Masaryk University
Faculty of Informatics

Adaptive System for Learning Programming

Master's Thesis

Tomáš Effenberger

Brno, Spring 2018

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Tomáš Effenberger

Advisor: doc. Mgr. Radek Pelánek, Ph.D

Acknowledgements

My journey towards this thesis started nine years ago at K-SCUK, a week-long interdisciplinary camp for high school students, organized by inspiring instructors. Three of them significantly contributed to my research interests and skills. Jan Rygl showed me the power of artificial intelligence and later became supervisor of my bachelor thesis. Jan Papoušek became my colleague in the Adaptive Learning group and helped me to overcome many challenges involved in the development of an adaptive learning system. Finally, Radek Pelánek invited me to the Adaptive Learning group, became my Master Yoda during my master studies, gave me a lot of actionable feedback, and never hesitated to dive with me into a fruitful brainstorming.

Several other friends contributed to the development of the adaptive learning system described in this thesis; by implementing some of its parts, performing analyses, discussing ideas, creating illustrations for the web, or by helping me with deployment of the system: Jaroslav Čechák, Jiří Mauritz, Jiří Řihák, Matěj Vaněk, and Barbora Vrzáková. I am also grateful to all people who tried the system and gave me feedback.

The development of the adaptive learning system was supported by Red Hat and Masaryk University Rector Project for outstanding master theses.

Abstract

This thesis presents an adaptive learning system for introductory programming. To support learning and motivation, the system uses block-based programming, a novel grid-world programming game, and progress visualization based on mastery learning. By adapting difficulty of tasks to the current skills, the system helps the students to immerse in the problem-solving activity and achieve the state of flow. Collected data are used in offline analyses leading to insights that help to iteratively improve the system.

Keywords

learning programming, domain modeling, student modeling, tutor modeling, programming game, adaptive learning system

Contents

1 Introduction

> *To prepare humanity for the next 100 years, we need more of our children to learn computer programming skills, regardless of their future profession. Along with reading and writing, the ability to program is going to define what an educated person is.* (S. Khan)

Programming is becoming an essential skill to learn. Not only is it useful in increasing number of professions, but also it develops abstract thinking and problem-solving ability. Our mission is to help children to learn basic programming efficiently while supporting their motivation for further learning. For this purpose, we build a personalized system for learning introductory programming, RoboMission[1] (Figure 1.1), that aims to adapt to the skills of the students to create an optimal learning experience.

Many tutorials for introductory programming already exists, including popular Hour of Code used by millions of students [68]. These tutorials combine several strategies to support learning and motivation, which are based on human needs, strengths, and weaknesses [29]. However, they are not personalized and offer the same sequence of tasks to everybody. RoboMission improves upon the existing systems by adapting to the students, using techniques of artificial intelligence, which were already successfully used in other domains [30, 41, 55].

Adaptation in learning systems can be performed at different time scales, ranging from an offline adaptation of the system to the entire population, to an online adaptation to individual students during their practice, either between tasks or even after each step of the student [2]. Learning systems can adapt to skills of the students, preferred learning styles, emotions (e.g., frustration, boredom), needs, motivation, and metacognition [4, 39].

In RoboMission, we focus on adaptation to skills and perform it at two time scales. First, we estimate skills of the students as they solve tasks so that we can recommend them optimally difficult tasks to practice next. By giving students tasks of difficulty matching their skill, we help them to achieve complete immersion into the problem-solving activity, known as the *state of flow* [13], which is essential for the optimal learning experience [8]. Second, using insight from analysis of collected data, we iteratively improve the behavior of the system, e.g., moving tasks to more appropriate problem sets, or adding new tasks practicing concepts that students often struggle with.

---

1. English localization available at `en.robomise.cz`.

Figure 1.1: RoboMission is a web application for learning introductory programming. The left panel displays a space-themed grid world. In the workspace on the right, students build programs using blocks describing actions such as `fly left` and `fly right`. The task is to create a program that would lead the spaceship to the last row.

We make the following contributions: First, we provide a summary of strategies to support learning and motivation used in existing online learning systems for introductory programming (Chapter 2) and an overview of relevant techniques for adaptive learning (Chapter 3). Second, we propose a new game designed for learning introductory programming (Chapter 4) and a practical approach to adaptivity (Chapter 5). Third, we implement this programming game and adaptive learning techniques in RoboMission (Chapter 6), test its usability at schools, and analyze data from four months of usage (Chapter 7 and [47, 51]).

2 Learning Programming

Learning to write programs stretches your mind, and helps you think better, creates a way of thinking about things that I think is helpful in all domains. (B. Gates)

This chapter describes the current state of the art of teaching programming, both from the view of successful learning systems and from the view of the research on learning programming.

2.1 Existing Systems for Learning Programming

There are many systems for learning programming, and they differ from each other in several significant aspects, which are summarized in Table 2.1. In order to be widely available, most of the systems for learning introductory programming are web applications and do not require any prerequisite other than reading and using a mouse. The systems often teach only a small subset of programming concepts, and there is no consensus on which concepts should be practiced first. Students usually spend most time by solving tasks, rather than by reading texts and watching videos. Most systems focus primarily on motivation and less on efficient learning, e.g., the sequence of tasks in the tutorials is usually same for every student.

In Sections 2.1.1 to 2.1.5, we briefly present the most notable existing systems, focusing on their features that support learning or motivation, which includes tasks with a visual output (e.g., turtle graphics), and block-based programming to avoid syntax errors. These successful strategies for teaching programming are described in more detail in Section 2.2.

Table 2.1: Differentiating aspects of systems for learning programming.

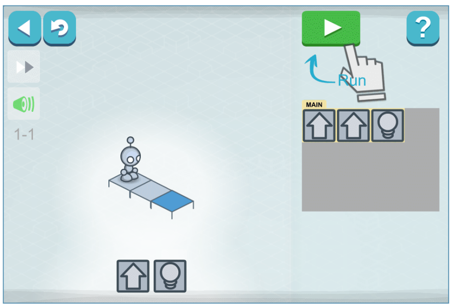
Aspect	Examples
Tangibility	computer applications, physical toys
Prerequisites	reading, typing, mathematics
Content	loops, variables, functions
Form	tasks, videos, texts
Tasks	robot on a grid, drawing with turtle
Programming language	block-based, textual
Adaptivity	task recommendation, mastery learning

2.1.1 LightBot

LightBot[1] is a mobile and web application offering a fixed sequence of tasks solved by a block-based programming language (Figure 2.1a). Students create short programs to control a robot in a grid world. The robot can not only walk and turn left or right but also jump and turn on lights. Having five different basic actions is useful for a diversity of simple tasks. The system covers sequences of commands, procedures, simple loops via tail-recursion, and conditional commands.

2.1.2 Problem Solving Tutor

Problem Solving Tutor[2] includes a few problem sets for practicing programming, such as Interactive Python, Turtle Graphics, or Robotanist [27]. Robotanist (Figure 2.1b) is similar to LightBot, with an addition of colored fields. Colors can be used in conditional commands, which allows for diverse tasks, including difficult problems with complex recursion. After each solved task, Problem Solving Tutor shows a recommendation for two next tasks with predicted solving times. Showing predicted time serves as a motivational element, posing a suitable challenge to overcome oneself [48]. A new version of the system, Umíme programovat[3], uses mastery learning for practicing basics of Python through multiple choice questions.



(a) LightBot

(b) Robotanist

Figure 2.1: LightBot and Robotanist provide square grids for commands.

1. `http://lightbot.com`
2. `https://tutor.fi.muni.cz`
3. `https://umimeprogramovat.cz` (in Czech only)

2.1.3 Blockly Games

Blockly is a popular block-based programming interface. In contrast to the blocks in LightBot, Blockly blocks can be nested and assembled arbitrarily (Figure 2.2a). Blockly Games[4] consist of several problem sets with Blockly, ordered by increasing difficulty. For example, in the first level, students learn how to compose blocks together like a puzzle, and in the second level, they learn loops and conditions by solving tasks in a maze. Final level serves as a transition from block-based programming to textual programming in JavaScript. Each level consists of 5-10 tasks, again ordered by increasing difficulty. The fixed order enables to build on the program from the previous task, thus gradually leading to more complex programs, resulting for example in sophisticated images in turtle graphics. Blockly Games includes non-ignorable interactive instructions, which require taking a described action before they disappear [23].

2.1.4 Hour of Code

Hour of Code[5] provides many one-hour tutorials, each containing about 15 tasks in fixed order, using Blockly-based language (Figures 2.2b and 2.3a). These tutorials focus on motivation, using themes from popular movies and games, and providing videos with famous people explaining programming concepts. The tutorials use high-level theme-specific blocks, such as "set droid to a random speed" or "create a snowflake branch", and they are restricted to only one or two programming concepts, e.g., sequences of commands and repeat loops. In some tasks, the built program is not a direct solution for a robot, but rather a game, in which the code specifies actions triggered on events.

2.1.5 Khan Academy

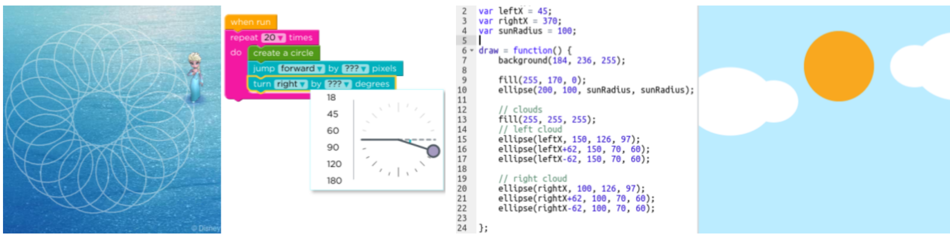
Khan Academy has a computer programming curriculum[6] that uses textual programming in JavaScript with functions for drawing shapes in absolute coordinates (Figure 2.3b). In addition to programming tasks, it contains text and video explanations, and projects. Some videos are in the form of interactive *talk-throughs*, in which the student can fiddle with the explained code at any moment to understand how it works.

---

4. `https://blockly-games.appspot.com/`
5. `https://hourofcode.com`
6. `https://www.khanacademy.org/computing/computer-programming`

(a) Blockly Games (Maze level) (b) Hour of Code (Star Wars theme)

Figure 2.2: Grid-world programming games using nested blocks.



(a) Hour of Code (Turtle graphics) (b) Khan Academy

Figure 2.3: Drawing tasks in relative and absolute coordinates.

2.2 Strategies to Support Learning

Learning programming is difficult because it requires to adopt algorithmic thinking, understand program execution, and remember the formal syntax of a programming language; all these three skills at once. To make learning easier, the systems presented in the previous section use diverse strategies, such as avoiding syntax errors, displaying visual output and providing hints. Various other strategies were tried in the past as well; article *Lowering barriers for Novice Programmers* [29] provides a detailed overview.

2.2.1 Avoiding Syntax Errors

A common strategy for avoiding syntax errors is to replace textual programming with drag-and-drop block-based programming. There are two basic types of block-based interfaces: either with a square grid defining the program shape, often one row per function (Figure 2.1), or the blocks are interlocking and can be assembled and nested arbitrarily, with no limit on maximum program length, often one vertical stack per function (Figure 2.2).

The interlocking blocks are more difficult to understand and manipulate with than the square grid. On the other hand, they are more flexible and correspond more closely to the usual textual programming. The square grid does not allow for nesting, which is a fundamental feature of computer programs. This restriction is usually overcome by combining condition and commands into a single block, by using recursion instead of loops, and by replacing nested sequences of commands by a new function.

A drawback of using block-based programming is that the students need to learn a proper textual programming language in the future to be able to implement more complex programs. Several controlled experiments were performed to test a hypothesis that it is still beneficial to start an introductory programming course with block-based programming, even when the students will be writing textual code later in the course [52, 66]. Results suggest that block-based interfaces indeed lead to increased learning and motivation; however, the evidence is not sufficiently convincing. For example, in some of these studies, the programming interfaces differed in more aspects than just in using blocks instead of text. Furthermore, these studies do not answer when it is the right moment to switch from block-based to textual programming.

To make the transition easier, the block representation of the code should match the programming language to which the students are expected to move in the next phase [67]. However, resemblance to a programming language sometimes conflicts with the readability of the blocks for novices. Instead of making compromises, the system can progressively change the available set of blocks in each level, making them more and more similar to a textual programming language. In the last level of Blockly Games, which employ this strategy, the text on the blocks matches the generated JavaScript precisely [23].

2.2.2 Visual Output

Learning systems can help students to track program execution by providing a clear visual representation of the current state and effects performed by the program. For example, in turtle graphics, the whole state is just a position and orientation of the turtle, and effects are the drawn lines (Figure 2.3a). Similarly, in simple games, such as those described in Sections 2.1.1 to 2.1.4, the grid world visualization contains complete information about the current state (Figures 2.1 and 2.2). For the simplicity of their visual output, drawings and grid world games have become prevalent task types in the current systems for learning programming.

2.2.3 Instructions and Hints

Most educational systems include instructions to explain new concepts such as loops and conditions. However, many students ignore instructions, no matter how prominent they are [23]. Figure 2.2a shows a possible solution to this problem: actionable non-ignorable instructions, which cannot be closed manually by the student, and disappear only once the student performs the action described in the instruction.

In addition to the instructions, some systems offer hints, which appear either upon a student request or automatically after a certain time of unsuccessful solving. Although it is possible to generate a hint in any state, using data of students who have successfully solved the task in the past [26], most existing systems use a few manually prepared hints, instead of relying on the automatic approaches.

2.3 Strategies to Support Motivation

In addition to strategies for easier learning presented in Section 2.2, it is equally important to create an engaging environment supporting students' motivation. All strategies for supporting motivation are based on fulfilling some human needs [57]. Table 2.2 links the needs to common strategies.

Table 2.2: Needs and strategies that help to fulfill these needs.

Needs	Strategies
Purpose	Emphasizing usefulness of the programming skill.
Progress, learning	Skills visualization, points, levels.
Effectiveness	Recommending tasks of optimal difficulty.
Autonomy	Allowing to choose a topic or a task.
Recognition	Badges, leaderboards.
Sharing	Possibility to share programs or achievements.
Cooperation	Pair programming.
Beauty, harmony	Appealing game world.
Fun	Entertaining tasks.
Creativity	Open-ended tasks, projects.

2.3.1 Appealing Game World

Ideally, game world should appeal to students – even without tasks to solve, it should be an attractive toy to play with [58]. That is why all Hour of Code tutorials are based on movies and games popular among children, such as Angry Birds, Frozen, or Star Wars (Figures 2.2b and 2.3a). In such environments, it is possible to assign open-ended tasks, or even let the students create whatever they want, which works well especially for creative students seeking for self-expression. For example, Khan Academy programming curriculum contains many open-ended drawing projects (Figure 2.3b).

2.3.2 Entertaining Tasks

For many students, giving them specific small problems works better than large, loosely defined, open-ended tasks. By solving small problems quickly, they get a feeling of progress and learning. Another advantage of closed tasks is a more straightforward implementation of gamification features and adaptive behavior.

Small closed tasks result in short programs, but their behavior should be still interesting. To achieve complex behavior, a system can either provide students with macro-commands (e.g., to draw a circle) or with a skeleton of complex code, with a few gaps to fill in by students. However, it is important for the students to feel ownership over the code, which is particularly a concern with the code skeleton. A solution, implemented in Blockly Games, is to make a series of tasks in which the students build on their program from the previous task [23].

2.3.3 Optimal Challenge

For a great learning experience, the difficulty of the task must match the skill of the student. If the task is too easy, the student is not challenged and gets bored. If the task is too difficult, the student becomes frustrated and desperate. On the other hand, if the task has appropriate difficulty, the student is likely to be challenged and focused (Figure 2.4). The complete immersion into the task the student is solving at the moment is called a state of flow [13], or a *zone of proximal development* [62]. Achieving the state of flow maximizes the learning outcome [8]. Chapter 3 describes techniques for estimating student's skills and recommending tasks of the optimal difficulty.

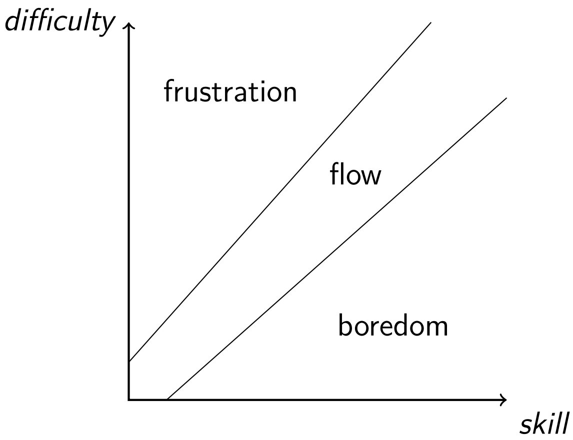
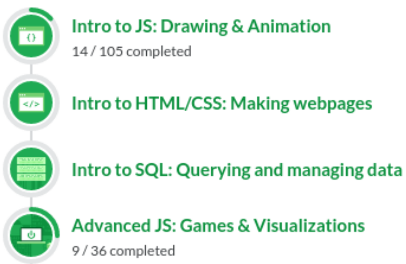


Figure 2.4: Relationship between difficulty and skill. Too easy tasks bore students, too difficult frustrate them. For efficient learning, the difficulty of the task and the skill of the student should be in balance.

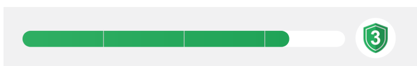
2.3.4 Gamification and Progress Visualization

A sense of progress and learning can be boosted by visualizations of solved tasks and completed problems sets, acquired skills, and progress towards mastery in the current topic. Figure 2.5 shows examples of progress bars from various systems. Although the programming tasks themselves can be considered as a game, most learning systems add further gamification elements to increase the sense of progress. Common gamification elements include points, levels, badges, and leaderboards.



(a) Khan Academy

(b) Hour of Code



(c) Umíme Programovat

Figure 2.5: A few possible forms of a progress bar.

3 Adaptive Learning

> *If it is possible that a student can effectively learn something in 10 seconds, it shouldn't take 11.* (T. Butt)

Adaptive Learning Systems optimize the efficiency of the learning process using techniques of artificial intelligence. Adaptivity is not necessary for a learning system to be useful[1]; however, it can significantly improve the learning. Artificial intelligence can be used, for example, to personalize the sequence of tasks [27], to visualize skills [7], to automatically generate hints [26], or to analyze collected data and get an actionable insight [49].

Adaptive learning systems have been already successful in some domains. For instance, Map Outlines[2], developed by the Adaptive Learning group at Masaryk University, is an intelligent web application for learning geography. It has been used by tens of thousands of students, and online experiments have confirmed that the adaptivity of the system helps to improve the learning outcome [41]. From the existing systems for learning introductory programming, Problem Solving Tutor (Section 2.1.2) can be considered as an adaptive learning system, since it recommends next tasks to solve after completion of a task.

Typical adaptive learning system consists of the following parts [60]:

- *Domain model* describes tasks and concepts, including relationships between them. It can also include knowledge of an ideal student (rules for solving tasks), and misconceptions (flawed rules) (Section 3.1).
- *Student model* describes past performance of a student on all tasks, estimates concept skills, and predicts future performance (Section 3.2).
- *Tutor model* (instructional policy) decides what should the student do next, e.g., which topic to practice, which task to solve (Section 3.3).
- *User interface model* describes how the domain, student, and tutor models are presented to the student (e.g., task environment, an overview of problem sets, achieved skills, recommended tasks) (Section 3.4).
- *Analysis layer* monitors and evaluates system behavior, allows performing AB experiments, and manages online training of model parameters. In addition to these online (inside-the-system) analyses, offline (outside-the-system) analyses are often performed as well to discover useful interventions to the system (Section 3.5).

---

1. Indeed, most of the systems presented in Section 2.1 are not adaptive.
2. Available at `https://outlinemaps.org`.

3.1 Domain Modeling

The domain model describes educational content, concepts, and relationships between them [59]. It is used in student models to provide structure for student skills, in tutor models to filter tasks containing a chosen concept, in the user interface to group tasks into problem sets, and in analyses, for example, to compare each task to the other tasks in the same problem set.

3.1.1 Learning Objects

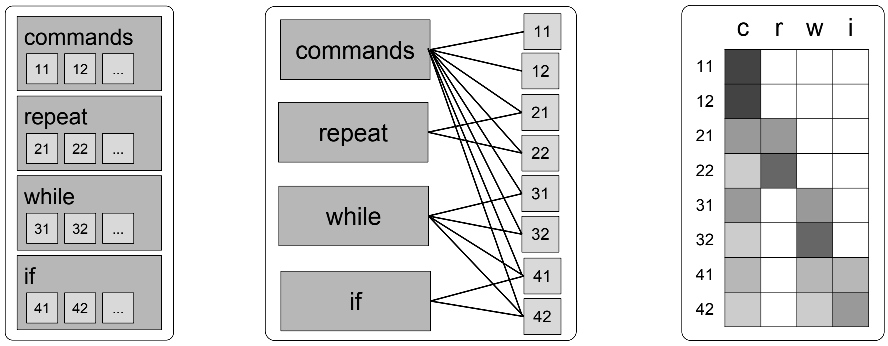
Entities in the domain, *learning objects* [11], include not only tasks, texts, videos, interactive visualizations, and other educational content, but also problem sets, concepts, and misconceptions. Learning objects are associated with content attributes (e.g., task setting) and parameters computed from performance data (e.g., median solving time). Typically modeled relationships between learning objects are generalization (e.g., programming is a more general concept than loops), inclusion (e.g., a problem set contains tasks), prerequisites (e.g., single loops are prerequisite for nested loops), and similarities (e.g., how similar are tasks with respect to their difficulty).

3.1.2 Concepts

Concepts, or *knowledge components* [1], are features of tasks that correspond to learnable skills required to solve the task. *Concept-free* models assume that all tasks belong to a single indivisible concept. In introductory programming, this assumption does not hold; for example, it is possible to master loops while not knowing functions and vice versa. *Concept-aware* domain models enable to build richer, more accurate student models.

Depending on the multiplicity of the concept-task relationship, concepts are either *disjoint* (1:m) or *overlapping* (n:m) (Figures 3.1a and 3.1b). Overlapping concepts can be represented as sets of tasks, a bipartite graph, or a feature matrix (*Q-matrix* [34], Figure 3.1c), where rows correspond to tasks, columns to concepts, and binary values denote whether given task contains given concept. In case of *soft membership*, the sets are fuzzy, the bipartite graph has edge weights, and the values in Q-matrix are continuous from the range $[0, 1]$.

Concepts can be either set manually or detected automatically [38, 54]. To use collected data while keeping the interpretability of manually defined concepts, automatic techniques can be used to suggest small improvements to the current mapping of tasks to concepts [59, chapter 3].

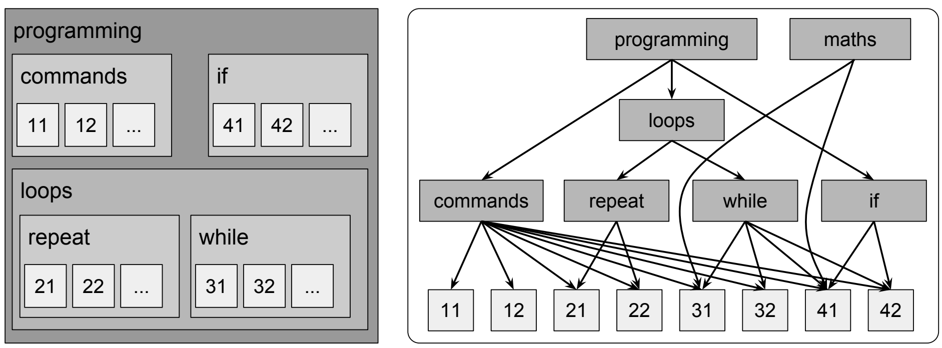


(a) Disjoint sets (b) Overlapping concepts (c) Q-matrix

Figure 3.1: Relationship between tasks (light squares) and concepts (dark rectangles). Darkness in the Q-matrix shows the strength of the relationship.

3.1.3 Hierarchical Models

Concepts vary in granularity and create hierarchies. For example, programming concept includes loops, loops include while loops, and while loops include programs with a single while-not-end loop. The hierarchy can be modeled either as a rooted tree (single parent concept allowed) or as a directed acyclic graph (DAG) (Figure 3.2). Although hierarchical concepts can be used without modeling the hierarchy (using overlapping concepts, e.g., a task can contain while loops and a more general loops concept), explicitly modeling the hierarchy of concepts in the introductory programming can improve the accuracy of the overlaying student model [25].

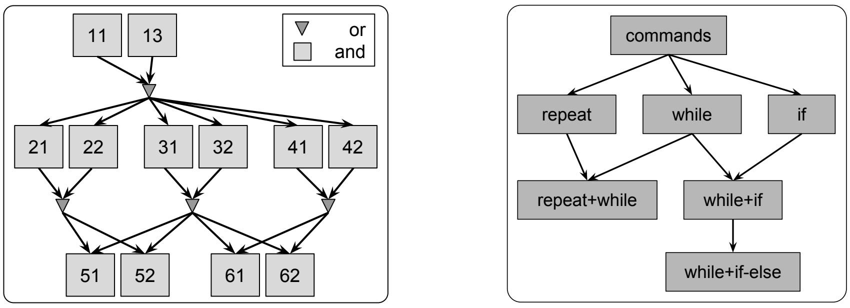


(a) Tree of concepts (b) DAG of concepts

Figure 3.2: Hierarchical domain models.

3.1.4 Prerequisites

Prerequisites specify which learning objects need to be mastered prior to tackling a given learning object. For example, *single loops* are a prerequisite for *nested loops*. There are two conventional interpretations of multiple prerequisites: *conjunctive prerequisites* (all must be mastered) and *disjunctive prerequisites* (one mastered is enough). Conjunctive and disjunctive prerequisites can be represented together by *and-or graph* (Figure 3.3a). If only conjunctive prerequisites are needed, then it is enough to represent prerequisites by a DAG [15] (Figure 3.3b). More generally, each learning object can specify a function from skills of its prerequisites into a prior skill (and leave the *ready-to-tackle decision* on a tutor model). If the skills can be interpreted as probabilities, then the resulting prerequisite structure is a *Bayesian network*; this approach was used in [9].

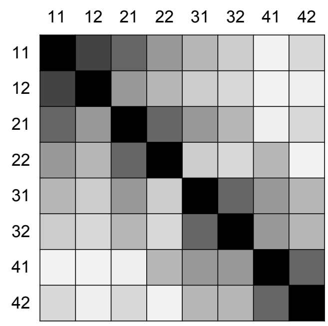


(a) And-Or graph of tasks (b) DAG of concepts

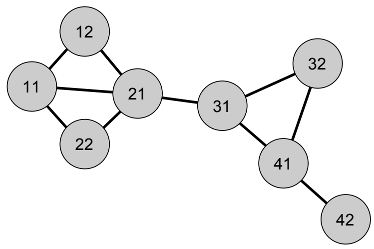
Figure 3.3: Domain models with prerequisites.

3.1.5 Similarities

Similarities can be represented either as a matrix of values from the range $[0, 1]$ (the higher number, the more similar objects), or as a graph with edges between the most similar learning objects (Figure 3.4). In the *network model* [54], task similarities are used to propagate knowledge about one task (observed performance) to other tasks (estimated performance) directly, without using concepts. The most crucial decision when computing similarities is what data to use [51]. The decision depends both on the intended usage and on the amount of collected data available. For example, performance data are useful if the similarities are used in a student model to predict future performance; however, it requires that enough performance data is collected in order for the computed similarities to be stable.



(a) Similarity matrix



(b) Similarity graph

Figure 3.4: Representation of soft and hard similarities between tasks.

3.2 Student Modeling

The student model captures the state of the student (e.g., skills) estimated from the past performance and it usually enables to predict future performance [14, 45, 60]. It is used in tutor models to detect mastery, or to select tasks matching student's skills, in the user interface to visualize skills [16], and in the analyses to get insight into how the students are learning.

3.2.1 Classification of Student Models

A lot of diverse student models were proposed, to cover many possible scenarios based on the available data and usage of the model. The following are the main properties that differentiate the models.

- *Modeled components.* In addition to modeling skills, broader student models can also include emotions (e.g., frustration, boredom), needs, preferences, motivation, and metacognition [4, 39]. In this thesis, we restrict our attention to the models of skill.
- *Underlying domain model.* The simplest models assume a single concept, more complex models can include multiple concepts, hierarchies, prerequisites, or similarities (Section 3.1).
- *Time scale of modeled events.* Some models track the state of the student during solving a task [12, 36], while others describe the state of the student only between tasks [32].
- *Discrete or continuous skills.* Some models assume discrete skills (e.g., known/unknown, or with multiple levels), while others assume continuous skill [45].

- *Representation of skill estimate.* Instead of modeling full probability distribution of the skill, most models track only a single point estimate (possibly with another parameter describing its uncertainty).
- *Representation of performance*. Performance can be represented as binary correctness (solved or failed), discrete performance levels (e.g., poor, good, and excellent), continuous performance (in the range $[0, 1]$), or (log-transformed) solving time.
- *Predictive or descriptive*. Predictive models can predict the performance of the student on a given task, while descriptive models only summarize the past performance of the student (e.g., a model can count the number of consecutive tasks solved with good performance).
- *Online or offline.* In online models, parameters can be updated gradually after each task session (i.e., there exist an efficient online algorithm for learning model parameters).
- *Assumptions about learning.* Models used in *adaptive testing* [63] assume no learning. Models that include learning use additional assumptions about how the learning happens. For example, some models assume constant skill increase after each solved task.

In Sections 3.2.3 and 3.2.4, we describe two broad families of predictive models of student's skills between tasks, focusing on their versions that can incorporate multiple concepts, learning and allow for online update.

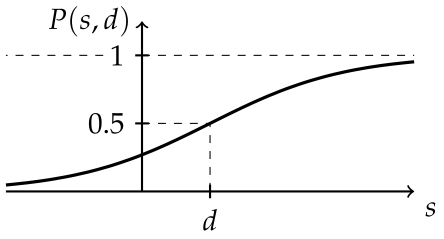
3.2.2 Skills and Performance

Student models use a domain model as an underlying structure, assigning estimated *skills* to each concept, and *performances* (either observed or predicted) to each task. Similarly as skills summarize all previous task sessions of the student, performance is a compressed information about the series of interactions with the task (code edits, executions, taken hints, user rating).

Performance measurement can range from simple heuristics based on summary statistics (e.g., specifying solving time thresholds for excellent and for good performance), through more complex algorithms that consider all interactions with the task, to a recurrent neural network reading the series of program snapshots embeddings [64]. While the compression causes a loss of information, it also reduces noise present in the raw data. Moreover, it helps to avoid a combinatoric explosion of considering each type of model with multiple different types of input data and their combinations. The most important consideration for performance measurement is which data to use (e.g., correctness, solving time) [49].

3.2.3 Logistic Models

Logistic models [42] predict performance $P(s,d)$ of a student with skill s on a task with difficulty d using a logistic function applied to the difference between the skill and the difficulty[3] (Figure 3.5).



$$P(s,d) = \frac{1}{1+e^{-(s-d)}} \quad (3.1)$$

Figure 3.5: One-parameter unidimensional logistic model.

Elo model [44, 30] extends the logistic model to capture changing knowledge. Inspired by the rating of chess players [17], the model interprets each attempt to solve a task as a match between the student and the task. After this match ends, skill and difficulty estimates are revised. Depending on whether the student solves the task better or worse than expected by the model, her skill is increased or decreased. The size of the change is proportional to the prediction error, which is the difference between the predicted and the actual performance ($p_{ij} - \hat{p}_{ij}$). Elo can be further extended to utilize more complex domain models, such as multiple concepts with a hierarchy, or similarities between tasks [54].

3.2.4 Dynamic Bayesian Networks

Another family of student models is based on *Bayesian networks* [43], in which nodes correspond to skills and performances, and edges are given by the underlying domain (e.g., prerequisites). In *dynamic Bayesian networks* (DBN) [37] nodes also depend on their own previous value. Each node is assigned a *state function* mapping possible states of its parents to the probability distribution of its values.

Bayesian Knowledge Tracing (BKT) [12] is a widely used special case of DBN, which makes several assumptions to allow for a simple online update: disjoint concepts, binary skills, and constant rate of learning. With these assumptions, each skill can be tracked by a separate *Hidden Markov Model* [5]. However, BKT is too restricted for the domain of introductory programming,

3. Some logistic models do not model difficulties (i.e., $d = 0$).

where skills are overlapping and not binary. Other models try to find a balance between BKT and general DBN. For example, the state function in DBN can be restricted to a specific form [14, section 5.1.2], domain of each node can be restricted to a small number of values (e.g., learned, ready to learn, not ready to learn) [9], or BKT can be combined with logistic regression to estimate the contribution of multiple skills [69].

3.2.5 Evaluation of Student Models

Ideally, student models should be evaluated with respect to their intended usage. For example, if the student model is used as a component for task recommendation, then the model that leads to the best recommendations is preferred. However, evaluating the impact of the student model is difficult, so proxy quality measures are used instead, such as accuracy of predictions, [46, section 3], *reliability and resolution* [46, section 5], and *plausibility* of parameters [25]. Although better predictions often have a positive impact on the application, it is not guaranteed [24], so it is safer to observe several proxy measures, instead of blindly maximizing one of them.

3.3 Tutor Modeling

Tutor models (a.k.a. *instructional policy*) [39, ch. 7] are used to help students with learning. They support teachers by making decisions that are difficult for people to do optimally, such as assigning the most suitable task to each student individually.

According to modeled decisions, tutor models can be divided into two categories: *problem selection tutors* (*outer loop*, *macroadaptivity*) for decisions between tasks, and *problem-solving tutors* (*inner loop*, *microadaptivity*) for decisions within a task [60]. Problem selection tutors can be further decomposed into *mastery decision* (Continue to practice the current topic?), *curriculum sequencing* (Which topic to learn next?), and *task recommendation* (Which task to solve next?). Problem-solving tutors decide when and which hint to show to the student. In this thesis, we restrict our attention to the outer loop decisions, which is the area where the computers perform best [19].

Existing systems and research focus on *reflexive tutors*, which provide a single decision with no planning. This approach is suboptimal. Consider a problem set with only two unsolved tasks left; neither of them is enough to master the topic. If the more difficult task matches the skill slightly better, then the reflexive agent first selects this, and then the easier task. *Planning tutor* would prefer more natural ordering, starting with the easier task.

Most tutor models can be viewed as *recommendation algorithms*, where items are either problem sets, tasks, or hints. An exception is mastery decision which is better formulated as a classification problem. While training tutor models can be approached as a *supervised learning problem*, obtaining labeled training data is not straightforward. *Reinforcement learning* [61] allows learning of action preferences using future rewards (derived, e.g., from the estimated time spent in flow) [10].

3.3.1 Mastery Learning

In mastery learning [56], each topic is practiced until it is mastered, and only then student moves to the next topic. Typically, mastery is decided by comparing skill estimated by a student model (or predicted probability of good performance) to a threshold. If the practiced problem set is homogeneous (all tasks have approximately same difficulty and cover the same concepts), then it is sufficient to use a concept-free single-difficulty descriptive student model, such as *Exponential Moving Average* of past performances [49].

The right value of the threshold depends not only on the used student model and its hyperparameters, but also on the application, which determines desired balance between over-practice and under-practice [20, 18]. *Effort-score* plot and metrics [49, 24, 25] help to choose the threshold. Alternatively, the threshold can be calibrated by requiring a specific time or number of tasks for specific students (e.g., exactly three tasks for a genius who solves each task with a high performance).

3.3.2 Curriculum Sequencing

Selecting the best task from the pool of all tasks in the system can be decomposed into two subproblems: (1) selecting a problem set, and (2) selecting a task from this problem set. This decomposition makes the problem easier to handle because both subproblems can be solved reasonably well even by simple heuristics. Furthermore, it ensures coherence of recommended tasks; students know what they are learning, and they do not need to adjust to a very different setting (e.g., toolbox) with every task.

Instead of selecting a single recommended problem set, some systems may instead need to filter, rank, or score all problem sets. A planning tutor can even generate a *learning path*, which is a sequence of problem sets that lead to acquiring a requested knowledge [9].

Problem set selection focuses on the prerequisite structure. If the number of problem sets is reasonably small, students often want to solve all of them, in which case it is preferable to recommend the problem sets from the easiest to the most difficult. If the ordering is not total, one of the problem sets with satisfied prerequisites can be recommended. If the number of problem sets is large, it may be useful to use a complex student model and choose a problem set with difficulty matching student's skills. An alternative approach is to describe which problem sets naturally *follow* another and recommend those that follow the one which was just solved.

3.3.3 Task Recommendation

Task selection is a problem of choosing a single best task from a given problem set for a given student. The problem is usually solved in two phases: First, filters (*hard constraints*) are applied, such as avoiding already solved tasks, avoiding the task which was just given up, and not violating prerequisites. Second, for all remaining tasks, a preference is computed based on several criteria (*soft constraints*), such as optimal difficulty (how well does the task match student skills), a balance between diversity and coherence (avoiding too similar tasks as the previous one, especially if the student solved the task easily, but also avoiding too large changes), and exploration (value of information gained by observing the student solving this task).

If the problem set is homogeneous, then all tasks should have similar preferences, in which case it is safe to select the task *uniformly at random*, maximizing exploration. Otherwise, a typical approach is to manually select an error function for each soft constraint, compute a *weighted sum of errors* for each task and select the task with the smallest error [40].

There are three common ways how to balance exploration and exploitation. The first approach is to incorporate the uncertainty about the task into the selection preference, second is to select the task at random with the probability proportional to its preference, and third is to choose the best task with probability ϵ, and a random task otherwise (*ϵ-greedy strategy*) [61].

3.3.4 Evaluation of Tutor Models

Tutor models can be evaluated either online (in a running system using that model), or offline (using collected data from a system using a different model). Offline evaluation of tutor models is difficult due to missing data. If the evaluated algorithm recommends a different task than the student attempted at that point, there is no easy way to tell if the recommendation is correct. The degree of data sparsity makes offline evaluation so challenging, that many adaptive learning systems skip this type of evaluation entirely, apply offline evaluation only to student models (Section 3.2.5), and check the quality of the deployed tutor model on the live traffic (Section 3.5.3). Nevertheless, realizing that the offline evaluation serves just as a limited proxy metric, it can be still useful to perform at least some offline evaluation, e.g., to limit the number of models to evaluate online to just a few.

Evaluation of recommendations accuracy requires to define which recommendations are correct. For example, a *suitable task* should result in neither too low nor too high performance. An ideal recommender would recommend all of the suitable tasks and none of the unsuitable tasks. The error can be quantified in many ways, e.g., by *RMSE* or *AUC* [53, section 8.3.2].

Unfortunately, collected data are highly biased by the adaptive behavior of the system and by the self-selection bias [50]. Strategies to avoid the bias in the collected data include using only random recommendations for the evaluation (decreases the amount of data), online evaluation (time-consuming), or evaluation using simulated students (requires strong assumptions).

In addition to the accuracy of recommendations, further properties of tutor models can also be explored, e.g., robustness, diversity, composition (balance between the introduction of new concepts and practicing the old ones) [65], or the amount of effort needed to learn new concepts [24].

3.4 User Interface

The user interface (UI) is also a common subject of personalization. UI provides visualizations that are based on the domain model (e.g., an overview of problem sets), the student model (e.g., visualization of skills [7]), or the tutor model (e.g., displaying recommendations). All these visualizations can be personalized using information from the student model [59, chapter 9]. For example, task stories can use student's favorite characters.

UI influences what are the best underlying models and how the collected data should be interpreted. For example, optimal task recommendation algorithm can be different if instead of a single enforced task, all tasks in the current problem set are displayed ordered by their score. Adaptive learning systems can model UI explicitly in order to compare different versions of UI in an online AB experiment, which allows to investigate the influence of various aspects of the UI, such as that different types of programming blocks are colored differently, or using text instead of blocks [66].

3.5 Analysis Layer

Analysis layer allows for *iterative improvement* of the adaptive learning system. To decide which interventions improve learning and motivation, suitable metrics must be measured. The system cannot be perfect from its first deployment, and its quality increases with the number of small incremental improvements performed, so it is beneficial to evaluate and update the system frequently. This *rule of the loop* [58] was observed to hold for adaptive learning systems [3, 59], as well as for recommender systems in general [70, Rule 16]. Frequent, high-quality evaluation is achieved by a combination of online AB experiments, live monitoring, and offline analyses.

3.5.1 Mission Statement

At first sight, the mission of a system for learning programming is clear: long-term increase in algorithmic problem-solving skill in the population. However, other factors than the skill should be considered as well. For example, how much students enjoy the time in the system, how they are satisfied with their accomplishments, and if they are motivated for further learning of programming. As a solution to this *multiple objectives dilemma* [70, Rule 39], the mission can be formulated as achieving a balanced increase in all of these important factors. Such formulation reminds developers of the system not to overfocus on one factor at the cost of the others.

3.5.2 Long-Term Objectives

Mission statements are not measurable. However, measurable metrics are needed to make informed decisions, such as which of two recommendation algorithms to prefer (e.g., after conducting an AB experiment). While being precise and measurable, these proxy metrics should still be related to the mission as much as possible.

The mission statement can be refined into the following long-term objective: “maximize the number of students who mastered elementary programming while having fun.” Next, we need to define terms used in this objective in such a way that they can be measured. While “achieving mastery in elementary programming” can be formulated as an objective criterion (e.g., the student solved at least three tasks for each concept with at least good performance), “having fun” requires the system to ask students about their subjective feelings. To avoid subjective ratings, the system can initially use simpler long-term objectives, which are also easier to measure and interpret (examples in Table 3.1).

Table 3.1: Examples of long-term objectives.

Metric	Description
Daily Active Users	students who have solved at least 1 task this day
Returning Users	students who solved a task in 2 different days
Converted Users	students who finished all levels in the system
Daily Solved Tasks	tasks solved in total by all students
Daily Flow Tasks	tasks solved with not too low/high performance

It is not clear which of these metrics is the best proxy for the mission. Fortunately, at the beginning of optimization process, all the metrics which reflect the mission tend to improve simultaneously, no matter which one is chosen to be directly optimized [70, Rule 12]. Nevertheless, it is useful to measure and report all of them from the beginning, to understand how their values are influenced by changes in the learning system [70, Rule 2].

Limitation of these simple metrics is that they only capture *engagement*, not learning. In experiments, learning is measured by a post-test, but adaptive learning systems need to evaluate students continuously to collect data about all students and not only those that stay in the system and are willing to take a post-test. For example, the system can assign a randomly selected task after every 20 minutes of practice, and measure the average performance. To avoid frustrating by too difficult tasks, only tasks that the student already have skills for can be considered for the random selection, weighting the observed performance by the difficulty of the chosen task.

3.5.3 Live Evaluation

New versions of models are deployed from time to time with parameters learned from the recent data. The behavior of the new models is monitored to detect problems as soon as possible, without waiting to evaluate an AB experiment. For this purpose, *online attributable metrics* that can be linked immediately to the tutor model actions are needed. These metrics are often formulated as a question concerning a single recommendation, such as "Did the student solve the recommended task?", "Did she solve it with a good performance?", "Did not she mark it as too easy or too difficult?". Each recommendation can be then attributed an error, which is either 0 or 1, depending on whether the statement is true or false. Finally, individual errors are aggregated, and the average error is reported.

3.5.4 Offline Evaluation

Offline experiments use historical data to avoid the cost of the live evaluation. They allow reusing the same data to evaluate several models, to obtain the results quickly, and to avoid negative impact on students if the evaluated model is poor. Offline evaluation is typically used for model selection (e.g., hyperparameters search), and as a check before deploying a new model.

The specific approaches to evaluation of student and tutor models and related challenges were discussed in Sections 3.2.5 and 3.3.4. A common consideration for an unbiased offline evaluation is to ensure that the model is tested on different data than on which it was trained. Unlike in many supervised learning problems, events from learning systems cannot be divided into train and test set randomly since that would lead to predicting past from the future. The evaluation should mimic as close as possible the real scenario of how the trained model will be used, which leads to a methodology called *online evaluation with generalization to new learners* [45]. If generalization to new tasks or problem sets is desirable, then the data should be split accordingly [14].

3.5.5 Simulation Experiments

Online experiments rely on live data, while offline experiments use historical data. Simulation experiments provide a third alternative, which does not require any collected data at all. Not needing collected data makes simulations widely applicable. On the other hand, researchers must be careful when interpreting the results of a simulation, as the simulation results are just consequences of used assumptions.

Nevertheless, simulations are perfect for model debugging, finding the best possible performance achievable by a model [20], exploring hypothetical scenarios where the collected data would be too biased and for which the ground truth does not exist (e.g., clustering of tasks into concepts [54, section 3.3]), or understanding which parameters are sensitive and should be learned from data, and which can be safely set manually (e.g., role of threshold and used student model in mastery learning [49]). Simulations can take advantage of historical data that are available. For example, in [16] an instructor dashboard is evaluated by combining collected performance data with simulated instructors interacting with the dashboard.

4 Design of Programming Game

A game is a problem-solving activity, approached with a playful attitude. (J. Schell)

Students learn programming by solving programming tasks. Therefore, tasks are the key component of a system for learning introductory programming, and the design of a programming game deserves careful attention. Similarly to the design of adaptive behavior, a design of a game is also an iterative process of prototyping and testing new ideas. Good programming games are a result of many gradual improvements (*rule of the loop*) [58].

The game should allow for tasks practicing all basic programming concepts, such as sequences of commands, loops, and conditional statements. To enable outer-loop adaptive behavior, multiple diverse tasks practicing the same concepts are needed, including many tasks using only sequences of commands without any advanced programming construct. To support engagement, tasks must be fun and immediately appeal to be solved.

We have designed a game which is a variation on a robot on a grid with a space theme, and which uses Blockly (Section 2.1.3) for building programs (Figure 4.1). These choices support all main strategies for easier learning of programming (Section 2.2), including avoiding syntax errors by using block-based programming and showing a visual output (the grid world). We combine several strategies to support motivation (Section 2.3), such as appealing game world, entertaining tasks, progressing through levels, and recommending tasks of optimal difficulty (adaptivity discussed in Chapter 5).

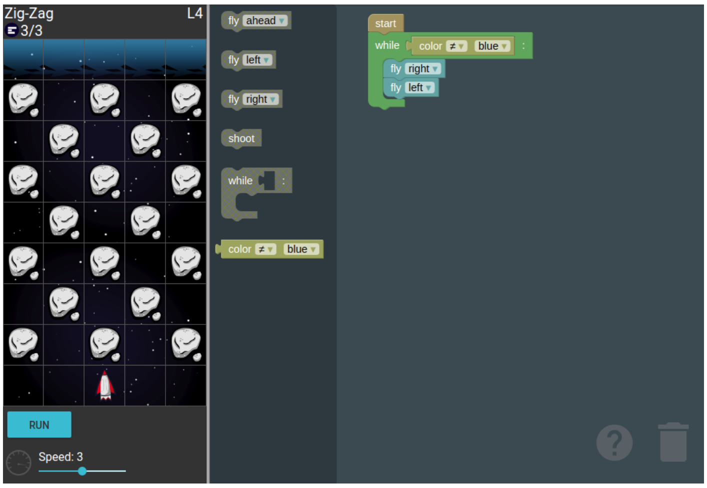


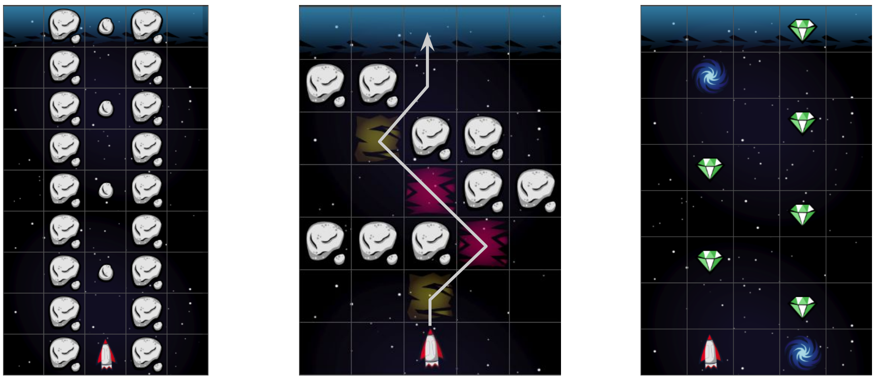
Figure 4.1: Example of a task with the space-themed grid world.

4.1 Game World

The game world itself should be a pleasure to look at and fun to play with, even without a specific task to solve [58]. We have based the game world on a popular choice of a robot on a grid, using a theme of a spaceship flying through space and collecting diamonds (Figure 4.2). Each field in the *space world* has a background color, which the spaceship can read and use for decisions (e.g., turning left on red fields). In addition to the spaceship controlled by the student, there are the following game objects: diamonds, that need to be collected, large asteroids, that destroy the spaceship if it hits them, small meteoroids, that can be destroyed, and wormholes that serve as teleports.

The spaceship starts on the bottom row, and it moves one row forward after any action. In addition to flying forward, left, and right, the spaceship can also shoot small meteoroids (Figure 4.2a). These four basic actions already allow for a diversity of simple tasks which only practice sequence of commands. The spaceship has two sensors, one for the color under the spaceship, and second for its horizontal position (column index). Having two different sensors allows for diverse tasks practicing conditions, including testing inequalities, and potentially also compound conditions.

A novel feature of the game is the default forward movement, which results in significantly shorter programs. For example, to fly around a stone, only two commands are needed instead of 8 (or 4 if the available commands include movement in any of the four directions without turning) (Figure 4.2b). Furthermore, as the spaceship is always facing up, a common left-right confusion [23] is mitigated. To avoid too long worlds, we introduced wormholes, that teleport the spaceship back, to reuse rows multiple times (Figure 4.2c).



(a) Shootable meteoroids (b) Path around asteroids (c) Diamonds, wormholes

Figure 4.2: Examples of Space Worlds.

4.2 Tasks and Programs

Each task asks the student to create a program in Blockly, that would guide the spaceship safely to the last row, collecting all diamonds on its way. Students can execute their current programs as many times as they need.

The set of available blocks is gradually expanding with increasing level (Section 4.3). This *toolbox standardization* is comfortable for task creators because it is enough to specify a toolbox only once for all tasks in the same problem set. It is also convenient for students since the available blocks are not changing chaotically with each task. The elementary tasks use only commands for actions (fly, left, right, shoot), while the more advanced tasks offer *repeat* loop, *while* loop, *if* and *if-else* statements, and tests for colors and position. No special test for the last row is needed since all tasks follow a convention of coloring the last row by blue color.

To simplify the future learning of a textual programming language, labels on blocks approximately match Python syntax. The most notable exception is the repeat block (`repeat 5`), where the Python equivalent (`for i in range(5)`) is more difficult to understand for beginners.

To force students to use loops instead of a long sequence of actions, a task can specify a *length limit* on the maximum number of statements in the program. Limit on the number of statements rather than blocks was chosen in order to make the limit a smaller number and thus the counting easier. (Another advantage is that this definition could be used for textual programming as well.) To force students to think when they need to shoot, a task can specify an *energy limit* on the number of shots.

Figure 4.3: Examples of tasks and corresponding solutions.

4.3 Level Design

RoboMission contains over 80 tasks divided into nine levels. The levels were initially created manually, based on the required blocks and an estimated difficulty. Later, after some performance data were collected, we further decomposed them into a two-level hierarchy to enforce prerequisites between tasks, and we moved a few tasks into levels that better match their observed difficulty for students.

Figure 4.4: Tasks overview shows tasks grouped by levels. The green tasks are solved, the orange task is recommended. RoboMission contains nine levels, each with about ten tasks.

As a motivational element that helps to reinforce the sense of progress, students receive credits for each solved task (Figure 4.5). After earning a sufficient number of credits, students progress to next level, which results in increased difficulty of recommended tasks.

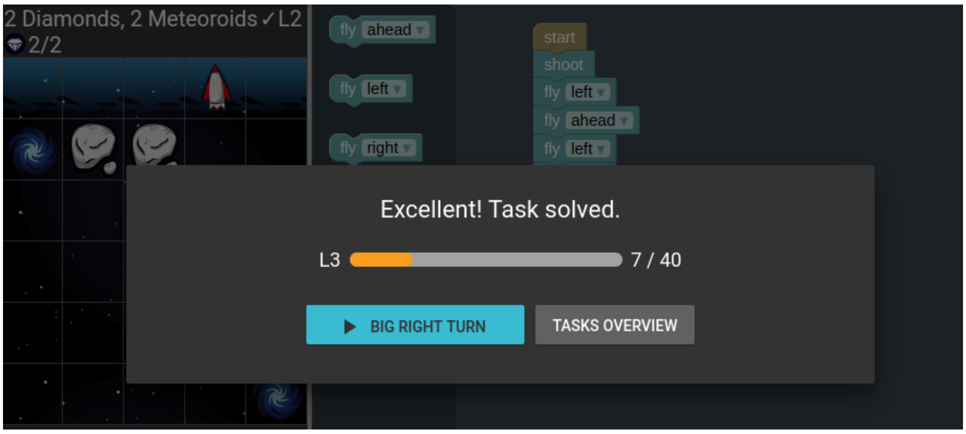


Figure 4.5: Students earn credits for each solved task.

4.4 Instructions and Explanations

Clear hints at appropriate moments is another strategy to support learning (Section 2.2.3). We combine simple adaptive instructions and reflexive post-event explanations. When a student encounters a new programming concept or a game element for the first time, the system displays a short instruction (Figure 4.6).

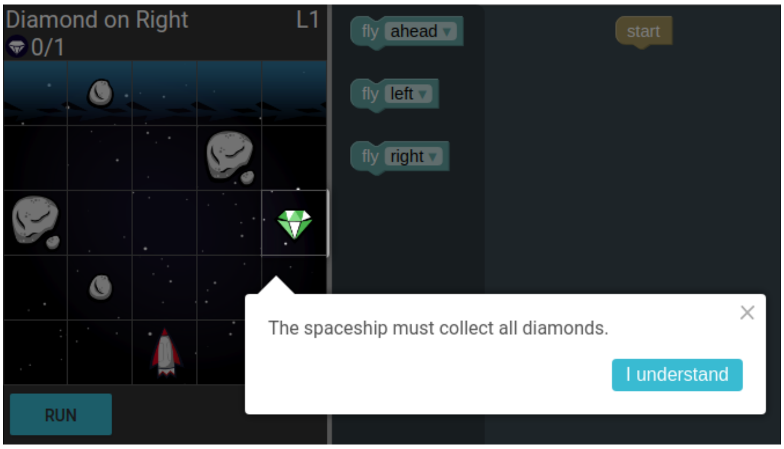


Figure 4.6: A mini-instruction for a new game concept.

The system also provides short explanations after each unsuccessful execution, describing why the task was not solved, e.g., some diamonds were not collected, or the spaceship has not reached the final row (Figure 4.7).

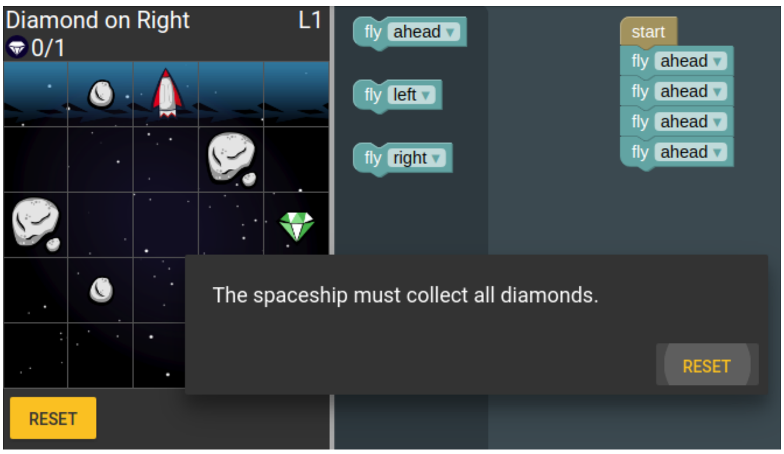


Figure 4.7: A mini-explanation of an unsuccessful attempt.

4.5 Game Editor

The online Game Editor (Figure 4.8) helps to create new tasks easily. In the editor, the author of a task can immediately see a visualization of the space world, test a solution in either Blockly or its text-based equivalent (Figure 4.9), and import or export tasks (using a custom Markdown-based format). The editor is public, so even students can create their own new tasks.

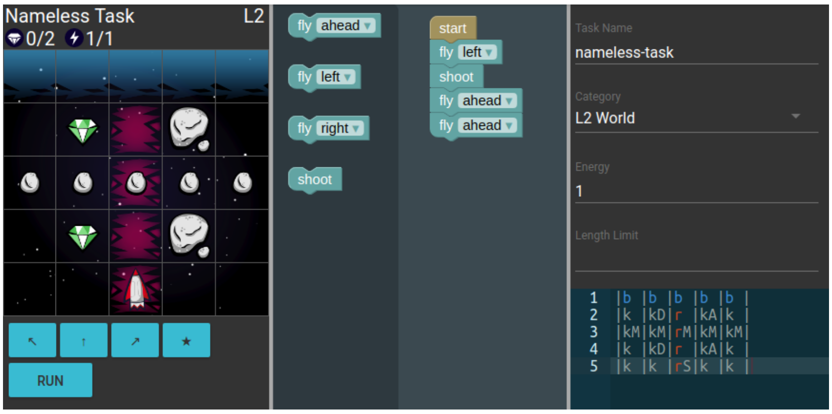


Figure 4.8: Game Editor includes a text editor for game world (bottom right), which uses an intuitive text representation described in Section 6.2.2.

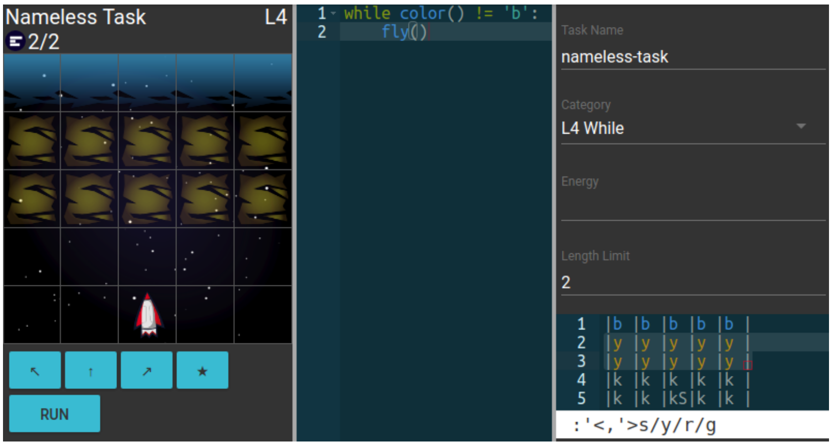


Figure 4.9: In Vim mode, Game Editor allows to perform some complex modifications, such as replacing all yellow fields on selected rows to red color, with a single command.

5 Design of Adaptivity

We are developing what one could flippantly call stupid tutoring systems: tutors that are not, in and of themselves, behaving in an intelligent fashion. But tutors that are designed intelligently, and that leverage human intelligence. (R. Baker)

Among the strategies to support learning and motivation, we focus on recommending tasks of optimal difficulty, which is the secret sauce of adaptive learning systems.

5.1 Expected Behavior

A wide range of difficulties, combined with the adaptive behavior, should make the system useful for everybody who wants to learn programming. However, currently the system targets children 10-15 years old. The main use case of the system is a 1-2 hour in-classroom tutorial. In line with the mission outlined in Section 3.5.1, goals of this tutorial are to teach the student basic concepts of programming, help them to spend the time in the state of flow, and motivate them for the further learning of programming.

After each solved task, the student is shown a recommended task and a link to the page with an overview of all tasks. Ideally, each recommended task would lead to the state of flow. As we cannot directly observe whether the student is in the state of flow, we must rely on proxy data. Currently, we only use objective attributable metrics based on observed performance (Section 5.3). From the general goals, we derive weaker, *necessary, but not sufficient* requirements, that can be more reliably observed, to guide us during the design of the system, and to provide us with more specific evaluation:

- All students are able to solve all tasks recommended by the system in a reasonable time (at most 15 minutes). Furthermore, they are able to solve the first few tasks quickly (each in 2 minutes), and progress to the second level in at most 10 minutes.
- The best-performing students progress through initial levels quickly, spending at most 5 minutes on tasks with only sequences of commands, and get to the more challenging tasks containing both types of loops and conditional statements in at most 20 minutes.
- At most one new programming concept and one new game concept appear in a task. Another concept is not introduced until the student solves a task with the last introduced concept with a good performance.

- All students should gradually start practicing all basic programming concepts (sequence, loop, conditional statement) during the first hour.

5.2 Domain Model

Currently, we do not model overlapping concepts, because that would require more data to properly analyze their interactions. Instead, we divide tasks into linearly ordered disjoint hierarchical problem sets (Figure 5.1). The hierarchy has two levels: the top-level problem sets (*levels*) contain about ten tasks, which are further split into three smaller problem sets (*phases*).

Levels and phases are ordered, while tasks within a given phase are not, since all the tasks in a single phase should be similarly difficult. The first two levels gradually introduce game elements (e.g., diamonds, meteoroids, wormholes), while the later levels usually focus on practicing one new programming concept (e.g., repeat loop, while loop, if statement).

The contribution of refining levels into phases is threefold: First, phases enforce critical prerequisites within a level (e.g., introducing wormholes before using them in more advanced tasks). Second, phases are approximately homogeneous, i.e. tasks in a phase have similar difficulty, which allows for simpler tutor models. Third, phases help to achieve a balanced composition [65]; many levels follow a pattern of introducing new concepts in the first phase, recombining them with previously learned concepts in the second phase, and further reinforcing known concepts in the third phase.

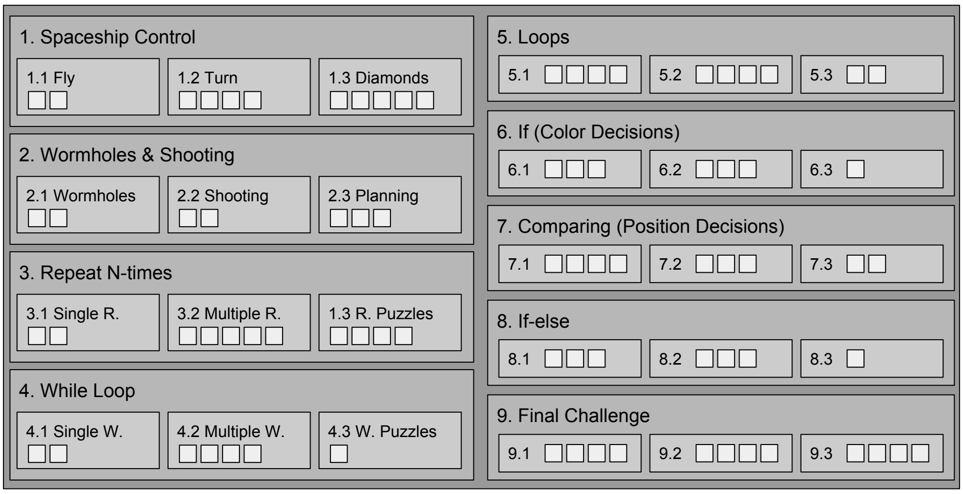


Figure 5.1: Domain model used in RoboMission: ordered hierarchical problem sets (dark rectangles) containing unordered tasks (light squares).

5.3 Student Model

For each student, the system keeps track of her skill for each problem set. The skill $s \in [0, 1]$ represents manifested ability to solve tasks in a given problem set. The initial skill is 0, and it is increased after each solved task from the problem set. Once the skill reaches 1, the problem set is solved.

While the model has a structure of a dynamic Bayesian network (Figure 5.2), it does not model the probability of the student solving a task with some performance, but rather an amount of verified skill; i.e., the model is only descriptive. An advantage is that this skill can be shown directly to the student, e.g., visualized as a progress bar towards completion of the current level, since it never decreases (which would be demotivating).

We use a discrete representation of performance with three levels (poor, good, excellent). Currently, only solving time is considered for the performance measurement: for each task, we define a threshold for a solution to be considered as good or excellent (1.5 and 0.75 multiples of median time).

After each solved task session, the corresponding skill is increased by an amount $p \in [0, 1]$ which depends only on the performance. The increment does not depend on the specific task solved, because we assume homogeneous phases. For each performance level, we set the increment such that $1/p$ tasks solved with such performance are enough to manifest mastery in this phase. Either a single task solved with an excellent performance, or two tasks solved with a good performance are enough to solve the phase, translating into $p_{excellent} = 1$ and $p_{good} = 0.5$. Furthermore, solving all tasks in a phase (even if with a poor performance), should be always enough to solve the phase. Therefore, the update rule is: $s \leftarrow \min(1, s + \max(p, \frac{1}{n}))$, where n is the number of tasks in the phase. To aggregate the skills of the phases into the skill of the level, we average the phase skills, which means that all phases must be solved in order to solve the level.

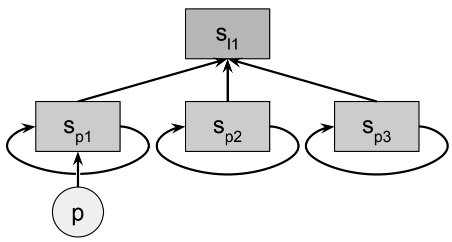


Figure 5.2: Student model used in RoboMission includes skill for each phase (computed from its previous value and a newly observed performance) and skill for each level (computed from skills of its phases).

5.4 Tutor Model

We focus on the outer-loop tutor modeling; concerning the inner loop, the system provides only a basic rule-based tutor for displaying instructions and explanations, as described in Section 4.4.

The outer-loop tutor is hierarchical; it first selects a problem set, then a task from this problem set. In order to assess if a problem set is mastered, the corresponding verified skill computed by the student model described in the previous section is compared to the threshold of 1.

The tutor selects for practice the first unsolved phase of the first unsolved level, using the total ordering provided by the domain model. The system contains only a small number of problem sets, so it is reasonable to assume that the student would like to solve all of them, in which case it is preferable to solve them in the order from the easiest to the most difficult. Homogeneity of phases allows to safely select a task uniformly at random from all unsolved tasks in that phase, which is a strategy maximizing exploration. Recommendations provided by the system are *soft*; i.e., students can ignore them and select any task from the overview of all tasks.

Progression through the problem sets guarantees a gradual increase of difficulty, but the increase is not monotonous. Perceived difficulty rises after progress to a new phase, but it decreases during solving tasks from the same phase, as the student's skill improves, while the objective difficulty does not change (Figure 5.3). On the one hand, it means that the difficulty does not match student's skills perfectly, possibly slightly overshooting at the beginning of a phase, and undershooting at the end. On the other hand, the perceived difficulty level is not same all the time; instead, it has a wavy character, which creates a more engaging experience for students [58].

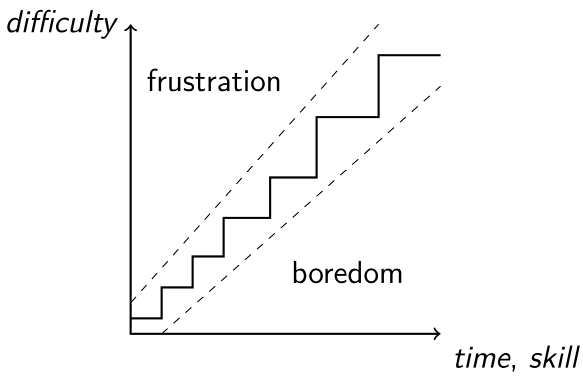


Figure 5.3: Dramaturgy of task difficulty in RoboMission.

5.5 Analysis Layer

In order to iteratively improve adaptivity, as well as the programming game and other aspects of the learning system, an analysis layer that provides several different views on the behavior of the system is crucial (Section 3.5). Our analysis layer includes the following components: Google Analytics, monitoring dashboard, error reports, user feedback, data exports, and logs.

Google Analytics shows the distribution of users with respect to space and time (Figure 5.4). In addition to page views, it can also process events sent from the frontend (such as clicking on the execution button), and divide them into groups, e.g., by the task being solved.

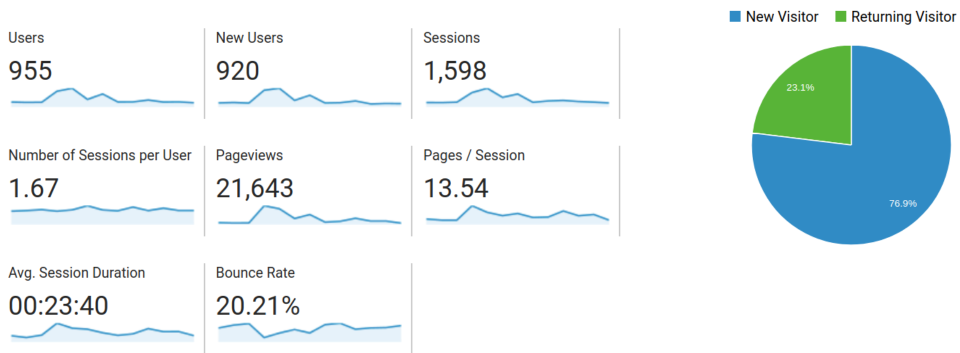


Figure 5.4: Preview from Google Analytics (January–March 2018) shows, for example, that an average user spends on the page over 20 minutes.

Monitoring dashboard shows several long-term objectives, namely daily active students, daily solved tasks, and solving hours (total time spent on successful attempts). The system also computes a few metrics for each task (solved count, median time, and success ratio) in order to detect issues with tasks. The dashboard is implemented as a weekly-recomputed Jupyter Notebook [31], that performs several analyses and creates visualizations using the latest data (Figures 5.5 and 5.6). It is easy to extend the dashboard simply by adding a cell in the notebook and testing it on historical data; no other modification to the backend or frontend is needed.

If an unhandled top-level error occurs on the server, it is not only logged but also sent to the administrators. Administrators also receive emails with messages provided by users via a feedback form that can be invoked on any page. Collected data are exported every week as a zip bundle containing CSV files prepared for offline analysis. (Structure of these CSV files is described in Appendix A.3.)

```
In [12]: metrics.active_students.resample('1W').mean().plot()
Out[12]: <matplotlib.axes._subplots.AxesSubplot at 0x7f9648d09048>
```

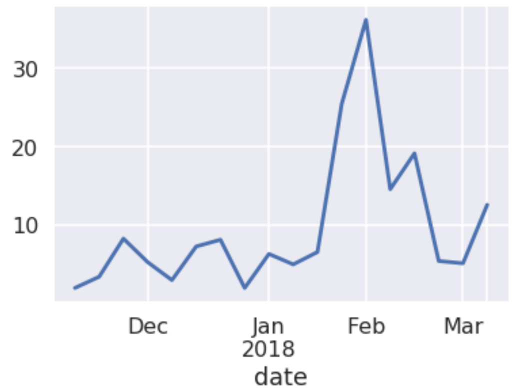


Figure 5.5: Number of students (users who attempted a task), weekly averaged, visualized in the monitoring dashboard (March 31, 2018)

.

id	name	n_attempts	success	time
1	moves	4530	94%	23s
2	world	2358	87%	92s
3	repeat	2039	87%	89s
4	while	1045	88%	72s
5	loops	1049	79%	123s
6	if	444	60%	177s
7	comparing	255	55%	218s
8	if-else	112	76%	153s
9	final-challenge	279	50%	288s

(a) Overview of all levels.

id	name	n_attempts	success	time
70	big-right-turn	156	96%	48s
71	big-left-turn	151	99%	55s
11	ladder	245	95%	58s
1	diamonds-in-meteoroid-cloud	185	89%	69s
76	blocked-wormhole	148	93%	87s
13	n	221	87%	89s
84	triangle	154	90%	100s
21	steal-the-nose	224	83%	106s
46	find-the-path	191	73%	123s
57	stairs	185	82%	132s
18	clean-your-path	179	72%	226s

(b) Level 3: Repeat loop.

Figure 5.6: Mean success and median time visualized in the monitoring notebook (March 31, 2018). This simple analysis reveals that the 7th level (Comparing) is probably more difficult than the 8th (If-else) and that Clean Your Path is significantly more difficult than the other tasks in the level.

6 Implementation of RoboMission

A programmer is ideally an essayist who works with traditional aesthetic and literary forms as well as mathematical concepts, to communicate the way that an algorithm works and to convince a reader that the results will be correct. (D. E. Knuth)

We present the overall architecture of RoboMission and describe aspects specific to the system for learning programming. We intentionally avoid discussing details of used technologies, which would make the text obsolete soon. An overview of technologies used in the project is included as Appendix B.

6.1 System Architecture

RoboMission has a client-server architecture with a fat frontend client communicating with the server via REST API [35]. In addition to the frontend app and backend services, there are two other parts of the system: scheduled jobs, which run periodically every week (e.g., metrics computation), and tools for offline analysis of collected data (e.g., Jupyter Notebooks).

The frontend is a single page application with a *redux architecture* [6, ch. 12], which means that a single immutable state stores all application data. A new state can be created only by dispatching an action. Each part of the state then defines its *reducer*, a pure function that takes a state and an action, and returns a new state. The view is then assembled using declarative reusable components, which are either *presentational* (e.g., describing how the game world is rendered), or *behavioral* (selecting data from the state, and dispatching actions) [21]. The application also defines a few asynchronous workflows (*sagas*), e.g., for program interpretation, and task solving process.

The backend is decomposed into modules defining database entities, their serializers (JSON for sending data to the frontend, CSV for exports), *view sets* describing REST API, and core modules with mostly pure functions for computing performance, skills, and recommendations.

6.2 Domain Representation

The domain is represented by a JSON file containing all problem sets and relationships between them, as well as their setting (e.g., which toolbox to use) and names of their tasks. Each task is then described in a separate file (Section 6.2.1). Furthermore, there is a JSON file containing parameters of domain entities (e.g., a time threshold for good performance for each task).

6.2.1 Task Sources

Each task is described by a single file in a markdown-based format [33], containing its name, setting, and solution. Task sources in markdown files have several advantages: they are human readable, each change is version-controlled, and the task can be edited easily in any text editor. Figure 6.1 shows a high-level grammar for task description, together with an example.

```
# <name>

## Setting
‘‘‘
<SpaceWorld>
‘‘‘

[- option: value]*


## Solution
‘‘‘
<RoboCode>
‘‘‘
```

(a) High-level grammar

```
# turning-left

## Setting
‘‘‘
|bM|b |b |bM|b |
|kA|k |kM|k |kA|
|k |k |kA|kM|k |
|kM|k |kS|k |kA|
‘‘‘

## Solution
‘‘‘
left()
fly()
fly()
‘‘‘
```

(b) Turning Left (rendered in Figure 1.1)

Figure 6.1: Task source grammar and an example.

Currently, there are only two setting options: length and energy limits. `SpaceWorld` and `RoboCode` fragments follow their own grammars, which are described in Sections 6.2.2 and 6.3.

6.2.2 Space World Description

Each Space World (Section 4.1) is described by a simple human-readable string. See Figure 6.2 for an example and description. Set of valid Space World descriptions is given by the following context-free grammar:

```
SpaceWorld -> Row+
Row -> '|'(Field'|')+ EOL  // EOL = End Of Line
Field -> Background Object*
Background -> 'r' | 'g' | 'b' | 'y' | 'k'
Object -> 'S' | 'A' | 'M' | 'D' | 'W'
```

```
|bD|b |bW|b |
|y |yD|r |rD|
|yD|y |rD|r |
|y |yD|r |rD|
|kW|kA|kM|kA|
|k |k |kS|k |
```

Figure 6.2: Example Space World with its description. Each line represents one row of the grid and is split by pipes ("|") into fields. Each field starts with a lower-case letter denoting color of the field (e.g., b = blue, k = black), followed by an optional upper-case letter denoting an object (A = asteroid, S = spaceship, etc.). For example, "rD" is a red field with a diamond.

6.3 RoboCode

RoboCode is a Python-like programming language for representing solutions in task sources. It closely corresponds to the text written on Blockly blocks with a few exceptions, such as abbreviated literals (e.g., `'b'` instead of `blue`), shorter function names and added parentheses (e.g., `left()` instead of `fly left`). It is also intended to be used in more advanced problem sets as the transitional phase from block-based to text-based programming. The language was designed to be simple for beginners, understandable without its previous knowledge, concise (short, but readable programs), and matching Python closely (for an easy transition to Python).

There are four basic commands: `fly()`, `left()`, `right()`, and `shoot()`. Each action is combined with moving one row forward. The movement takes place after the action, except for left and right turning actions, where the movement and the action happen simultaneously, i.e., the spaceship flies diagonally to the left or right.

Loops and conditional statements are same as in Python, with an exception of the repeat loop, which was simplified to be easier to understand:

```
repeat 4:
    fly()
```

Conditions are restricted to the following forms. (Color codes are same as in the Space World description, i.e. `'r'` is red, `'g'` is green, etc.)

```
position() [==|!=|>|<|>=|<=] [1..6]
color() [==|!=] ['r'|'g'|'b'|'y'|'k']
<test> [and|or] <test>
```

Control structures can be nested arbitrarily:

```
while color() != 'b':
    if position() == 1:
        right()
    if position() >= 4:
        shoot()
    fly()
```

6.3.1 RoboAST

While Python-like RoboCode is convenient for writing sample solutions, a more compact form would be better for logging, storing in DB and analysis. Secondly, a block-based presentation of the code is needed for students. Last but not least, a JavaScript equivalent of the code is useful for interpreting the code in the browser. Table 6.1 shows an overview of all representations.

Table 6.1: Different code representations used within the system.

Name	Form	Usage
RoboCode	text (Python-like)	sample solutions in task sources
MiniRoboCode	text (compact)	logging, storing in DB, analysis
RoboBlocks	blocks (Blockly)	code editor for students
RoboJS	text (JavaScript)	interpretation in browser
RoboAST	JSON (AST)	intermediate representation

To avoid implementing separate transformations between each pair of these representations, we introduced a common intermediate representation, *RoboAST*, which is an *Abstract Syntax Tree* [28] in JSON (Figure 6.4). For each of the four other representations, it is enough to implement its parser returning RoboAST object, and its generator from RoboAST (Figure 6.3). With a parser and a generator for each representation, any representation A can be transformed into any representation B by first parsing A into RoboAST and then generating B from this RoboAST.

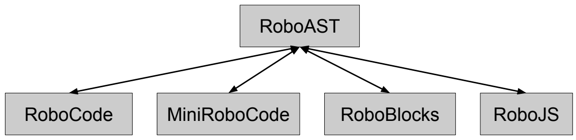


Figure 6.3: Transformations between RoboAST and other representations.

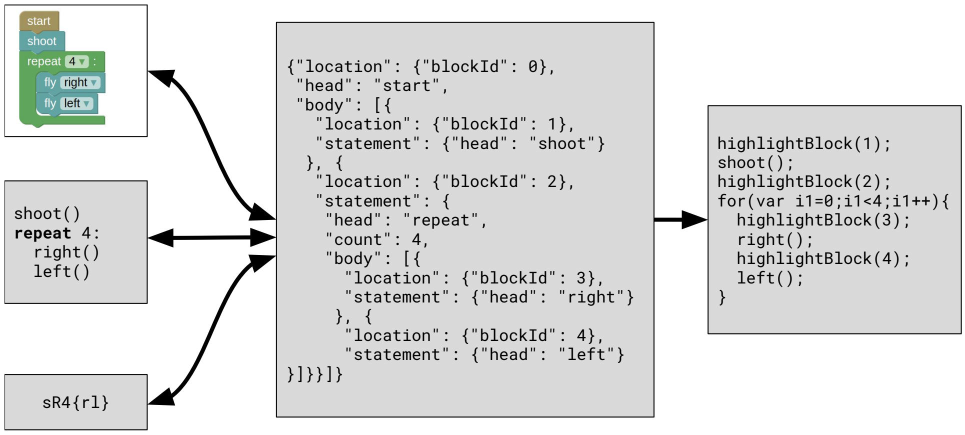


Figure 6.4: Example of a RoboAST (in the middle), and corresponding RoboBlocks, RoboCode, MiniRoboCode, and RoboJS.

6.3.2 Parsing Expression Grammar

For parsing RoboCode into RoboAST, we use a *parsing expression grammar* (PEG) [22]. PEGs are essentially unambiguous context-free grammars with prioritized rules and more compact syntax. The PEG implementation we use[1] allows to define a *match result* for each parsed subexpression, using an arbitrary JavaScript code returning a subtree of the final AST. For illustration, there are two examples of RoboCode parsing rules:

```
CompoundStatement = IfStmt / WhileStmt / RepeatStmt
WhileStmt = "while" __ t:Test ":" b:Body
  { return { head: "while", test: t, body: b } }
```

PEGs can be parsed in linear time, they do not require a separate lexer, and the resulting grammar is readable and concise; the PEG for RoboCode has only about 100 lines. However, a preprocessing of the RoboCode is necessary, because PEGs are context-free, while RoboCode is a context-sensitive language – the context is created by indentation. In addition, it is useful to

1. PEG.js, available at `https://pegjs.org`.

store line numbers alongside the statements to enable *meta-interpreting*, such as showing executed line, or linking errors to the point in the source code. Therefore, the preprocessing step transforms the code in a context-free form, in which each line of the code is prepended corresponding line number, and adding and removing indentation levels is denoted by { and } characters respectively. For example, the preprocessed code for the program shown in Figure 6.4 would be:

```
1| shoot()
2| repeat 4:
{
3| right()
4| left()
}
```

6.3.3 MiniRoboCode

MiniRoboCode is a condensed form of the RoboCode. It replaces indentation by curly brackets, keywords and functions by their first letters, and removes whitespace characters in order to fit programs into a single short line. The mapping from the RoboCode is described by the following rules:

```
repeat --> R
while --> W
if --> I
else --> /
position() --> x
== --> =
```

```
fly() --> f
left() --> l
right() --> r
shoot() --> s
color == 'y' --> y
color != 'y' --> !y
```

Two examples of complete transformations into MiniRoboCode:

```
fly()
while color() == 'b':
    left()
    right()
fly()
↓
fWb{lr}f
```

```
repeat 6:
    if position() > 1:
        shoot()
    else:
        right()
↓
R6{Ix>1{s}/{r}}
```

MiniRoboCode is useful not only for logging and storing programs in the database but also for many analyses because it is easy to process by counting letters or matching simple regular expressions, and because the codes are short enough to be used as labels in plots.

6.3.4 RoboBlocks

Blockly[2] is an implementation of a block-based programming environment from Google, which we use in the code editor for students (shown, e.g., in Figure 1.1). Blockly allows to import and export the currently assembled program in an XML format, that we call *RoboBlocksXML* (Figure 6.5). Both transformations between RoboBlocksXML and RoboAST are straightforward.

```
<xml xmlns="http://www.w3.org/1999/xhtml">
 <block type="start">
  <next><block type="shoot">
   <next><block type="repeat">
    <field name="count">4</field>
    <statement name="body">
     <block type="fly">
      <field name="direction">right</field>
      <next><block type="fly">
       <field name="direction">left</field>
      </block></next>
     </block>
    </statement>
   </block></next>
  </block></next>
 </block>
</xml>
```

Figure 6.5: RoboBlocksXML for the program shown in Figure 6.4.

As the RoboBlocks are used by children, it is important to use localized labels on the blocks. Depending on the language domain, we initialize Blockly blocks with the corresponding version of localized block labels (Figure 6.6).

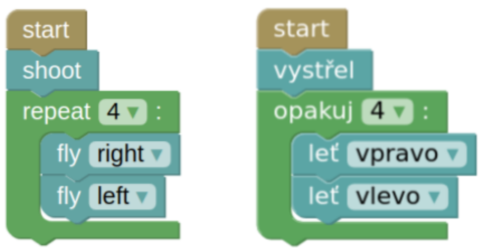


Figure 6.6: Same program in English and Czech localizations of RoboBlocks.

2. `https://developers.google.com/blockly/`

6.3.5 RoboJS and Interpretation

Instead of implementing our own interpreter of RoboCode, we transform the code through RoboAST into JavaScript and then use a JavaScript interpreter to run the code. The generated *RoboJS* assumes to be executed within a context providing hooks for functions corresponding to actions (`fly`, `left`, `right`, `shoot`), sensors (`color`, `position`), and meta-interpreting (`isStopped`, `highlightBlock`) (Figure 6.7).

The interpreter does not have to be necessary a *JavaScript* interpreter since it is easy to transform RoboAST to any other common imperative language. However, the interpreter must satisfy the following requirements: be implemented in JavaScript (to run in the browser), allow to define custom asynchronous hooks (to perform actions), and allow to step the code, or at least to impose a limit on the number of steps (to break infinite loops).


```
highlightBlock(1);
shoot();
highlightBlock(2);
for (var i1_ = 0; i1_ < 4; i1_++) {
  highlightBlock(3);
  right();
  highlightBlock(4);
  left();
}
```

Figure 6.7: Example of a RoboJS (right) for given RoboBlocks program (left). Each original command is accompanied by `highlightBlock(blockId)` so that the meta-interpreter knows which block to highlight.

We use an existing JavaScript interpreter[3] and wrap it into two additional layers: the first layer is a generator that yields actions, sensor requests and meta-interpreting effects (e.g., block highlighting), and the second layer is a saga that handles all these asynchronous effects. The separation into these two layers allows to test the interpreter logic easily without dealing with asynchronous effects.

3. `https://github.com/NeilFraser/JS-Interpreter`

7 Analysis of Collected Data

The goal is to turn data into information, and information into insight. (C. Fiorina)

In order to get insight into how the deployed system works and what should we focus on in next iterations, we analyze collected data. All analyses in this chapter use data collected during four months (10th November 2017 – 9th March 2018). This data, as well as the code performing the analyses, are available as attachments of this thesis (described in Appendices A.2 and A.3).

7.1 Data Description

During the four months, about 1000 users tackled at least one task, and 800 of them solved at least one task. About 100 students returned and solved another task another day. Figure 7.1 shows complete engagement curves.

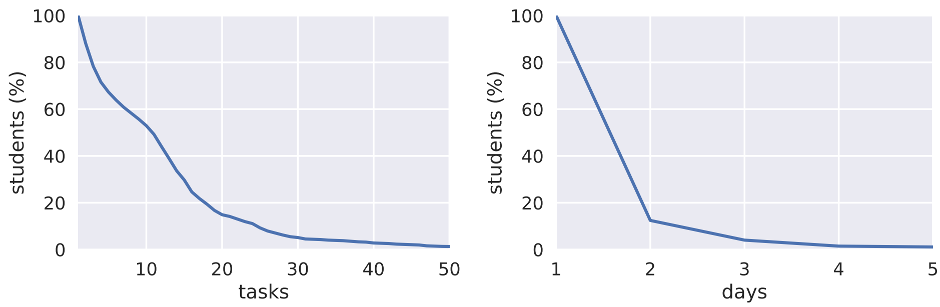
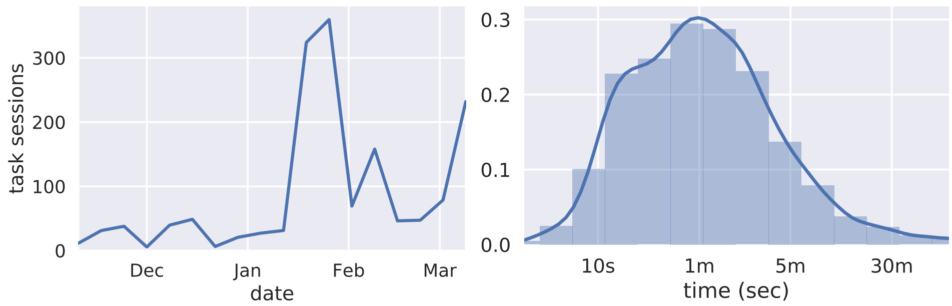


Figure 7.1: Engagement curves show how many students solved at least given number of tasks (left), and how many students solved a task at least given number of days (right).

About 11000 tasks were attempted (counting only attempts with at least one edit), and 9600 of them were solved, resulting in the overall success rate 86%. The daily numbers of solved task sessions are not stable, with high peaks on days when RoboMission was used in a programming competition or Hour of Code session at a school (Figure 7.2a). Over 180 thousand program snapshots was collected, of which 140 thousand corresponds to edits and 40 thousand to executions.

Median solving time is 1 minute (interquartile range: 24–145 seconds). Solving times follow log-normal distribution (Figure 7.2b), so the mean solving time is much higher (about 3 minutes).



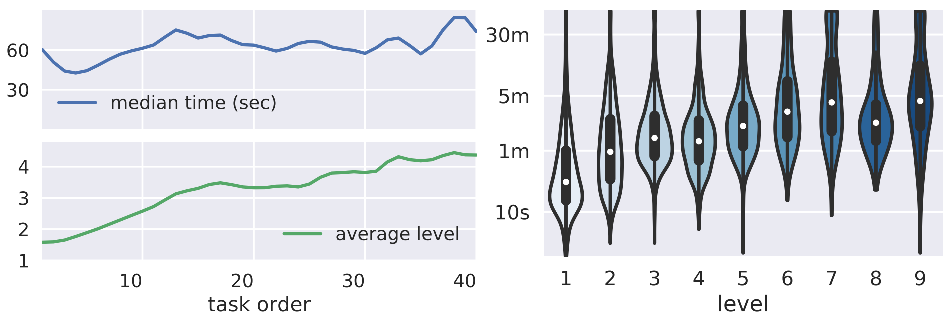
(a) Daily number of solved tasks (weekly averaged).

(b) Distribution of log-transformed solving times.

Figure 7.2: Task sessions.

7.2 System Behavior

We can use the collected data to check the requirements on system behavior mentioned in Section 5.1. Unfortunately, the system is currently not logging which tasks were recommended and which were self-selected, so we cannot determine how much are the results influenced by the adaptive behavior. From all attempts, 86% are successful, and 84% are solved in less than 15 minutes. If we look at the first 5 practiced tasks for each student, 87% of them are solved, 85% are solved in less than 15 minutes, but only 66% are solved in less than 2 minutes.



(a) Median solving time and average level per task order during practice. First 10 tasks have the median solving time below 1 minute.

(b) Distributions of log-transformed solving times for all levels, with highlighted medians and interquartile ranges.

Figure 7.3: Levels and solving times.

The median solving time of the first 10 practiced tasks is below 1 minute. Surprisingly, the median time does not increase too much even for the later attempts during practice (Figure 7.3a). The curve of median times stays similar if only students with at least 40 task sessions are included in the computation, which rules out the possibility that this behavior is caused by *attrition bias* (only better students staying in the system, making the later tasks seem easier than they are). The plot of the average level per task order during practice (Figure 7.3a bottom) reveals a possible explanation: an average student is still in the 4th level after 30 tasks, and the 3rd and 4th level have similar difficulty with a median solving time below 90 seconds (Figure 7.3b).

After 10 minutes of practice, 73% of students have reached the second level, and after 20 minutes, nearly 70% of students have started practicing loops. Then the progress through levels slows down, and only 37% of students are practicing both loops and conditional statements after the first hour of the tutorial (Figure 7.4).

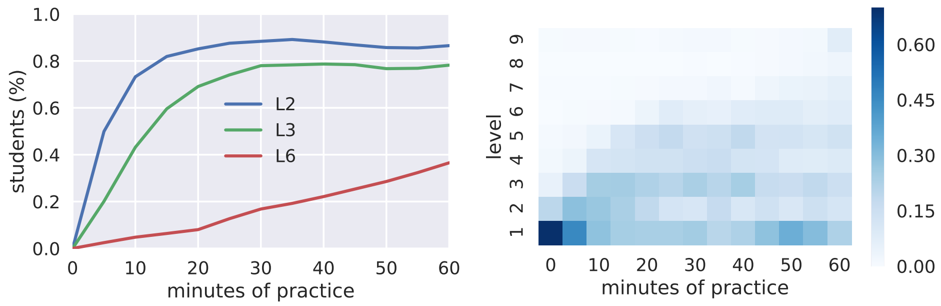


Figure 7.4: Left: the proportion of students in the system after some time of practicing, who progressed to the 2nd, 3rd, and 6th level (sequence of commands, repeat loops, if-statements). Right: how the proportion of tasks at different levels is shifting during the practice.

7.3 Performance Measurement

Measurement of performance has an enormous impact on the estimated skills, and transitively on the task recommendations. Currently, we measure performance using only observed solving time (Section 5.3). However, due to variations in speeds of different students (which are influenced by external factors, such as used device), solving time is a rather noisy approximation of the true performance. Could using other observational data, such as the number of edits, bring more information about the performance?

Overall Pearson correlation between solving times and the number of edits (both log-transformed) is quite high, about 0.73 (Figure 7.5a). However, if computed for individual tasks, median correlation drops to 0.64, and 25% of tasks have correlation below 0.5 (Figure 7.6a), which suggests that incorporating the number of edits into the measurement should be considered.

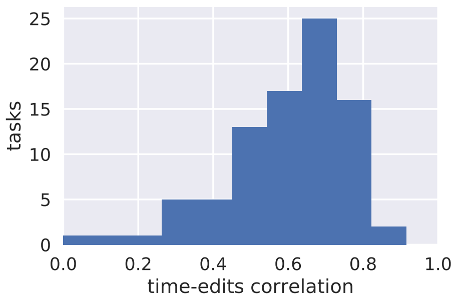
We propose the following heuristic for combining solving times with the number of edits and runs (executions): a log-transformed sum of edits, runs, and *thinking actions*. Thinking action is only assumed to be between two edits or runs if the time interval is at least 5 seconds long, in order to reduce the noise caused, e.g., by different screen sizes (5 seconds seems to be enough to drag and drop a block anywhere on any screen). If the interval is very long, we count it as multiple thinking actions (taking log_5 from the time interval and rounding it down to the closest integer).

With this definition, an average student performs about 5 thinking actions in the first two levels (practicing only sequences of commands), and about 10 thinking actions in the 3rd and 4th level (practicing repeat and while loops) (Figure 7.6b). This heuristic retains reasonable correlation with both solving time and number of edits for all tasks (Figure 7.5b).

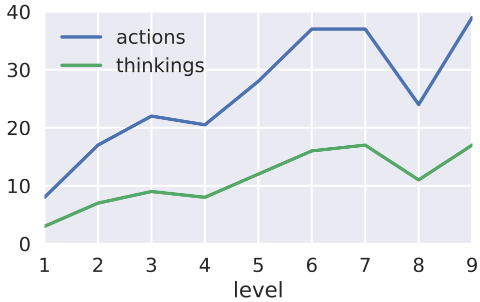
(a) Pearson correlation between performance measures computed over all task sessions.

(b) Aggregated per task, bottom 10th percentile (i.e., for 10% of tasks the correlation is below the shown value).

Figure 7.5: Correlation between four performance measures (all of them log-transformed): E = number of edits, R = number of runs, S = sum of edits, runs, and thinking actions, T = solving time.



(a) Distribution of Pearson correlations between median solving times and number of edits (both log-transformed) for individual tasks.



(b) Median number of actions (edits, runs, and thinking actions) with the increasing level.

Figure 7.6: Analysis of performance measures.

7.4 Task Difficulties

Our tutor model (Section 5.4) assumes that difficulties of tasks in each phase are approximately the same, and that the overall difficulty is increasing as the level increases. Using collected data, we can determine whether these assumptions are satisfied, and if not, suggest adjustments to improve their validity. Figure 7.3b shows that on average the difficulty of levels is increasing, but it also reveals that level 7 is more difficult than level 8. Levels 3 and 4 have similar difficulty, which is expected because they practice two similar concepts, repeat and while loops.

Looking at the difficulty of individual tasks (Figure 7.7) help us to discover outliers, whose difficulty is significantly different from the other tasks in the same level. The Figure 7.7 also suggests that dividing levels into phases is necessary to achieve reasonable homogeneity. For example, in the 2nd level (World), there are two clear groups of tasks with significantly different difficulty. Although in the other levels such a clear split does not exist, they still contain tasks with a wide range of difficulties.

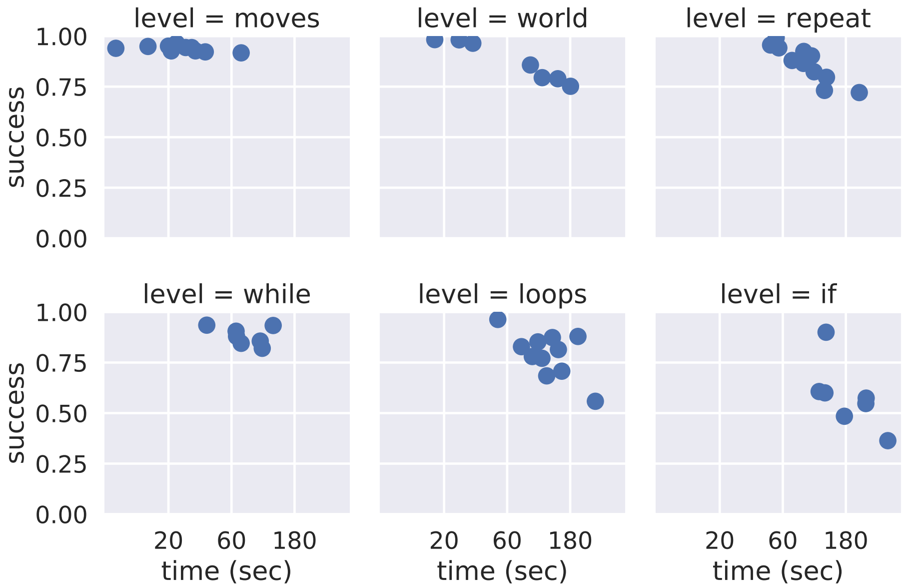


Figure 7.7: Difficulties of tasks in the first 6 levels. The difficulty is measured as success rate and median solving time.

8 Conclusion

Shoot for the moon. Even if you miss, you'll land among the stars. (B. Littrell)

This thesis describes how a combination of several strategies to support learning and motivation, together with the adaptivity of the system through techniques of artificial intelligence, can lead to efficient learning of introductory programming. Creating such a system is, however, not a one-shot task, but rather a long series of iterative improvements of a programming game, domain, student and tutor models, user interface, and the analysis layer.

With our adaptive system for learning programming, RoboMission, we have already completed a few iterations of improvements. Since its first prototype, which was implemented in 2015, we have completely changed both the programming game and the adaptive behavior. The version deployed in November 2017 was used during the following four months by about 1000 users, who solved about 10000 tasks. We tested its usability at schools during Hours of Code, as well as in two competitions for primary and secondary school students, Purkiáda[1] and InterSoB[2]. We have published our initial research on programming tasks similarities [51] and on the adaptive approach to learning programming [47].

Many improvement iterations are yet to be undertaken. After adding new game elements and programming concepts (e.g., functions and compound conditions), it will be possible to create more advanced problem sets and extend the use case of the system beyond the short motivational tutorial. Tasks could gradually transition from block-based to Python programming and teach even advanced algorithmic concepts (Figure 8.1). Next, we plan to implement a real-time dashboard for teachers, which would visualize the progress of students and suggest which students need help right now, and possibly even how to pair students for efficient pair programming.

With growing content, it would become useful to build more complex domain, student and tutor models. In order to add overlapping concepts into the domain and use them to predict student performance more accurately, we need to analyze how multiple skills interact in a single task to produce the final performance. Using data from a single task session, we want to detect which concepts contained in the task is the student struggling with. Models might be further improved by including uncertainty of estimates, forgetting of skills, and planning of tutor actions.

1. `http://purkiada.sspbrno.cz`
2. `https://intersob.math.muni.cz/2018/`

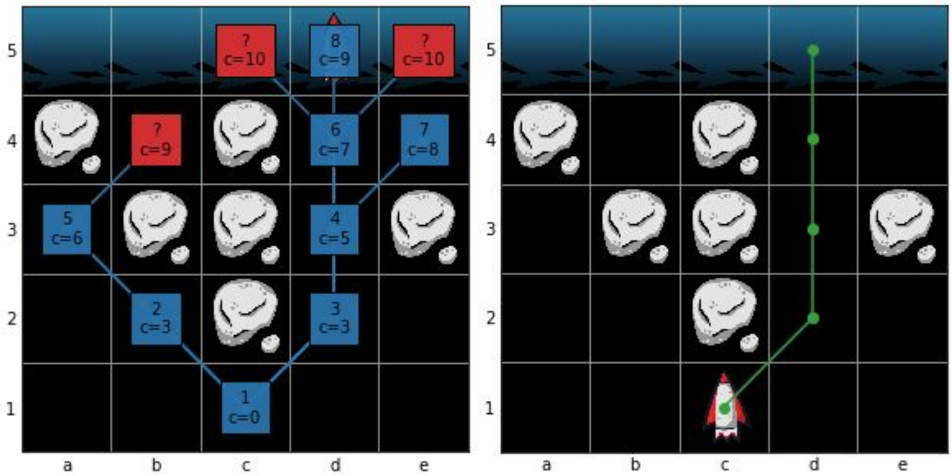


Figure 8.1: Example of how the Space World can be used for learning algorithms, in this case Dijkstra's shortest path algorithm (used at a workshop for high school students).

To evaluate which versions of models improve learning and motivation, analysis layer should be extended to allow for AB experiments and a careful attention should be paid to both long-term and online attributable metrics. We need to make sure that we collect enough data for reliable unbiased evaluation, which is complicated by feedback loops caused by the fact that the collected data which the system is evaluated on are influenced by the adaptive behavior of the system itself. In order to further align measured metrics with the system mission, we can explore how to reduce noise in flow observations by combining subjective user feedback with objective performance measures. Having reliable flow observations would be a great step towards optimizing the total amount of time spent in the state of flow.

Bibliography

[1] Vincent Aleven and Kenneth R. Koedinger. “Knowledge component (KC) approaches to learner modeling”. In: *Design Recommendations for Intelligent Tutoring Systems* 1 (2013), pp. 165–182.

[2] Vincent Aleven et al. “Instruction based on adaptive learning technologies”. In: *Handbook of research on learning and instruction. Routledge* (2016).

[3] Ryan S. J. Baker. “Stupid tutoring systems, intelligent humans”. In: *International Journal of Artificial Intelligence in Education* 26.2 (2016), pp. 600–614.

[4] Ryan S. J. Baker et al. “Towards Sensor-Free Affect Detection in Cognitive Tutor Algebra.” In: *International Educational Data Mining Society* (2012).

[5] Phil Blunsom. “Hidden markov models”. In: *Lecture notes* (2004).

[6] Adam Boduch. *Flux architecture*. Packt Publishing Ltd, 2016.

[7] Susan Bull and Theson Nghiem. “Helping learners to understand themselves with a learner model open to students, peers and instructors”. In: *Proceedings of workshop on individual and group modelling methods that help learners understand themselves, International Conference on Intelligent Tutoring Systems*. Vol. 2002. Citeseer. 2002, pp. 5–13.

[8] Thomas Butt. “Adaptive Practice: High-Efficiency Game-Based Learning”. `http://brainrush.com/pdf/AdaptivePracticeWhitepaper.pdf`. 2012. (Visited on 05/01/2018).

[9] Cory J. Butz, Shan Hua, and R. Brien Maguire. “A web-based intelligent tutoring system for computer programming”. In: *Web Intelligence, 2004. WI 2004. Proceedings. IEEE/WIC/ACM International Conference on*. IEEE. 2004, pp. 159–165.

[10] Min Chi et al. “Empirically evaluating the application of reinforcement learning to the induction of effective and adaptive pedagogical strategies”. In: *User Modeling and User-Adapted Interaction* 21.1-2 (2011), pp. 137–180.

[11] Daniel Churchill. “Towards a useful classification of learning objects”. In: *Educational Technology Research and Development* 55.5 (2007), pp. 479–497.

[12] Albert T. Corbett and John R. Anderson. “Knowledge tracing: Modeling the acquisition of procedural knowledge”. In: *User modeling and user-adapted interaction* 4.4 (1994), pp. 253–278.

[13] Mihaly Csikszentmihalyi. "Flow: The Psychology of Optimal Experience". In: (1990).

[14] Michel C. Desmarais and Ryan S. J. Baker. "A review of recent advances in learner and skill modeling in intelligent learning environments". In: *User Modeling and User-Adapted Interaction* 22.1-2 (2012), pp. 9–38.

[15] Michel C. Desmarais and Xiaoming Pu. "A Bayesian student model without hidden nodes and its comparison with item response theory". In: *International Journal of Artificial Intelligence in Education* 15.4 (2005), pp. 291–323.

[16] Nicholas Diana et al. "An instructor dashboard for real-time analytics in interactive programming assignments." In: *LAK*. 2017, pp. 272–279.

[17] Arpad E Elo. *The rating of chessplayers, past and present*. Batsford London, 1978.

[18] John A. Emrick. "An evaluation model for mastery testing". In: *Journal of Educational Measurement* 8.4 (1971), pp. 321–326.

[19] Alfred Essa. "A possible future for next generation adaptive learning systems". In: *Smart Learning Environments* 3.1 (2016), p. 16.

[20] Stephen Fancsali, Tristan Nixon, and Steven Ritter. "Optimal and worst-case performance of mastery learning assessment with Bayesian Knowledge Tracing". In: *Educational Data Mining 2013*. 2013.

[21] Artemij Fedosejev. *React. JS Essentials*. Packt Publishing Ltd, 2015.

[22] Bryan Ford. "Parsing expression grammars: a recognition-based syntactic foundation". In: *ACM SIGPLAN Notices*. Vol. 39. 1. ACM. 2004, pp. 111–122.

[23] Neil Fraser. "Ten things we've learned from Blockly". In: *Blocks and Beyond Workshop (Blocks and Beyond)*. IEEE. 2015, pp. 49–50.

[24] Jose Gonzalez-Brenes and Yun Huang. "Your model is predictive—but is it useful? theoretical and empirical considerations of a new paradigm for adaptive tutoring evaluation". In: *Proceedings of the 8th International Conference on Educational Data Mining*. University of Pittsburgh. 2015.

[25] Yun Huang et al. "Learner Modeling for Integration Skills". In: *Proceedings of the 25th Conference on User Modeling, Adaptation and Personalization*. ACM. 2017, pp. 85–93.

[26] Barry Peddycord Iii, Andrew Hicks, and Tiffany Barnes. "Generating hints for programming problems using intermediate output". In: *Educational Data Mining*. 2014.

[27] Petr Jarušek and Radek Pelánek. "A web-based problem solving tool for introductory computer science". In: *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*. ACM. 2012, pp. 371–371.

[28] Joel Jones. "Abstract syntax tree implementation idioms". In: *Proceedings of the 10th conference on pattern languages of programs*. 2003.

[29] Caitlin Kelleher and Randy Pausch. "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers". In: *ACM Computing Surveys (CSUR)* 37.2 (2005), pp. 83–137.

[30] Sharon Klinkenberg, Marthe Straatemeier, and Han LJ van der Maas. "Computer adaptive practice of maths ability using a new item response model for on the fly ability and difficulty estimation". In: *Computers & Education* 57.2 (2011), pp. 1813–1824.

[31] Thomas Kluyver et al. "Jupyter Notebooks-a publishing format for reproducible computational workflows." In: *ELPUB*. 2016, pp. 87–90.

[32] Kenneth R. Koedinger, Albert T. Corbett, and Charles Perfetti. "The Knowledge-Learning-Instruction framework: Bridging the science-practice chasm to enhance robust student learning". In: *Cognitive science* 36.5 (2012), pp. 757–798.

[33] Sean Leonard. *The text/markdown Media Type*. RFC 7763. 2016.

[34] Jingchen Liu, Gongjun Xu, and Zhiliang Ying. "Data-driven learning of Q-matrix". In: *Applied psychological measurement* 36.7 (2012), pp. 548–564.

[35] Mark Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media, Inc., 2011.

[36] Antonija Mitrovic and Stellan Ohlsson. "Evaluation of a constraint-based tutor for a database language". In: (1999).

[37] Kevin Patrick Murphy and Stuart Russell. "Dynamic Bayesian networks: representation, inference and learning". In: (2002).

[38] Juraj Nižnan. "Learning from problem-solving data". MA thesis. Masaryk University, 2013.

[39] Roger Nkambou, Riichiro Mizoguchi, and Jacqueline Bourdeau. *Advances in intelligent tutoring systems*. Vol. 308. Springer Science & Business Media, 2010.

[40] Jan Papoušek, Radek Pelánek, and Vıt Stanislav. "Adaptive practice of facts in domains with varied prior knowledge". In: *Educational Data Mining*. 2014.

[41] Jan Papoušek, Vıt Stanislav, and Radek Pelánek. "Evaluation of an adaptive practice system for learning geography facts". In: *Proceedings of the Sixth International Conference on Learning Analytics & Knowledge*. ACM. 2016, pp. 134–142.

[42] Ivailo Partchev. "A visual guide to item response theory". In: (2004). Friedrich Schiller Universität Jena.

[43] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Elsevier, 2014.

[44] Radek Pelánek. "Applications of the Elo rating system in adaptive educational systems". In: *Computers & Education* 98 (2016), pp. 169–179.

[45] Radek Pelánek. "Bayesian knowledge tracing, logistic models, and beyond: an overview of learner modeling techniques". In: *User Modeling and User-Adapted Interaction* 27.3-5 (2017), pp. 313–350.

[46] Radek Pelánek. "Metrics for evaluation of student models". In: *Journal of Educational Data Mining* 7.2 (2015), pp. 1–19.

[47] Radek Pelánek and Tomáš Effenberger. "Towards Making Block-based Programming Activities Adaptive". In: *Proceedings of Learning at Scale*. To appear. ACM, 2018.

[48] Radek Pelánek and Petr Jarušek. "Student modeling based on problem solving times". In: *International Journal of Artificial Intelligence in Education* 25.4 (2015), pp. 493–519.

[49] Radek Pelánek and Jiřı Řihák. "Experimental analysis of mastery learning criteria". In: *Proceedings of the 25th Conference on User Modeling, Adaptation and Personalization*. ACM. 2017, pp. 156–163.

[50] Radek Pelánek, Jirı Řihák, and Jan Papoušek. "Impact of data collection on interpretation and evaluation of student models". In: *Proceedings of the Sixth International Conference on Learning Analytics & Knowledge*. ACM. 2016, pp. 40–47.

[51] Radek Pelánek et al. "Measuring Item Similarity in Introductory Programming". In: *Proceedings of Learning at Scale*. To appear. ACM, 2018.

[52] Thomas W. Price and Tiffany Barnes. "Comparing textual and block interfaces in a novice programming environment". In: *Proceedings of the eleventh annual International Conference on International Computing Education Research*. ACM. 2015, pp. 91–99.

[53] Francesco Ricci, Lior Rokach, and Bracha Shapira. "Recommender Systems Handbook". In: 2nd. Springer, 2015.

[54] Jiřı Řihák. "Modeling Techniques for Adaptive Practice Systems". PhD thesis. Masaryk University, 2017.

[55] Jiřı Řihák. "Use of Time Information in Models behind Adaptive System for Building Fluency in Mathematics." In: *International Educational Data Mining Society* (2015).

[56] Steve Ritter et al. "How mastery learning works at scale". In: *Proceedings of the Third (2016) ACM Conference on Learning@ Scale*. ACM. 2016, pp. 71–79.

[57] Marshall Rosenberg. *Nonviolent Communication: A Language of Life: Life-Changing Tools for Healthy Relationships*. PuddleDancer Press, 2015.

[58] Jesse Schell. *The Art of Game Design: A book of lenses*. CRC Press, 2014.

[59] Robert A. Sottilare et al. *Design Recommendations for Intelligent Tutoring Systems: Volume 4-Domain Modeling*. Vol. 4. US Army Research Laboratory, 2016.

[60] Robert A Sottilare et al. *Design recommendations for intelligent tutoring systems: Volume 1-learner modeling*. Vol. 1. US Army Research Laboratory, 2013.

[61] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2018.

[62] Lev Vygotsky. "Zone of proximal development". In: *Mind in society: The development of higher psychological processes* 5291 (1987), p. 157.

[63] Howard Wainer et al. *Computerized adaptive testing: A primer*. Routledge, 2000.

[64] Lisa Wang et al. "Learning to represent student knowledge on programming exercises using deep learning". In: *Proceedings of the 10th International Conference on Educational Data Mining; Wuhan, China*. 2017, pp. 324–329.

[65] Shuhan Wang, Fang He, and Erik Andersen. "A unified framework for knowledge assessment and progression analysis and design". In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM. 2017, pp. 937–948.

[66] David Weintrop and Uri Wilensky. "Comparing block-based and text-based programming in high school computer science classrooms". In: *ACM Transactions on Computing Education (TOCE)* 18.1 (2017), p. 3.

[67] David Weintrop and Uri Wilensky. "The challenges of studying blocks-based programming environments". In: *Blocks and Beyond Workshop (Blocks and Beyond)*. IEEE. 2015, pp. 5–7.

[68] Cameron Wilson. "Hour of code—a record year for computer science". In: *ACM Inroads* 6.1 (2015), pp. 22–22.

[69] Yanbo Xu and Jack Mostow. "Using Logistic Regression to Trace Multiple Sub-skills in a Dynamic Bayes Net." In: *EDM*. Citeseer. 2011, pp. 241–246.

[70] Martin Zinkevich. “Rules of Machine Learning: Best Practices for ML Engineering”. 2017. URL: http://martin.zinkevich.org/rules_of_ml/rules_of_ml.pdf (visited on 11/19/2017).

A Attachments

Source code, exported data and performed analyses are available as attachments in the Archive of Thesis[1] in the Information System of Masaryk University, and also in GitHub repositories of the project.

A.1 Source Code

Snapshot of the source code (including setup instructions in `README.md`) is attached as `code.zip`. The most recent version of the project is available at `https://github.com/adaptive-learning/robomission`.

A.2 Analyses

Attachment (`analysis.zip`) contains several Jupyter Notebooks and Python modules with the analyses presented in the thesis. It is also available at `https://github.com/effa/flocs-thesis/tree/master/analysis`.

A.3 Exported Data

Both the domain and collected data used for analyses in this thesis (exported on 9th March 2018) are attached as `data.zip`, and they are also available at `https://github.com/effa/flocs-thesis/tree/master/data`. Data is exported as a few CSV tables, with the columns described below.

Table A.1: Tasks.

Attribute	Description
id	ID of the task
name	text label of the task
level	name of the level which contains the task
setting	JSON with "fields" and optional limits ("length", "energy")
solution	MiniRoboCode representation of the solution (Section 6.3.3)

1. `https://is.muni.cz/th/410350/fi_m/`

Table A.2: Problem sets.

Attribute	Description
id	ID of the problem set
level	order of the problem set (by difficulty)
name	text label of the problem set
toolbox	name of the available toolbox
tasks	list of task names contained in the problem set

Table A.3: Task sessions.

Attribute	Description
id	ID of the task session
student	ID of the student
task	ID of the task
solved	whether the student solved the task (boolean)
start	timestamp of opening the task
end	timestamp of the last action in the task session
time_spent	length of the task session in seconds

Table A.4: Program snapshots.

Attribute	Description
id	ID of the program snapshot
task_session	ID of the task session
time	timestamp of creating the snapshot
program	MiniRoboCode representation of the code (Section 6.3.3)
granularity	"edit" or "execution"
order	order of the snapshot of this granularity in the session
correct	whether the execution was successful (empty for edits)
time_from_start	seconds from the start of the task session
time_delta	seconds from the previous snapshot of same granularity

B Used Technologies

The technologies used in the project change quite quickly in order to meet new requirements, especially on the frontend, where the tools and best practices are evolving rapidly. The following tables show the overview of technologies used in the project in spring 2018.

Table B.1: Project Management.

Area	Technology	Main Reason
Version control	git, GitHub	features
Custom commands	make, django, npm	easy to use
Monitoring	Google Analytics	easy to use

Table B.2: Backend.

Area	Technology	Main Reason
Language	Python 3	concise, high-level
Dependencies	pip	standard tool, easy to use
Environment	virtualenvwrapper	easy to use
Unit tests	pytest	concise, readable
Web framework	Django	features, well-documented
REST API	DRF	features, well-documented
Database	PostgreSQL	widely used
Web server	Nginx, Gunicorn	widely used
Scheduled jobs	cron, django-crontab	standard tool
Data export	Django Rest Pandas	DataFrame tranformations

Table B.3: Frontend.

Area	Technology	Main Reason
Language	ES6	concise, readable
Dependencies	npm	easy to use
Bundling	webpack	feature-complete
Compiling	babel	feature-complete
State	redux	predictability, testability
Side effects	redux-sagas	readability, testability
Views	React	declarative, reusable
UI components	Material-UI	good appearance
HTTP client	axios	Promises API
Code blocks	Blockly	well-tested
Parsing	PEG JS	declarative rules
Interpretation	JS-interpreter	stepping, custom hooks
Localization	react-intl	one place for all messages

Table B.4: Analysis.

Area	Technology	Main Reason
Language	Python 3	concise, high-level
Documents	Jupyter Notebook	interactivity
DataFrames	pandas	features, well-documented
Plotting	matplotlib, seaborn	features, nice plots