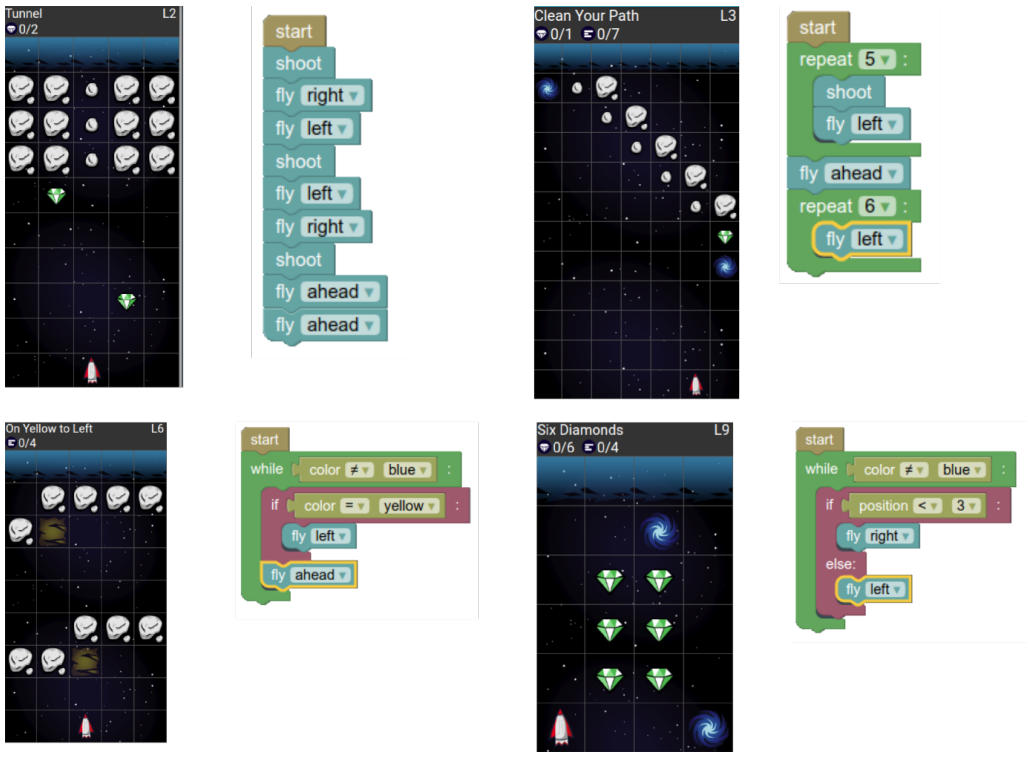

Figure 1. RoboMission examples

program, there are four other types of objects: asteroids, meteoroids, diamonds, and wormholes. The difference between the large asteroids and the small meteoroids, which both serve as obstacles, is that meteoroids can be destroyed by shooting. Diamonds prescribe the trajectory even more explicitly because all of them must be collected by the spaceship. If a spaceship flies into a wormhole, it immediately appears at the other wormhole (of the same color). Without wormholes, only a single field on each row is flown through (as a result of the default forward movement), which leads to long grids. Wormholes allow to reuse part of the grid and they can also make the planning of the path more interesting.

Programs are created by dragging-and-dropping blocks. The blocks include basic actions (fly forward, left, right, shoot), loops (repeat, while), conditional statements (if, if-else), and tests (background color, position on the x-axis). To enforce usage of loops, tasks define a limit on the number of statements that can be used. In addition, some tasks also set a limit on the number of shots to disallow shooting without thinking.

Figure 1 shows four examples of game words with a corresponding solution. The first problem (Tunnel) demonstrates that four basic actions allow creating a variety of simple tasks on sequential composition of commands. In the more advanced tasks, it is often enough to have a single action inside a body of a loop or a conditional statement (e.g., "if the color is yellow then fly left" in the On Yellow to Left task), which is an advantage brought by the default movement forward. The last example provides an illustration of a nontrivial programming example with a compact solution.

We provide open source implementation of this programming game called RoboMission. The implementation is based on the Blockly editor [4]. The source code is publicly available[1]. The game with 70 exemplary tasks is also available online at `https://en.robomise.cz/`.

ADAPTIVITY

Activities of the discussed type are typically presented as a fixed series of problems (levels) without any personalization. There are several ways how to make activity of this type adaptive. One possibility is to include hints – when a student is stuck, he may get a hint which helps him with progress. The generation of such hints in programming has been thoroughly explored (e.g., [8]). This approach, however, is useful particularly for classical textual programming, where students can get stuck on syntax and other minor problems and hints can help them overcome these problems. In the "robot in the grid" programming it is often hard to provide small hints. A useful hint often gives away the main idea of a solution. Thus we believe that rather than providing hints it is more useful to focus on the adaptive choice of task, i.e., giving the student a task that is neither too difficult nor too easy. Nevertheless, it would be useful to evaluate this hypothesis and potentially to combine the adaptive choice of tasks with adaptive hints.

A systematic approach to adaptive choice of tasks is to build a student model that estimates knowledge of a student and to use the estimated knowledge to select a suitable task. Such a student model could measure knowledge in individual knowledge components corresponding to different programming

[1] `https://github.com/adaptive-learning/robomission`

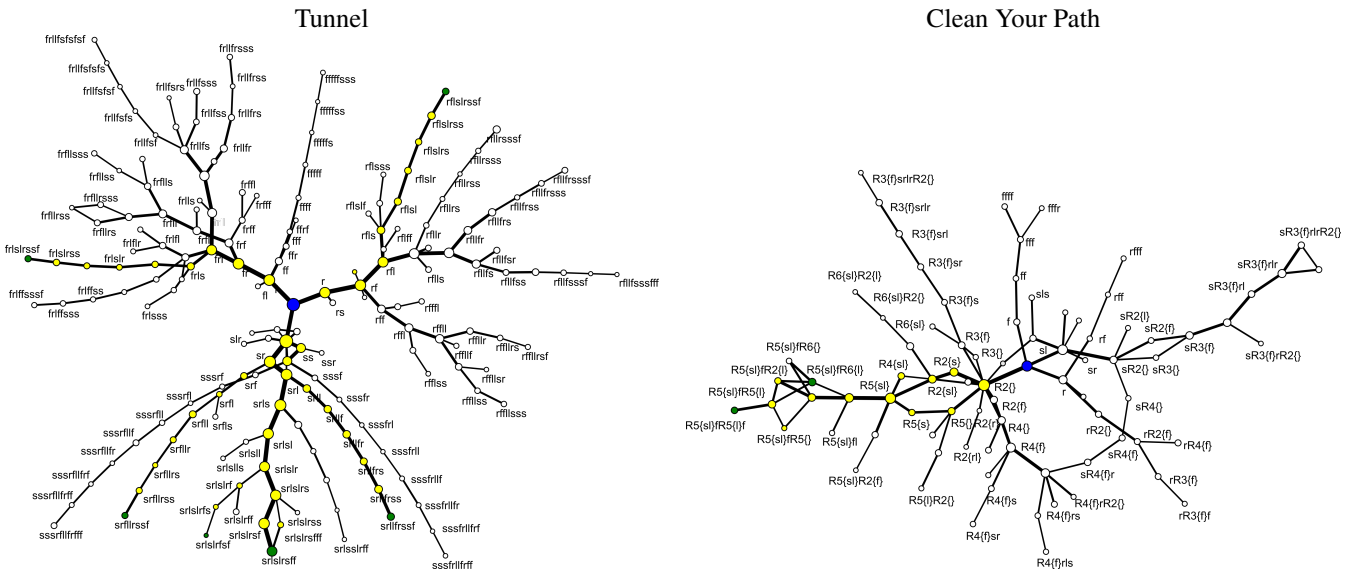

Figure 2. Examples of interaction networks. Nodes correspond to programs (f = forward, l = left, r = right, Rx = repeat x times); some node descriptions are omitted for readability. The size of nodes and the width of edges corresponds to the frequency of visits by students. Only most often visited nodes and edges are shown. Blue denotes the initial state, green denotes correct solutions, yellow denotes nodes on a direct path towards a solution.

concepts, with a Q-matrix mapping between tasks and knowledge components and potentially with knowledge components in a hierarchical domain model [6].

Such a systematic approach may be necessary for general programming activities. For the introductory block-based programming we believe that it is more advantageous to use a simpler, but well-tuned approach. Specifically, we propose to group tasks into a linearly ordered sequence of levels, selecting a task from the current level uniformly at random, and using a variation on the mastery learning principle [7] to decide when the student should move to the next level.

Levels should be fine-grained (e.g., "single simple while-loop") and consist of homogeneous tasks in order not to miss an important prerequisite and to achieve a consistent increase of the difficulty even with the exploration-maximizing random selection of tasks. We propose the following criterion to check if the level is fine-grained enough: a single excellently solved task should be enough to manifest mastery of the concepts covered in the current level.

When a student starts a new level, we set her skill to 0, and increase it after each solved task, until it reaches 1. As opposed to other domains, we never want to decrease the skill, so we only vary how much is the skill increased, based on the performance $p \in [0,1]$. Value of the performance should be chosen in such a way that $1/p$ tasks solved with such performance are expected to manifest the mastery. For example, if 2 tasks solved with "good performance" are needed, then we set $p = 0.5$ for good performance. Such convention leads the following update formula for skill: $s \leftarrow \min(1, s + \max(p, \frac{1}{n}))$, where n is the number of tasks in the current level. The $1/n$ term ensures that the student eventually masters the level after solving all tasks.

PERFORMANCE EVALUATION

Many student modeling approaches focus on modeling the domain structure but use simple methods for quantifying the performance of students on a given task (often using just correct/incorrect). In the context of introductory programming, a simple domain model could be sufficient, but it is necessary to pay attention to the evaluation of performance.

Once a student finishes solving a task, we need to characterize her performance by a value $p \in [0,1]$. Solving a programming problem (even a simple one) leads to quite rich data about the solving trajectory: we can take into account the individual edits, executions of the code, and time taken between steps. What is a suitable measure of performance?

On a general level, it is reasonable to separate the measurement of performance and the adaptive algorithm. It would be possible to provide all the detailed data about solving trajectory to the adaptive algorithm (or student model), but that would unnecessarily complicate the algorithm. It should be enough to use just a coarse summary of the performance, e.g., distinguishing four categories like "excellent performance", "good performance", "solved with problems", "solved with significant struggle".

The basic approach to such classification is to utilize the summary data about the performance like the total problem solving time and number of edits or executions.

We can also look at the specific trajectory of a student. The advantage of block-based programming over textual programming is that the programs are directly in the abstract-syntax tree format and mostly in "canonized form" (for textual programming it is nontrivial to get canonized abstract syntax tree [8]). Using student data we can thus easily build an interaction network [3] for a problem.

Figure 2 shows examples of such interaction network for two examples from Figure 1. These interaction networks are based on collected student data and the visualizations show the distribution of student trajectories. From these visualizations, we can see that we can distinguish "typical paths towards goal state" and "blind alleys". These can be useful to classify student performance.

ACKNOWLEDGMENTS

The authors thank members of the Adaptive Learning group at Masaryk University for fruitful discussions about RoboMission.

REFERENCES

1. Peter Brusilovsky, Eduardo Calabrese, Jozef Hvorecky, Anatoly Kouchnirenko, and Philip Miller. 1997. Mini-languages: a way to learn programming principles. *Education and Information Technologies* 2, 1 (1997), 65–83.
2. Michael E Caspersen and Henrik Bærbak Christensen. 2000. Here, there and everywhere- on the recurring use of turtle graphics in CS 1. In *ACM International Conference Proceeding Series*, Vol. 8. 34–40.
3. Michael Eagle, Drew Hicks, Barry Peddycord III, and Tiffany Barnes. 2015. Exploring networks of problem-solving interactions. In *Proceedings of the Fifth International Conference on Learning Analytics And Knowledge*. ACM, 21–30.
4. N Fraser and others. 2013. Blockly: A visual programming editor. (2013). `https://code.google.com/p/blockly`
5. Richard E Pattis. 1981. *Karel the robot: a gentle introduction to the art of programming*. John Wiley & Sons, Inc.
6. Radek Pelánek. 2017. Bayesian knowledge tracing, logistic models, and beyond: an overview of learner modeling techniques. *User Modeling and User-Adapted Interaction* 27, 3 (2017), 313–350.
7. Radek Pelánek and Jiří Řihák. 2017. Experimental Analysis of Mastery Learning Criteria. In *Proc. of User Modelling, Adaptation and Personalization*. ACM, 156–163.
8. Kelly Rivers and Kenneth R Koedinger. 2017. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education* 27, 1 (2017), 37–64.
9. Cameron Wilson. 2015. Hour of code—a record year for computer science. *ACM Inroads* 6, 1 (2015), 22–22.