UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie
master in de informatica

Masterthesis

PERSSISTANT : A Progress Estimation System to Personalize Learning Trajectories

Tijl Elens
Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :
Prof. dr. Davy VANACKEN

COPROMOTOR :
Prof. dr. Gustavo Alberto ROVELO RUIZ

2023
2024

De transnationale Universiteit Limburg is een uniek samenwerkingsverband van twee universiteiten in twee landen: de Universiteit Hasselt en Maastricht University.

UHASSELT
KNOWLEDGE IN ACTION

www.uhasselt.be
Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

PERSSISTANT : A Progress Estimation System to Personalize Learning Trajectories

Tijl Elens
Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :
Prof. dr. Davy VANACKEN

COPROMOTOR :
Prof. dr. Gustavo Alberto ROVELO RUIZ

Hasselt University

Master's thesis nominated for obtaining a master's degree in computer science

PERSSISTANT: A Progress Estimation System to Personalize Learning Trajectories

Author:

Elens Tijl

Promotor:

prof. dr. Davy Vanacken

Co-promotor:

prof. dr. Gustavo Rovelo

Mentors(s):

Sebe Vanbrabant

Academic Year 2023-2024

Acknowledgements

I would like to thank my promotor, prof. dr. Davy Vanacken, and my mentor, Sebe Vanbrabant, for their thorough reading of important parts of the thesis text throughout the year. I would also like to thank Jarne Thys, for consistently marking all half-baked phrases, sentences, and captions I forgot to finish, and by extension the rest of the mentoring team. I am sure I would have forgotten about them when working towards the deadline. Finally, I would like to thank my co-promotor prof. dr. Gustavo Rovelo for providing an additional eye to the thesis text, and also making suggestions. Additionally, I would like to thank my promotor and co-promotor for helping me broaden my perspectives, whenever my constraint-solving thinking had run stuck. This happened a few times throughout the year and was necessary to make valuable progress in the thesis. In general, I would like to thank everyone of the mentoring team, as many of the ideas presented in this thesis have been brainstormed upon during the many meetings. I believe this thesis would not have reached the same quality if I were to do it on my own.

I would like to thank the EDM for providing me the facilities to work on the thesis in their office and using their equipment. I would also like to thank everyone who showed interest and provided their perspectives on my work. In particular, I would like to thank Raf Menten for his useful feedback, who briefly mentored me during the first month of the thesis.

Finally, I would like to thank my peers and family for the support they provided. In particular, my parents, who supported my workflow by making dinner when I was working late and driving me, when needed. Furthermore, I specifically would like to thank my father to dedicate a full day to thinking and designing a clearer version of my prototype overview image used in Chapter 7.

Abstract

In the contemporary educational landscape, the integration of Artificial Intelligence (AI) has revolutionized personalized learning, particularly within programming education. Through a comprehensive review of related work in AI in Education (AIED) and the deployment of Large Language Models (LLMs), this thesis outlines a framework for personalized learning trajectories that adapt to students' unique needs. This thesis explores the development and implementation of a novel system named PERSSISTANT (Progress Estimated Representation through Structure-based Skill testing to Individualized Student learning Trajectories via interventions Activated through Need or Time). The primary objective of PERSSISTANT is to leverage AI technologies to continuously monitor and assess students' programming skills to provide them clear progress overviews. Furthermore, the measured progression is used to provide early attempts at personalized interventions that enhance learning outcomes. The findings of the research collected and our own work suggest that while AI can significantly improve personalized learning experiences, careful consideration must be given to integrating AI techniques that influence the learning process. Ultimately, this research contributes to the ongoing discourse on the role of AI in education, providing insights into how intelligent systems can support personalized learning.

Summary

In the evolving landscape of education, the integration of Artificial Intelligence (AI) has emerged as a transformative force, particularly in personalized learning within programming education. This thesis presents a comprehensive exploration of this integration through the development of a novel system named **PERSSISTANT**: Progress Estimated Representation through Structure-based Skill testing to Individualized Student learning Trajectories via interventions Activated through Need or Time.

The primary aim of PERSSISTANT is to utilize AI technologies to continuously monitor and assess students' programming skills, thereby providing clear progress overviews and facilitating personalized interventions to enhance learning outcomes. Through a thorough review of existing literature in the field of AI in Education (AIED) and the application of Large Language Models (LLMs), the thesis outlines a framework for personalized learning trajectories tailored to individual student needs.

Key findings indicate that while AI has the potential to significantly improve personalized educational experiences, it is crucial to carefully consider the integration of AI techniques that influence the learning process. Our research highlights three pivotal questions regarding the effective monitoring of student progress, real-time feedback of this progress, and the initial attempts at personalization of learning trajectories.

The discussions and insights presented in this thesis contribute to the ongoing discourse on the role of AI in education, emphasizing how intelligent systems can support and enhance personalized learning experiences. Ultimately, this research not only advances the understanding of AI's capabilities in educational contexts but also discusses the challenges and ethical considerations surrounding its application.

Contents

Chapter 1

Introduction

In today's educational landscape, **personalized instruction** plays a pivotal role in enhancing the learning experience. There is broad consensus that tailoring tuition to students' specific needs, is the most effective form of educational interaction. These benefits are well-documented, notably through Bloom's influential findings in 1984 comparing one-to-one (private) tutoring with classroom instruction [Nwa90]. The research of Bloom revealed that students receiving one-to-one tutoring performed two standard deviations better than those only participating in group classes [MG23]. In total, 98% of students who received one-to-one tutoring outperformed their classroom-only peers, despite both groups spending the same amount of time on the material [Nwa90]. Further research collected by Nwana [Nwa90] also highlighted a four-to-one advantage for private tutors in terms of the time required for students to reach the same level of proficiency. However, by 1990, educational systems have become geared towards group teaching out of necessity, thereby losing many of the advantages of one-to-one tutoring.

In the meantime, we witnessed rapid technological advancements [Ros22], which strongly influence higher education and its future [PK17]. For instance, the introduction of the internet led to the democratization of knowledge, as any student with a connection now has access to free learning content regardless of nationality [RW16; CCL20]. It also spurred the development of **e-learning platforms and Massive Open Online Courses (MOOCs)**, enabling students to learn anywhere and at any time [RW16]. At the same time, it enables institutions of higher education to increase the student-to-instructor ratio, supporting millions of students every year [RW16]. On the other hand, MOOCs have a significantly high drop-out rate [MG23], and typically offer limited personalized abilities, as they primarily consist of a single instructor giving lectures and basic multiple-choice assessment [RW16].

Recently, as research into computing power and machine intelligence advanced, Artificial Intelligence (AI) has begun outperforming humans in complex tasks [Ros22]. For instance, since the past decade, AI has shown to perform more accurately at handwriting recognition, image recognition, and speech recognition, as well as reading comprehension and language understanding than humans. Given AI's growing ability to mimic or even outperform human tasks, it is evident that AI-tools are being integrated in the classroom and research is ongoing on using **AI in Education (AIED)**. For instance, AI already helps higher institutions with organizational tasks, such as timetabling of exams and courses [MG23], and plagiarism detection. Automatic plagiarism detection tools have greatly improved the identification of misconduct, whereas in the past, suspicions often went unverified due to the time-consuming nature of manual checks [CCL20; NMS22]. Such automated tools allow instructors to focus more on one-to-one tutoring [CCL20]. Furthermore, research continues to develop new AI methods and tools, for instance, to help reduce the number of reviews and review time that experts need to evaluate graduate program applicants [MG23]. These automations are becoming increasingly important for defunded universities that seek economical solutions [PK17].

These advancements in AIED also include methods focused on personalized instruction. For instance, **Intelligent Tutoring Systems (ITSs)** provide virtual tutors that model the domain knowledge and the student's knowledge and motivate states, to provide personalized assistance [CA94; Nwa90]. They are built on the concept of mastery learning, envisioning that all students can achieve expertise in a domain if 1) the domain knowledge is appropriately analyzed into a hierarchy of component skills, and 2) learning experiences are structured so that students have mastered prerequisite skills before tackling skills at a higher level in the hierarchy [CA94]. Just like AI in general, ITSs attempt to mimic a good human teacher that gives immediate feedback to students on the tasks they perform [Nwa90]. ITSs can even provide one-to-one tutoring, without losing the benefits of a group teaching environment when utilized per student in class [Nwa90].

With the advent of publically available **Large Language Models (LLMs)** that provide human-like responses and understanding context, further personalization opportunities are being investigated. For instance, a study showed that a conversational agent was able to provide explanations tailored to students' misconceptions and could adapt to their level of understanding [BO23]. Furthermore, LLMs bring the opportunity of generating personalized exercises to ensure that students are mastering the learning material [Kas+23]. LLMs can produce contextualized problem statements within certain thematic topics [Bec+23]. Furthermore, since including personal details or favorite items can increase the interest of students [RCH18], further opportunities exist in generating personalized exercises.

Research however indicates that, similar to other AI technologies, there are significant concerns regarding the integration of AI in education. For example, using AI to offer personalized learning could exacerbate inequality due to disparities in access to learning platforms [Mag+21], which is a concern also raised by UNESCO more recently [Kas+23]. Moreover, due to biases in the training data, certain subgroups may benefit more than others in AI-driven personalized education [Mag+21]. Furthermore, these tools are able to collect a lot of personal and often sensitive data [Kas+23]. This opens up the risk that students are being closely monitored or controlled [Mag+21], their data is exploited for non-school purposes [WMB24], or their data is compromised in the event of a bigger company's data breach [Kas+23]. Moreover, the marketing of AI tools integrating OpenAI's GPT suggest providing one-to-one tutoring, at a similar ability of human teachers, devaluing the latter's in-class experience and subject expertise [WMB24]. Another important difference between business driven recommender systems and personalized education in the classroom is the optimization objective, since the former focuses on some form of user engagement to maximize profit, whereas the latter focuses on some form of learning outcomes [Mag+21].

In response to the growing need for personalized learning tools, we propose **PERSSISTANT**: Progress Estimated Representation through Structure-based Skill testing to Individualized Student learning Trajectories via interventions Activated through Need or Time. This system uses AI to assess programming students' skills, track their progress and provide early attempts at personalized interventions based on the progress to improve learning outcomes. The system's role is to assist students by recommending exercises that it believes best support their current trajectory, with respect to the student's autonomy. The goal is that students will better persist in making exercises, rather than getting demotivated.

Our research is grounded in three pivotal questions:

- Q1: How can we effectively leverage AI to continuously monitor the progress of a single student in an introductory programming course, ensuring accurate tracking of their knowledge acquisition?
- Q2: What are the most effective methods for communicating real-time progress to students, so they can make informed decisions on the next learning unit?
- Q3: How can AI-driven progress monitoring be used to personalize students' learning trajectories?

In Chapter 2, we provide an overview of the related work in the field of AIED, narrowing down to research applicable to programming courses.Next, we discuss our conceptual framework to layout personalized learning trajectories in Chapter 3. The use of GPT3.5 to support this personalizing framework is further briefly explored in Chapter 4. In Chapter 5, we briefly explore the submissions on an introductory programming course collected from fellow students. With the knowledge gained from real-world submissions, we discuss conceptually how skills can be assessed from students' programming submissions to represent progress in Chapter 6, in order to support the student in the learning process. Next, we describe our prototype to test our skill assessment approach in Chapter 7, which consists of an HTTP-server and a client where the student can make programming exercises. In Chapter 8 we will give a critical review of the prototype and discuss opportunities for future work. Finally, we will conclude this thesis in Chapter 9.

Chapter 2

Related Work

The field of **Artificial Intelligence in Education (AIED)** has witnessed remarkable advancements, reflecting a growing recognition of the potential benefits AI technologies can bring to personalized learning. This chapter delves into the existing body of research surrounding AIED, providing a comprehensive overview of its applications, challenges, and opportunities. In Section 2.1, we provide a broad overview of work on AIED important to the thesis. In Section 2.2, we overview the opportunities that AI brings for instructors to lower the time burden. Next, in Section 2.3, we overview methods to track the knowledge of a student, necessary to effective personalization. As our work leverages simpler feedback mechanisms, we also overview methods dependent on student code in Section 2.4. Finally, we discuss the challenges of integrating AI in education in Section 2.5.

2.1 Overview of AIED

AIED is a major field of research and development [WMB24] that has evolved dramatically since its inception. The origins of AIED can be traced back to the 1920s with Sidney Pressey's first teaching machine, which aimed to help students find correct answers to multiple-choice questions [NMS22]. **Computer-based Training (CBT)**, **Computer Assisted Instruction (CAI)** systems from the 1960s and 1970s, followed by **Intelligent Tutoring Systems (ITS)** from the 1980s, marked the early applications of AI in education [BSH96; WMB24; Nwa90]. In this section, we will outline some broad AEID research lines important to our thesis.

2.1.1 Interactive learning environments

The integration of AI in education has led to the development of **Interactive Learning Environments (ILEs)** that offer unique affordances compared to traditional classroom settings. These ILEs can be collaborative, omnipresent, and portable, supporting learning anytime, anywhere, by anyone. A clear example of these affordances is the growing movement of **Massive Online Open Courses (MOOCs)** [RW16]. According to Maghsudi et al. [Mag+21], the last decade has witnessed an explosion in the number of web-based learning systems due to increasing demand in higher-level education, the limited number of teaching personnel, advances in information technology and AI, and, more recently, COVID-19. Research on better supporting human assistance in large-scale online education is still ongoing. For example, Guo [Guo15] created Codeopticon, where an assistant overviews students' code editors while they are doing exercises and running into errors to provide them timely feedback through chat. However, according to research collected by Tetzlaff et al. [TSB21], there is no advantage to using e-learning environments compared to regular instruction, despite the personalization strategies in digital learning environments.

Another important part of the research in ILEs is the aforementioned ITSs, which have evolved from the CAI methods. Beck et al. [BSH96] remark that CBT and CAI systems often followed rigid scripts (such as "if question 21 is answered correctly, proceed to question 54; otherwise go to question 32.") that have some positive learning effect but lack the individualized guidance of a human tutor. In contrast, ITSs provide considerable flexibility in the presentation of material and a greater ability to respond to idiosyncratic student needs. ITSs follow the findings of Bloom that students who received additional tutoring fared two standard deviations better than those who only participated in group classes [MG23]. This is further demonstrated by Smithtown, an ITS with personalized learning paths that allowed students to learn economics as effectively as those in traditional settings, but in only half the time, showcasing the efficiency of said personalized learning paths [BSH96]. Furthermore, Tetzlaff et al. [TSB21] claim that ITSs are suitable for promoting learning gains as they take a much more dynamic approach to personalization by modeling the student.

2.1.2 Personalized learning and student motivation

Other relevant research is that of **personalized education**, which Tetzlaff et al. [TSB21] define as "the data-based adjustment of any aspect of instructional practice to relevant characteristics of a specific learner." Notice that the definition does not necessarily require automation; it also includes human teachers providing personalized instruction in the classroom. In their contribution, Tetzlaff et al. provide a dynamic framework of personalized education based on three learner dynamics: 1) on the macroscale of higher-order goals, such as mastery and skill acquisition; 2) on the mesoscale of instructional units, such as processing bundles of tasks, reading material, etc.; and 3) on the microscale driven by short-term fluctuations in relevant student characteristics such as the affective state or working memory capacity.

The framework itself has a broad application, especially for ITSs, as their work supports beliefs in ITSs. If the student is provided too little assistance (e.g., by hints or direct instructions), he/she may be unable to make progress on the task or become frustrated, whereas too much assistance does not require the learner to engage in the cognitive processes necessary for deep processing [TSB21; BSH96]. In an attempt to reduce students seeking too much assistance, research on programming by Marwan et al. [MDP20] cuts incidence of help avoidance almost in half and prevents help abuse altogether. Schiaffino et al. [SGA08] investigate the opportunities of personalized proactive assistance, by modeling intuitive students and visual students to better decide the level of assistance. Other work also attempts to balance providing assistance. Fossati et al. [Fos+15] distinguish 'reactive' feedback that interrupts the student when they get stuck in iList Tutor, and 'proactive' feedback, that tries to anticipate future movements by possible initiating a tutorial with the student. Hobert [Hob19] also created a personalized assistant called Coding Tutor that automatically proactively offers assistance in programming tasks.

Similar to any other task, students require motivation for learning [Mag+21], which is why research further investigates this behavior and attempts to model it. According to Reber et al. [RCH18], it is essential to foster students' interest, as it has been shown to improve persistence and learning outcomes, and to have a positive effect on academic choices. In the research collected by Maghsudi et al. [Mag+21], the authors claim conventional incentives like grades and certificates play a crucial role in maintaining motivation, but can be extended with online bonuses for online learning material. Gamification can further motivate students to pay attention [Mag+21]. Silapachote and Srisuphab [SS14] claim that a high level of abstract knowledge contributes to student discouragement, and used practical games to introduce a new concept before explaining its theories, which effectively increased students' attention span with 50.79 minutes in a 3-hour-long class. According to some papers overviewed by Reber et al. [RCH18], it is suggested that the mere choice, or just the experience of making a choice, is enough to increase interest, presumably through increasing autonomy support. Furthermore, ITSs have been shown to be highly effective in increasing student's motivation and learning [BSH96].

2.1.3 Large Language Models

Finally, the research on education was shaken up by recent advances in **Large Language Models (LLMs)**. An LLM is a type of artificial intelligence model that utilizes deep learning techniques to learn the structure and relationships of human language and give human-like responses. These models are characterized by their extensive training on massive datasets, often comprising billions of words [Kou22], which enables them to predict which word is most likely to follow the words preceding it [WMB24]. Therefore, these general-purpose models can perform various Natural Language Processing (NLP) tasks such as text generation, summarization, translation, and question-answering.

LLMs have shown promising results in assisting various aspects of education. Kasneci et al. [Kas+23] provide a broad overview of all the opportunities for ChatGPT. For students, these range from generating assessment questions that human experts favor, generating interactive educational material such as quizzes and flashcards to acting as a conversational partner in language education, etc. Baidoo-anu and Owusu Ansah [BO23] highlight a study remarking that by using ChatGPT, an adaptive learning system could provide more effective support to programming students based on their progress and performance, which lead to improved performance on assessment.

In educational domains where knowledge is factual (such as history or biology), NLP-based tools can be used for extracting key facts from long and redundant textbook sections [Mag+21], which can help students find information better. This finding is somewhat confirmed by research testing the ability of LLMs to present correct facts. In a recent study in medical education by Kung et al. [Kun+23], GPT3.5 was able to reach the passing threshold of the United States Medical Licensing Exam or do so nearly. This was remarkable, because this was without any domain-specific fine-tuning, which demonstrates its potential to assist a human learner in medical education and clinical decision-making processes. LLMs can also produce useful knowledge for students in programming courses. Wang et al. [Wan+23] selected 30 to 36 exercises consisting of multiple choice questions, short answer questions, and coding questions for 7 courses, and provided each of these problems to GPT to solve them correctly in three attempts. They found that for 61.5% of all problems, ChatGPT solved them correctly within at most three attempts and in particular for the introductory programming course, this number was much higher (81.5%). Becker et al. [Bec+23] note that code generation tools can also be used to help expose students to the variety of ways that a problem can be solved.

2.2 Administrative and instructive opportunities in AEID

AI has significantly reduced the paperwork and workload on instructors, particularly in the performance of various administrative functions. As stated by multiple authors, this enables them to focus on their core mandate, instruction, dissemination of content and materials [CCL20; WMB24], although some warn that the adoption can also introduce new pressures and burdens, as teachers will have to learn how to use these new technologies and their limitations [WMB24]. In the next sections, we will focus on instructors' content creation, grading of exercises, and misconduct tools.

2.2.1 Content creation

Several works have researched the opportunities for instructors in the domain of education. AI methods can assist instructors in the creation of learning material [Mag+21], which currently initially costs those in computer science 6 to 8 hours per class meeting [MHP23]. Contemporary AI-driven tools like Jasper and Grammarly have already simplified the preparation of instructional materials, such as tutorials, lecture notes, and handouts [NMS22]. LLMs can further help with these applications, as well as generate course syllabi, creating prompts that encourage participation and critical thinking across different knowledge levels [Kas+23].

Maghsudi et al. [Mag+21] noted that AI methods can be used to maintain and update the learning material by identifying the areas of priority for new content and assessment questions that need to be improved and providing drafts. Furthermore, Kasneci et al. [Kas+23] found that large language models assist teachers with their professional development by providing resources, summaries, and explanations of new teaching methodologies and technologies, which ultimately helps them create useful learning material.

2.2.2 Exercise generation

Another opportunity for AIED is exercise generation to enhance student learning and comprehension, reducing the time educators spend creating assessments or preventing plagiarism [MG23]. In their overview of ChatGPT, Kasneci et al. [Kas+23] remark that LLMs can be used to generate question-answer pairs to test factual knowledge for reading comprehension, as well as generate distractors: incorrect answers intended to mislead a multiple-choice question. Dijkstra et al. [Dij+22] created EduQuiz based on GPT3.5, an initial step towards fully automatic quiz generation. They found that it is promising in reducing teachers' burdens, but acknowledge that it sometimes produces lower-quality quizzes compared to human-generated ones. However, as noted in the related work of Radošević et al. [ROS10], using multiple-choice questions in programming assists only with learning basic theoretical aspects, and not with developing practical skills.

LLMs are also applicable when it comes to generating exercises specifically for computer science. Early research by Radošević et al. [ROS10] has attempted to generate code by using templates with placeholders that would be filled in based on the student's ID. The student then had to answer questions. The Codex model demonstrates capability in generating sensible and novel exercises and an appropriate sample solution from a single priming example [Bec+23; Kas+23]. This also included additional code explanations and automated tests to verify the student's solutions [Kas+23].

2.2.3 Grading

When it comes to grading, much research has been conducted, and many tools have been developed. This is necessary, as the research by Mirhosseini et al. [MHP23] demonstrates that instructors in computer science spend up to 50% of their time on grading and giving feedback, which largely varies depending on their skills and automation available. Baidoo-anu and Owusu Ansah [BO23] have outlined a study demonstrating that LLMs can be trained to grade student essays with high accuracy, closely matching human grading and providing similar feedback. Additionally, as commented by Chen et al. [CCL20], online tools such as TurnItIn and Ecree assist instructors by offering suggestive grading and plagiarism checking. More advanced systems like the GRADE utilize explainable AI to streamline the admission process by predicting the likelihood of an applicant's acceptance, which helps to reduce the need for exhaustive application reviews [MG23].

AI and machine learning are also instrumental in grade prediction and educational data analysis. Rechkoski et al. [RAM18] used model-based collaborative filtering methods to predict student grades for a course, given their past grades. They obtained ± 1 accuracy, with grades ranging from 5 up to 10, where a score of 5 means the student has failed the course. Rechkoski et al. remark that predictive grading can help students make informed decisions about course enrollments, ultimately impacting satisfaction, motivation, and reducing dropout rates. As collected in the related work of Moore et al. [Moo+22], even keystroke data analysis can be used to predict which students will pass in introductory coding classes. Predicting which students will not pass, allows early intervention to support struggling students [Moo+22; RAM18].

2.2.4 Misconduct

An important but time-consuming part of an instructor is verifying misconduct, which becomes only more when searching manually for similar solutions. Many works praise the advent of AI to detect text-based plagiarism, saving the instructors tremendous time as they would otherwise not have the resources to prove it [CCL20; NMS22]. It is worth noting however that AI itself makes plagiarism detection an uphill battle. The automatic grader TurnItIn can, for example, not accurately detect plagiarism through translations [Laa+23], which is problematic given the prevalence of easily accessible translation tools and the increasing prevalence of LLMs. InstructGPT and ChatGPT are capable of producing natural-sounding essays and short answers, and even working code snippets in response to a text prompt, making it easier than ever for dishonest learners to misuse such systems for authoring assignments, projects, research papers, or online exams [MG23]. This situation has led to measures such as the New York City Department of Education's ban on ChatGPT, the development of new strategies and tools like GPTZero to identify AI-generated content, and watermarking generated content to bias AI detection [Kas+23].

The situation with the advent of LLMs is no different regarding programming courses. Mallik and Gangopadhyay [MG23] noted that ChatGPT can also generate high-quality code that is hard to distinguish from human work. Wang et al. [Wan+23] raise concerns about students copying over solutions, given that GPT usually provides working solutions in introductory courses. Radošević et al. [ROS10] experienced how programming students often avoid grasping the concepts by relying on shortcuts such as rote memorization or copying from peers. Furthermore, they remark that for uniform programming assignments, it is challenging for educators to verify the originality of a student's work. According to Becker et al. [Bec+23], the advent of AI-based code generators has compounded these issues, enabling students to produce code that achieves high marks with minimal effort, bypassing the risk of asking similar code from peers. Therefore, they warn to urgently review the educational practices in light of these new technologies. However, Wang et al. [Wan+23] found through 6 interviews that most instructors had not yet adapted their courses to the advent of freely available language models, because they were lacking the effective strategies or tools.

2.3 Skill modeling

An important part of providing personalization in ITSs is modeling the student, which can include emotional state, preferences, etc. We limit ourselves to skill modeling in this section, as this is most closely related to our work. Skill modeling has the basic goal of modeling knowledge and skills to estimate a learner's current knowledge state and use that to predict future performance [Pel17]. Early approaches utilized graded standardized test summaries, so-called **Classical Testing Theories (CTTs)** [Mag+21]. In Section 2.3.1 and Section 2.3.2, we will discuss more recent approaches, namely **Knowledge Tracing (KT)**, which pursues the evolution of a learner's knowledge, and **Item Response Theory (IRT)**, which facilitates the estimation of latent-knowledge mastery levels.

2.3.1 Knowledge Tracing

Trying to determine how to effectively track the learning progress of a student through their (online) interaction with teaching materials is known as the **Knowledge Tracing (KT)** problem [AWN23]. Abdelrahman et al. provide an overview of the broad KT-landscape and its applications, to which we refer the reader for a more elaborate take on KT.

Bayesian Knowledge Tracing (BKT) is one such approach that knows many variants [AWN23] in which the latent knowledge can be modeled by a **Hidden Markov Model (HMM)** learning from historical data [BWP21]. Central to models for BKT is the Bayes' theorem, which claims that, for two events A and B, the following holds [AWN23]:

$$p(A|B) = \frac{p(B|A)p(A)}{p(B)}$$

Standard BKT The first BKT-models (referred to as Standard BKT) associate a skill with a binary knowledge state: it is either learned or unlearned [CA94]. A transition is possible from the unlearned to the learned state prior to practice, through reading the text, or at each opportunity to prove mastery of a skill. Transitions in the opposite direction are not possible however [CA94] (i.e., the probability of transition from a learned state to an unlearned state is always zero, even though that is not realistic), thus the model is overlooking the forgetting theory [BWP21]. Corbett and Anderson [CA94] also note that a learner may make a mistake (slip) while in a learned state, or guess correctly in an unlearned state. The following formulae from Badrinath et al. [BWP21] more clearly demonstrate the relation of the probabilities:

$$\text{prior} = P(L_0)$$
$$\text{learn} = P(T) = P(L_{t+1} = 1|L_t = 0)$$
$$\text{guess} = P(G) = P(obs_t = 1|L_t = 0)$$
$$\text{slip} = P(S) = P(obs_t = 0|L_t = 1)$$

Note that while $P(L_0)$ denotes the prior parameter, the probability that the student has mastered the skill at time step t is defined as $P(L_t)$. Bayesian Knowledge Tracing updates $P(L_t)$ given an observed correct or incorrect response to calculate the posterior using the following formulae: [BWP21]

$$P(L_t|obs_t = 1) = \frac{P(L_t)(1 - P(S))}{P(L_t)(1 - P(S)) + (1 - P(L_t))P(G)}$$
$$P(L_t|obs_t = 0) = \frac{P(L_t)P(S)}{P(L_t)P(S) + (1 - P(L_t))(1 - P(G))}$$

The updated prior for the following time step, which incorporates the probability of learning from immediate feedback and any other instructional support, is defined by: [BWP21]

$$P(L_{t+1}) = P(L_t|obs_t) + (1 - P(L_t|obs_t))P(T)$$

As noted by Pelánek [Pel17], parameter fitting for the learner probabilities $P(L_0)$, $P(T)$, $P(G)$, $P(S)$ is typically done using the standard expectation-maximization algorithm.

Variants of BKT The standard BKT model, however, has no parameters specific to students [AWN23]; it assumes the heterogeneity of the learner population, and thus the individual differences are not reflected among students in learning and performance [CA94]. As a result, the standard BKT model will underestimate the learning and performance for above-average students, and overestimate the learning and performance for below-average students[CA94]. This limitation can be an important factor when the set of skills is coarse-grained, because different students may excel in a different set of skills [Pel17]. To alleviate this limitation, further research explored student-specific parameters categorized as **Individualized BKT**, which is overviewed by Abdelrahman et al. [AWN23]. According to them, the individualized models can provide better correlation between actual and expected accuracy among students, leading to more effective decisions or improving the accuracy of predicting student performance. Other extensions they have overviewed try to tackle the limitation that for a question only one skill is assumed to be required. This extension to the modeling is referred to as **Dynamic BKT (DBKT)**.

Deep KT Finally, **Deep Knowledge Tracing (DKT)** pioneered the use of deep learning for knowledge tracing [AWN23], which has the advantage that these models do not require student interactions to be manually labeled [Wan+17]. Piech et al. [Pie+15] described how to use traditional **Recurrent Neural Networks (RNNs)** to map an input sequence of vectors $x_1, ..., x_T$, to an output sequence of vectors $y_1, ..., y_T$. This is achieved by computing a sequence of 'hidden' states $h_1, ..., h_T$ which encodes the sequence information obtained from previous interactions, useful for future predictions [Pie+15; AWN23]. At each time step t, the model calculates the hidden state h_t and the probability of the student correctly answering p_t as follows: [AWN23]

$$\mathbf{h}_t = \tanh\left(\mathbf{W}_{hx}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h\right),$$

$$\mathbf{p}_t = \sigma\left(\mathbf{W}_{ph}\mathbf{h}_t + \mathbf{b}_\mathbf{p}\right),$$

Both $tanh = (e^{u_i} - e^{-u_i})/(e^{u_i} + e^{-u_i})$ and the standard sigmoid function, $\sigma(\cdot) = 1/(1 + e^{-u_i})$, are activation functions applied elementwise [Pie+15; AWN23]. The model is parameterized by an input weight matrix W_{hx}, recurrent weight matrix W_{hh}, initial state h_0, and readout weight matrix W_{yh} [Pie+15]. Biases for latent and readout units are given by b_h and b_p.

A more complex alternative to RNNs are **Long Short Term Memory (LSTM)** networks as they often prove more powerful than RNNs. These models more naturally retain information for many steps, which is why they are believed to be easier to train [Pie+15]. LSTM models are, for example, used in both works by Shi et al. [Shi+22] and Wang et al. [Wan+17] that attempt to use DKT in programming courses. The formulae for these models can be consulted in the work of Piech et al. [Pie+15].

Datasets and tools Getting started with the recent approaches to KT has been simplified, with researchers releasing their models in public code repositories, which is important for reproducing and verifying results. Badrinath et al. [BWP21] created a very accessible and well-documented Python library (pyBKT) that allows to easily test BKT and its extensions on two standard datasets (the KDD and Assistment datasets), or any other manually labeled dataset. On the other hand, programming datasets to which the KT tools are applicable are not as easy to find. Liu et al. [Liu+22], for instance, remark that CSEDM is the only college-level, publically available dataset to their knowledge that contains actual code submissions. Gupta et al. [GKS19] provided their collected dataset of C programs written by students for 29 different programming tasks in an introductory programming course. Wang et al. [Wan+17] used Hour of Code submissions, which are formed through students' drag-and-drop operations of code constructs. Finally, Freitas et al. [Fre+23] recently made a dataset available with submissions for an introductory course on Python.

2.3.2 Item Response Theory

Another approach to skill modeling is **Item Response Theory (IRT)**. Research also has ground in applying this in programming, although we will not include these applications. Different to KT is that IRT is typically used in formal evaluation settings (standardized tests or exams), and thus does not expect skills to change during the the evaluation [Pel17].

According to Abdelrahman et al. [AWN23], IRT is built upon the following assumptions:

1. The probability that a student correctly answers an item (question) can be formulated as an **item response function (*IRF*)** based on the parameters of the student and the item;
2. The *IRF* monotonically increases with respect to the learning ability of a student;
3. For a student, items are considered conditionally independent [AWN23].

Among the literature reviewed, the basic **one parameter logistic (1PL)** model, or Rasch model, seems fundamental to IRT [Pel17; Mag+21; AWN23]. Using the standard logistic function $\mathcal{L}(x) = 1/(1+e^{-x})$ [Pel17] as the *IRF*, the probability p_{ij} that a student i answers an item j correctly is defined as:

$$p_{ij} = \mathcal{L}(\theta_i - b_j) = \frac{e^{(\theta_i - b_j)}}{1 + e^{(\theta_i - b_j)}}.$$

where $\theta_i \in \mathbb{R}$ and $b_i \in \mathbb{R}$ are scalars that correspond to the learning ability and the item difficulty, respectively [Mag+21; AWN23]. Later extensions include 2PL models, which add another multiplicative scaling parameter to differentiate between high-capacity students and low-capacity ones [Mag+21]. Further, 3PL models add another scalar that corresponds to the probability that an item is guessed correctly. Abdelrahman et al. [AWN23] provide an example of a 4PL model that further includes slipping:

$$p_{ij} = c_j + (d_j - c_j)\mathcal{L}(a_j(\theta_i - b_j)) = c_j + (d_j - c_j)\frac{e^{a_j(\theta_i - b_j)}}{1 + e^{a_j(\theta_i - b_j)}},$$

where a_j is a discrimination parameter to model how well an item j can differentiate students, c_j is a guessing parameter to model the effect of guessing, and d_j is a slipping parameter to model the effect of careless errors [AWN23].

Although an item in the original IRT corresponds to a question involving a single existing or predominant skill (therefore unidimensional), later works have been generalized to consider an item that may involve multiple skills [AWN23], by using vectors instead of scalars [Mag+21]. According to Zaffalon et al. [Zaf+22], research has shown that **Multidimensional Item Response Theory (MIRT)** model adapts better to real data, because, in education, students' responses are determined by more than one skill at the same time.

2.4 Formative and summative feedback in programming

Another important part of the learning process is assessment of students' exercises and providing feedback to students. As noted by Koutcheme [Kou22], feedback is typically considered from two perspectives: summative feedback, which often focuses on the end result (and sometimes includes grading), and formative feedback, which provides guidance during the learning process. Traditionally, in computer education research, automated feedback has emphasized summative aspects, such as correctness [Kou22], but there is a growing trend to utilize assessments not only to measure knowledge but also to capture learning trajectories and processes, thereby shifting from summative evaluations to formative measures [RW16]. In the following sections, we discuss approaches to providing feedback to students in programming courses.

2.4.1 Correctness assessment and static analysis

As noted by Koutcheme [Kou22], automated assessment traditionally focuses on the correctness of programming exercises, inferred through unit tests. These are predefined tests usually defined by human instructors that compare the output of a routine with the expected output, given certain inputs. Senecode is one such example of an automatic program grading tool for an introductory programming course in Python that additionally provides a progress percentage [Sal20]. Online compilers such as JXXX and editors further facilitate the assessment process by allowing students to develop and test code without dealing with installing software on their platforms [ROS10]. Finally, self-hostable tools like Judge0 provide robust correctness assessment out of the box, though some research prototypes, such as the one by Nguyen and Hoang [NH24], simply make use of isolated namespaces to execute code. However, it is remarkable that contemporary instructors still spent significant time on configuring their own grading tool, using the Web-CAT's plugin-based system and Github Actions to run an internally developed grading tool [MHP23].

With the advent of publically available LLMs, Becker et al. argue that current assessment approaches should shift focus from code correctness to code quality and style [Bec+23], given the ability of LLMs to produce working solutions. This focus on additional quality metrics is already being demonstrated by some systems such as ASSYST that evaluate the efficiency, style, complexity [ROS10] or even excessive coding [EKR17] additionally to correctness. Edwards et al. [EKR17] used the Web-CAT plug-in-based system to execute static analysis using tools like Checkstyle and PMD to provide feedback on documentation, formatting, naming, style, etc. in addition to the feedback of correctness. However, through informal interviews, students indicated that formatting and documentation errors are reported too often, which is why students started ignoring them and miss situations where these tools report problems that should be addressed.

2.4.2 Feedback based on program structure

Although a lot of research focuses on plagiarism or explaining compiler theory when utilizing **Abstract Syntax Trees (ASTs)** for education, some research also utilizes them to give more abilities to traditional approaches to assessment. An AST is a tree representation of compilable code commonly used, where each node represents an element in the code, which will be connected to associated nodes. Typically, more broad objects, such as functions and for loops, will typically appear closer to the root of the tree, while variables, operators, and constants will appear closer to the leaf nodes of the tree. As an example by Moore et al. [Moo+22], the leaf nodes branching off a function node will represent different for loops, while loops, and variables located within that function.

Some research has been conducted on comparing functions with model functions [Liu+19; Sal20]. Both Liu et al. and Salazar Paredes acknowledge that determining whether two programs are equivalent is an undecidable problem. While not directly using ASTs, Liu et al. [Liu+19] use a constraint solver to efficiently find an input that leads to two deviating execution paths (i.e., two different semantic traces of if tokens) for the student's and the reference program. This has the benefit that instructors would not have invest time in providing sets of test inputs. Salazar Paredes [Sal20] developed an algorithm that is capable of determining how strongly equivalent two programs are through syntax comparison. One of their interesting findings was that programs can be syntactically similar, yet semantically different, with one producing correct results in contrary to the other.

Hovemeyer et al. [Hov+16] did analysis on student submissions through Control-Flow ASTs (CFASTS), which are pruned to only contain control-blocks. They had many interesting findings. For instance, even simple problems can yield a surprising number of CFAST, indicating the significant variation that exists in student submissions, but also noting that most submissions can be mapped to a few CFASTS. Those CFAST that are mapped to fewer submissions tend to be larger and more complex, but they can still respond to correct solutions. This is consistent with earlier research by Luxton-Reilly et al. and later research by Salazar Paredes [Sal20] noting many students have submissions in a few families of solutions and most of the ones that are not, are in their own solution family. Moreover, Hovemeyer et al. [Hov+16] posit that additional requirements for conditional logic can lead to an exponential increase in the number of solutions to a problem, which Liu et al. [Liu+19] believe to be challenging to address in AST-based solutions. These findings are particularly relevant for our research, as we are comparing submissions against some model solutions provided by instructors, and attempting to capture those by comparing AST-patterns.

Moore et al. [Moo+22] created tools to visualize temporal ASTs, which are AST in which nodes also maintain a timestamp of when they were added to the tree. They hypothesize that a tree, representing the code of an experienced programmer, would have its oldest nodes near the root and the newest nodes in the leaves, whereas a novice programmer would have the opposite effect or might have a random disbursement of color. This is because they assume that novices think more linearly without high-level vision of the solution's structure.

Finally, where previous research was more summative, some research has also been conducted in more formative approaches. Research collected by Hovemeyer et al. [Hov+16] clustered submissions on their features including control structures, and proposed to show students examples from other clusters. Hovemeyer et al. also collected research where students are provided hints by comparing their submissions' ASTs to the ones from submissions from other students. Liu et al. [Liu+19], however, collected research showing that AST-based methods are unable to provide students with hints for the root causes that lead to the failure.

Aside from its usefulness, it looks like the AST-based methods will mainly complement the unit testing rather than replace it. Salazar Paredes [Sal20] provide an example where two programs are syntactically the same, but semantically different, with only one of them passing correctness testing. According to Nguyen and Hoang [NH24], online platforms such as LeetCode and HackerRank already make use of AST-based code evaluation, although no other sources were found that confirm this. Tools like PMD also already allow instructors to write XPATH expressions that will be matched to the AST of the student submission, which Edwards et al. [EKR17] find both powerful and convenient, making its method similar to our work.

2.5 Challenges for AIED

Even though we have shown a lot of opportunities, the integration of AI in education and the digitalization thereof also comes with a lot of challenges that cannot be ignored. Many authors are concerned about privacy, security, overreliance, bias, inclusiveness, and the corporate push to integrate AI into education.

Privacy and security Most AIED platforms and applications are currently owned by the private sector, but there is little transparency and regulations regarding the data accessed or the privacy and security of the stored data [MG23]. Especially with the growing movement for personalized education, the risk grows of students being closely monitored [Mag+21]. This is especially worrisome, given that some governments already use AI systems for surveillance and oppression [Ros22]. There are also concerns about privacy and data security when integrating LLMs in education [BO23]. One of the reasons is that student data is often sensitive and personal, and entrusting this data to bigger companies would put it at risk of data breach attacks, hacks or unauthorized access [Kas+23]. As an example, a data breach at the student-tracking ed tech company Illuminate compromised the data of at least a million public school students and prompted New York City's Department of Education to ask schools to stop using their products [WMB24].

Overreliance AI in education also presents challenges that particularly concern overreliance. Code generation tools like Copilot may lead junior students to depend on auto-suggested solutions without engaging with problem statements or computational steps [Bec+23]. This reliance on LLMs can negatively impact students' critical thinking and problem-solving skills, and as a result amplify laziness and counteract the learner's interest to conduct their own investigations [Kas+23]. As remarked by Wang et al. [Wan+23], students relying solely on the answers provided by ChatGPT without verifying their accuracy may potentially develop flawed mental models, and therefore provide a method to use this limitation as an opportunity in generating multiple-choice questions. This overreliance is not exclusive to students. Kasneci et al. [Kas+23] note that regardless of the accurate and relevant information LLMs can provide, those models cannot replace the creativity, critical thinking, and problem-solving skills that are developed through human instruction. It is therefore important that these models are used as a supplement for human instructors, rather than their replacement. Many authors also remark or find by their studies that AI should not replace humans in the loop [BO23; WMB24] (i.e., they emphasize the importance of human oversight and involvement in the final decision) and that in its current form it can only augment the instructor [MG23].

Bias and fairness Williamson et al. [WMB24] note that AI can amplify existing forms of inequality in education by using datasets containing examples of historic bias and discrimination. This is also the case for LLMs [BO23] that can even be prompted to produce racist, denigratory, and otherwise harmful coding comments [Bec+23]. In a study collected by Baidoo-anu and Owusu Ansah [BO23], the authors note that the essay grading software may be unfair when certain demographics are underrepresented. Ensuring diverse training data and regular monitoring is, therefore, essential to mitigate these biases and prevent introducing automatic discrimination in education [Kas+23]. To address these biases, researchers pay significant attention to developing approaches to promote fairness, focusing on defining fairness and enforcing it in predictive algorithms. This includes preprocessing data, post-processing outputs, and applying regularizers during training to achieve a balance between fairness and accuracy [Mag+21]. Despite these concerns, a recent survey mentioned by Mallik and Gangopadhyay [MG23] showed that students rated algorithmic decision-making higher than human decision-making in admission decisions in both procedural and distributive fairness aspects.

Inclusiveness and democratization The use of AI in education also presents significant inclusiveness challenges, as highlighted in various studies. On one hand, the advent of MOOCs has broadened accessibility and diversified student populations [RW16], automatic code generation tools like AlphaCode have made programming more accessible [Bec+23], and AI-based translation tools allowed students to process material in other languages On the other hand, according to Maghsudi et al. [Mag+21], existing research shows that some subgroups of students, many of whom are also privileged with respect to conventional education platforms, would profit more from personalized education than their peers would. For example, as commented by Wang et al. [Wan+23], students who do not use assistants may be at a disadvantage when learning new concepts. Mallik and Gangopadhyay [MG23] also make a similar remark, saying that the AI deployment may alienate communities with limited means, which can further widen the gap in education. Beck et al. [BSH96] highlight that estimated construction time indicate that 100 hours of development translates to 1 hour of instruction.

Integration push and AI regulation Popenici and Kerr [PK17] note that due to the MOOC hype, there is already a temptation in reducing expensive academic staff with a shift towards casual and short-term contracts that is aggressively pushed in Australian universities. In the UK, similar trends exists, where teaching is outsourced, much like cleaners and catering staff. Given that AI is promising to reduce administrative burden and make time available to instructors [WMB24], this will likely not put a hold on these trends. Williamson et al. [WMB24] also claim that the high operational costs of AI integration could negate potential financial savings, since schools may redirect funds to tech providers instead. Furthermore, major technology companies such as Google, Microsoft, and Meta are racing to fend off regulation to integrate AI into their platforms, while research depends on the availability of AI models that are costly to run locally. This way, Popenici and Kerr [PK17] rightfully remark that those who control algorithms that run AI solutions now have unprecedented influence over people and every sector of a contemporary society.

Chapter 3

Personalizing learning trajectories

As previously mentioned in Section 2.1, students have different learning rates influenced by many factors over time [TSB21], and thus also have considerable differences between each other. Furthermore, it is useful to hypothesize that certain students are conceptual learners, whereas others are more visual learners, as found by Schiaffino et al. [SGA08]. With a one-size-fits-all trajectory, some students will inevitably be making exercises that are slightly ahead of their understanding (what Schiaffino et al. would describe as 'global learner'), and some students will be making exercises that will not challenge them enough.

In this chapter, we will provide an overview of the concepts of a system that models students' knowledge to personalize their learning trajectory. The desired outcome is that students will spend their time more efficiently on the exercises and as a result maintain more motivation to progress in the course. With a more efficient progression, they are hopefully encouraged to continue making more exercises, or to make more challenging exercises at the microscale (described in Section 2.1.2) that they otherwise would have no time for, ultimately increasing the learning gain.

Even though the activities of reading course material and solving exercises to practice lectured concepts are common among courses in higher education, we will focus mainly on solving exercises in an introductory course on programming when discussing the applicability of personalized learning trajectories. In Chapter 6, we will continue this focus, as we describe how these concepts can be applied to a programming course using concrete student modeling, where some assumptions are made on the type of exercises. Nonetheless, the concepts presented in this chapter are sufficiently high-level to apply them in fields other than programming.

In Section 3.1, we will discuss several use cases in which personalizing students' learning trajectories may be useful. In Section 3.2, we will further discuss applications in which the system may interrupt the student during a learning activity.

3.1 Dynamic path proprosing

In this thesis, we make a few assumptions about a course. We assume that whatever is provided in a typical day for a certain course (e.g., lectures) all surrounds a handful of somewhat related concepts that are all addressed in one chapter. For each chapter, exercises are provided for the student to train these concepts, as well as reading material. So conceptually, a traditional *course* consists of a few *chapters* and each chapter consists of several *exercises* and references to reading *sections*.

A student can progress through exercises in a certain chapter and attempt to solve them until all requirements that indicate a good solution are fulfilled. However, if a student does not make the first exercises well, this student may have more difficulties progressing through the next exercises, where the gain of particular knowledge may be expected, increasing the barrier to solve the next exercise. **Dynamic Path Proposing (DPP)** is the concept of an agent adding small intermediate exercises to further maximize the learning gain for students progressing through exercises, as well as making the learning trajectory more efficient for students getting stuck.

Figure 3.1 further details the example. When the student solved the first exercise, `E1`, a lot of trial-and-error was detected in achieving the correct outcome, and it is unclear whether this outcome was the result of thorough understanding or an accident. Therefore, the agent proposed to evaluate the student *to assess the level of understanding* through `E1?`. When the answer again does not prove understanding, a new intermediate exercise `E1+` is provided, asking for an extension to the exercise. When this is realized successfully, the student is proposed to continue to the next exercise, `E2`, which was solved successfully. With the third exercise, `E3`, another issue came to light. The student got stuck and had not been able to overcome the barrier in a long time period. Therefore, the student is interrupted and proposed doing intermediate exercise `E3_`, followed by intermediate exercises `E3+` and `E3*` *to get the student back on track* before continuing `E3'`.

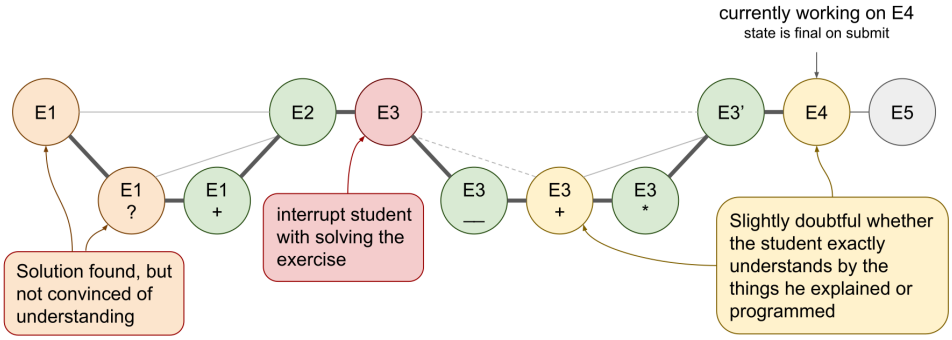
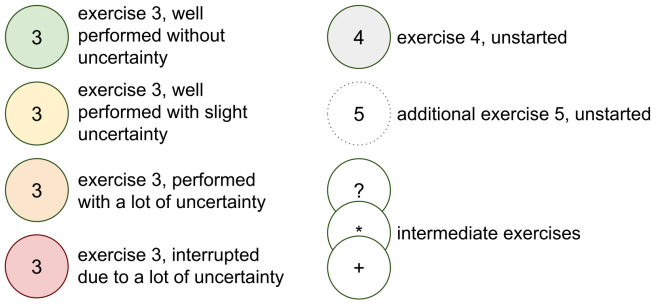


Figure 3.1: An exemplar scenario highlighting the important concepts of Dynamic Path Proposing, with `?`, `+`, `__`, `*` denoting intermediate exercises.

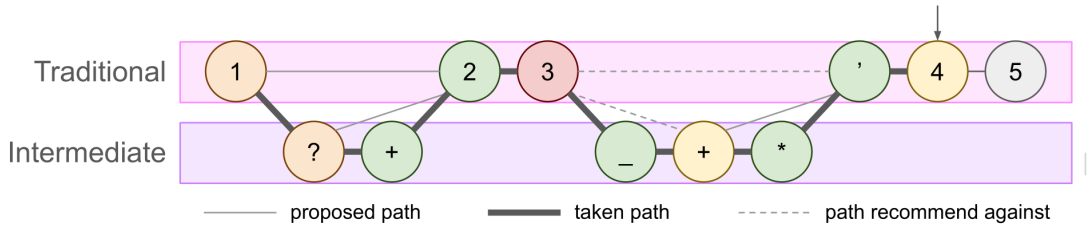
To express certain scenarios in a student's trajectory, we captured the previous scenario in a small design language, as shown in Figure 3.2, and we will use this language throughout the rest of this chapter. Figure 3.2a gives an overview of the several states an exercise (visualized as a circle) can be in. The four colors represent the system's predicted level of understanding. The exercise currently being solved has a blue arrow at the top. In the case an exercise has not been attempted yet, the exercise will be represented with a gray color. An additional (optional, 'advanced') exercise has a dotted border. Finally, intermediate exercises can be represented by several symbols. In the case of the programming course, symbol `?` will represent a simple question to test a concept. Symbol `+` represents a modification to the original exercise, where the student has to make small changes. Symbol `*` is for debugging an alternative solution with a (hidden) bug. Finally, symbol `_` is a template exercise in which the user has to fill in certain parts of code.

Figure 3.2b details how these exercises can be connected together. The proposed execution flow between exercises is shown as a gray path. The exercise chosen by the student is shown with a bold line. When a path is specifically recommended against, it will be represented with a dashed line. Furthermore, when describing a student's progression, we will place exercises in two bands as in the figure: one for traditional exercises (pink) and one for intermediate exercises (purple). These bands will not be visualized to the students, but are merely present in the picture for illustrative purposes. Depending on the level of understanding, multiple intermediate exercises

or none at all may be proposed and/or recommended.



(a) The states an exercise (circle) can be in (left and right top), and intermediate exercises (right bottom).



(b) Paths between exercises, visualized on top of the traditional and intermediate bands.

Figure 3.2: The DPP design language.

The first scenario already illustrates the potential for improvements, but more cases exist. Figure 3.3 shows how a student has been progressing through exercises that are estimated to be solved well. Given that in this scenario, the student has done this very quickly compared to his or her peers, the student is recommended to continue with additional exercises. On the other hand, if the same progress was observed, but the time it took is much longer (regardless of whether additional intermediate exercises have been made or not), the student could instead be suggested to finish exercises that have a bigger impact or work on another course.

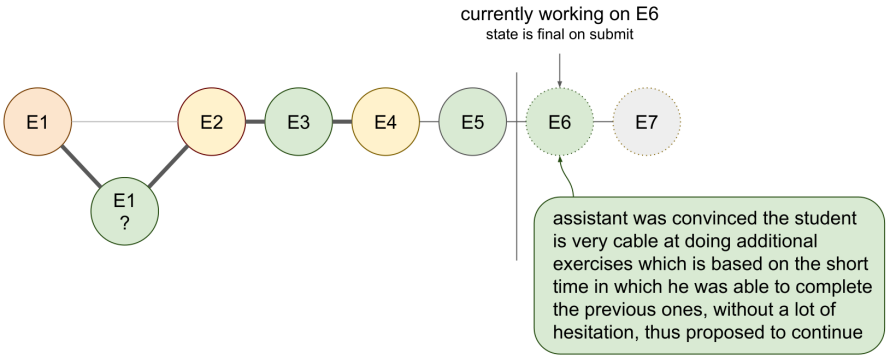


Figure 3.3: A scenario in which the agent correctly proposes a fast student to work on additional exercises.

Figure 3.4 demonstrates a different scenario in which the student has progressed through the first three exercises of a chapter and realized during the fourth that (s)he misunderstood a certain concept that went unnoticed in the first three exercises. Alternatively, the detection can be poorly, while the student is unable to make progress and may feel (s)he is lacking something without knowing what exactly. At that point, the expected understanding on the exercise conflicts between the student and the agent. In these cases, the student can request

feedback from the system and perhaps be proposed to (re)do an exercise that addresses these misconceptions.

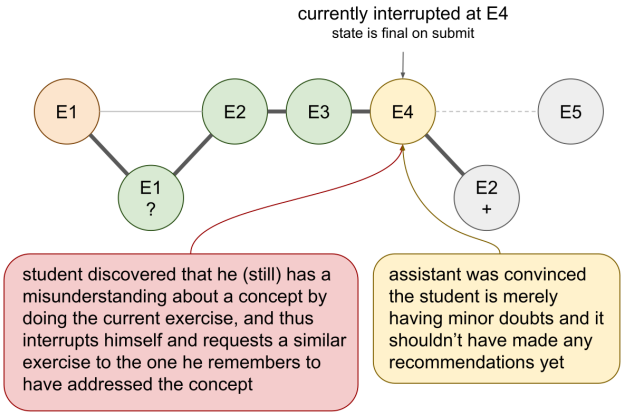


Figure 3.4: A scenario in which the agent incorrectly estimates the student doing well, unable to detect the student's misunderstanding of concept.

3.2 Proactive recommendation

An important aspect already briefly covered in the previous section is 'interrupting the student' when the algorithm detects knowledge gaps that are necessary to solve the exercise. Figure 3.1 demonstrates how intermediate exercises can be used to quickly get a student back on track, but this is not the only useful form. Figure 3.5 shows a situation in which a student has completed certain exercises from a previous chapter with low certainty of understanding, and given that the student fails at concepts from that chapter, the system will redirect her/him to those chapters. (Intermediate exercises have been omitted in this example.) Alternatively, the student may be failing a certain concept that is expected to be learned by reading the chapter's course material. In this case, the algorithm can recommend revising a certain section.

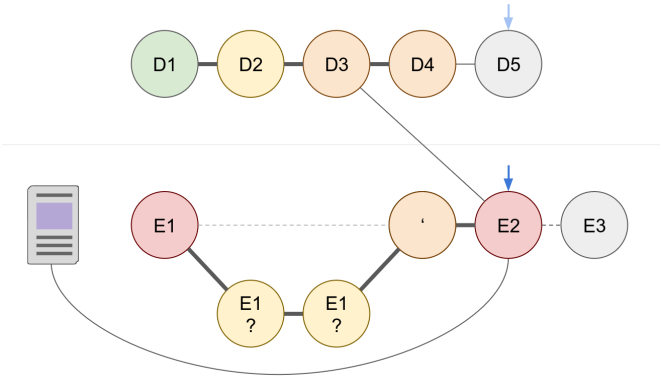


Figure 3.5: A dually overlaid scenario in which the agent interrupts the student and proposes to stop the current execution of exercise `E2` 1) to refresh `D3` as it was estimated negatively, or 2) to reprocess reading material required for `E2`.

To make effective and proactive recommendations, two challenges must be taken into account. First, reliable metrics are required to decide *when* a student should be interrupted, if at all. Doing so at inappropriate times will be frustrating. For a programming course, research on iList by Fossati et al. [Fos+15] already showed how to detect uncertainty when a student is solving an exercise. However, this approach works on small exercises and requires lots of data

collection. Instead, other metrics can be used, such as excessive time spend on an exercise with no submissions passing the unit tests. Looking at the number of submissions can also be useful, which is often the main approach in KT-approaches discussed in 2.3.1. Hovemeyer et al. [Hov+16] remarked that this may indicate trial-and-error behavior. However, they also warned that given a clear submit button, does not imply that students therefore intentionally send each submission, as missing response from the interface or the network may make them think that a submission was not sent; or students may accidentally send images.

The next challenge is to decide *what* proactive actions should be taken (e.g., recommending reading, recommending exercises) that will actually lead to more or better knowledge gain than spending the same amount of time on continuing the current exercise (assuming the student would not give up). Giving bad recommendations makes them annoying or may result in the student simply ignoring the recommendations by default.

In the next chapter, we will explore the abilities of OpenAI's GPT to take the role of a personalizing agent, as described in this chapter.

Chapter 4

Exploring LLMs for personalized learning trajectories

Preparing numerous intermediate exercises is often unrealistic for instructing teams, as it poses a major administrative burden on them to first create these exercises and to keep them up to date afterward. The alternative is to have them generated by LLMs in real time, while the student is solving exercises, which allows for personalization. However, then the ML algorithms used for skill assessment will not have these exercises as part of their training data. The middle ground (i.e., having instructors review generated exercises) would ensure human review, but also consume a lot of time. In the end, a trade-off will have to be made.

In this chapter, we describe our exploration of using GPT3.5 as LLM through Perplexity to generate intermediate programming exercises (previously discussed in Section 3.1), as this was the only known way to freely prompt GPT at the time. In Section 4.1, we discuss how GPT can be used to generate similar (extended), template and debugging exercises. In Section 4.2, we discuss the ways in which GPT can be used to small question assessment based on student code. Additionally, we explore GPT's ability to decide upon when to provide proactive feedback in Section 4.3.

4.1 Generating intermediate coding exercises

For the first three use cases, we used the prompt template in Listing 4.1, focusing only on situations where the student has completed the exercise successfully but with convincing understanding, like the scenario discussed in Figure 3.1. Placeholder {`ExerciseDescription`} is consistently filled with the description of the exercise, in this case the exercise on prime numbers from Section 5.5. Placeholder {`Status`} is consistently something like "The student created a solution (see below) that covers all of our test cases, so we are convinced he got the exercise right. However, we detected a lot of uncertainty in the process of creating the solution." Placeholder {`Request`} is provided in each paragraph by tables and usually also includes the clarification "We want to verify if they actually sufficiently understands what they have done." at the end (or one similar to it), that we omit from the tables. Placeholder {`StudentCode`} reserves the position for the student's solution, consisting of functions, commented code and self-written test-cases. This code is exactly the same for each prompt, besides #8, where it is not provided at all. Placeholder {`Requirements`} is consistently something like "Your answer will be provided to the student as is, so do not hint on how it can be solved, only if that's part of the exercise. Just provide the exercise as if it's directly from a book. Also, keep in mind that the student only has seen the following concepts: if, for, function, lists, variables.", which is to ensure that GPT is constrained to the set of constructs intended to be used.

````
A student got the following exercise:

> {ExerciseDescription}

{Status}

{Request} Here is the code the student used to solve the exercise:

```py
{StudentCode}
```

{Requirements}
````

Listing 4.1: The prompt template used for similar, template and debugging exercises.

Extended exercise The first type of exercises are extensions to the provided exercise (e.g., ask to implement an additional function, ask to also support another case or input, etc.), to provide additional training of the concepts. Table 4.1 provides an overview of the text in placeholder {`Request`}. For request #2, GPT suggested implementing new functions that are variations on the current methods to implement, such as the self-explanatory `count_primes` and `nth_primes`, and `prime_factors` which returns a list, instead of printing. GPT, however, additionally provided tips that may take away the mental learning effort. With a slightly modified request (#3), this was not present anymore, and GPT suggested implementing `sum_primes` and `nth_prime` as an intermediate exercise. Note that these exercises can be solved with the (weak) constraints on constructs we provided, showing opportunities that can be further explored.

#2	Can you generate a different exercise (with a similar degree of difficulty) with some template code he has to fill in, in order to test what he learned so far? They can be just a few lines he has to complete.
#3	Can you generate a different exercise (with a similar degree of difficulty) with some template code he has to fill in, in order to test what he learned so far? They can be just a few lines he has to complete in a partial solution.

Table 4.1: The content of {`Request`} for each prompt to similar exercises.

Template exercise The second type of exercises we tried to generate are coding solutions with blanks left for the student to fill in to make it work. The blanks may replace variable names, constants, code lines or code blocks, similar to exercises found in DataCamp courses containing `___` as blank placeholders. Table 4.2 provides an overview of the text in placeholder {`Request`}. Request #1 was initially intended to create an extension of the exercise, but presumably, due to the word 'template', GPT produced a template exercise instead for `is_prime`. While GPT was creative in selecting 6 places to put a blank, the exercise itself is not useful since it uses the student's solution as a templating exercise. The student will already know the answers, given that in our scenario it was just submitted before the interrupt happened. When directly asking to create new exercises and adding blanks to them, GPT seems to miss the intent and provides some solutions that are small modifications to the student's solution.

Debugging exercise The third type of exercises are different solutions to the exercises that have a non-obvious bug hidden in them, so the student has to search for this bug. This exposes them to a different solution, while training debugging skills on unknown code. Table 4.3 provides an overview of the text in placeholder {`Request`}. For request #6, GPT added seemingly useful bugs. But just like template exercises, GPT took the student's original code, instead of coming

#1	Can you generate a similar exercise with some template code he has to fill in, in order to test what he learned so far? They can be just a few lines he has to complete.
#4	Can you generate a better solution than he has (or just different in case there is little to improve) and remove a few expressions or statements so he has to fill them in?
#5	Can you generate a better (or just different) solution than he has and replace a few expressions or statements with '___' in the solution so he has to fill them in?

Table 4.2: The content of {`Request`} for each prompt to template exercises.

up with a new solution. Re-executing the prompt (#7) similarly added bugs to the functions. In an attempt to prevent GPT from reusing the student's code, it was omitted in request #8, and additionally clarified to use a distinctly different solution. To our surprise, GPT provided the original code, proving that it remembered this code from the other sessions.

#6, #7	Can you generate a bugged solution to the exercise, so that he has to search for the bug. The solution should not crash at runtime, it should simply give the wrong results.
#8	Can you generate a bugged solution to the exercise, so that he has to search for the bug. The solution should not crash at runtime, it should simply give the wrong results. Your answer will be directly provided to the student, so `**do not hint on how it can be solved or explain the code**`, only if that necessary to solve the exercise. Just provide the exercise as if it's directly from a book.

Table 4.3: The content of {`Request`} for each prompt to debugging exercises.

4.2 Assessing intermediate code-based questions

In this section, we look into the opportunities of using LLMs for automatic question assessment, similarly focusing on the scenario where the student has solved the exercise. We used GPT to assess whether questions were answered correctly and to generate multiple-choice questions. Similar to Listing 4.1, we used Listing 4.2 as the template. Placeholder {`ExerciseDescription`} is filled with the exact same description. Similarly, placeholder {`StudentCode`} is the student's solution consisting of three functions, but without comments or tests. Placeholders {`InstructorQuestion`} and {`StudentAnswer`} are filled with self-explanatory content, and omitted when generating multiple-choice questions. Finally, {`Request`} is analogous, and {`Request`} is filled with "Write the answer as if you are directly talking to him, because your answer will be directly provided to him."

Open-ended question With open-ended questions, it is risky to let GPT generate both the question, the answer and perform the assessment, as it may hallucinate at times. Therefore, the instructor can provide a code-dependent question, and GPT can be used for assessment. In case the question is not dependent on the student's code, the instructor can also provide the answer and GPT is best used to verify the answer's correctness given such a model answer, which further reduces the risk of hallucination. Table 4.4 provides an overview of the text in placeholder {`Request`}. In both prompts, the same example question and answer are used: placeholder {`InstructorQuestion`} is filled with "When asked 'explain in exact detail what `all_primes_upto` does as implemented by your code'." and placeholder {`StudentAnswer`} with "In the function, primes will be made an empty list. Then a for loop will check if variable i is prime. If it is, it will be added to the empty list, else, it will be ignored. i is 0 at the start and

````
A student got the following exercise:

> {ExerciseDescription}

He provided the following code as a solution:

```py
{StudentCode}
```

{InstructorQuestion?} He answered the following:

> {StudentAnswer?}

{Request}

{Requirements}
````

Listing 4.2: The prompt template used for similar, template and debugging exercises.

is incremented every time after the if test is completed. Finally, the list of primes is returned." For request #9, GPT suggested to further clarify the answer and did not provide a grade. For request #10, GPT was given an example score, and returned the same positive score, along with some 'minor errors' with detailed explanations. These findings are insufficient to make a proper conclusion, thus real-world question-answer pairs, associated with scores, are required for further investigation.

#9	Can you grade this feedback and add argumentation for why the student would get such a grade for his explanation?
#10	Can you grade this feedback and add argumentation for why the student would get such a grade for his explanation? Write the answer as if you are directly talking to him, because your answer will be directly provided to him. Also, put the grade separately on the first line with no other words in that line, like "4/5".

Table 4.4: The content of {`Request`} for each prompt to assess a student answer.

Multiple choice question Using the multiple-choice format allows for easier integration of the output from GPT in automated environments. Table 4.5 provides an overview of the text in placeholder {`Request`}. For request #11, GPT correctly created a multiple-choice question, but it was not useful for post-assessment to verify understanding, as the choices were too basic. For request #12, GPT asked which implementation for `is_prime` is correct, and provided three different implementations, one of which was the student's solution. Furthermore, GPT claimed that only the choice with square root optimization is correct, which may conflict with the way the exercise is corrected. For requests #13, #14, #15, and #16, GPT created a question asking for the correct output choice for `factorize(12)` or `factorize(30)`, which can be useful for more complicated algorithms that cannot be calculated mathematically (unlike factorizing a number). For request #13, GPT corrected the flawed student implementation (although its correction crashes as well), to provide a valid multiple-choice question with useful choices. For request #14, GPT did not correct the code, and provided a valid multiple-choice question (although it also did not state which was correct). For request #15, GPT provided the expected answer for `factorize(12)`, an error choice and a bogus choice, and correctly identified the middle to be the correct choice. For request #16 the student's code was changed not to crash by introducing a bugged initialization. Remarkably, GPT provided plausible choices for

`factorize(30)`, while the code would never print any of such choices.

#11	Could you generate a multiple-choice question with three answers to test the student's understanding, one of which is correct?
#12	Could you generate a difficult multiple-choice question with three answers to test the student's understanding, one of which is correct?
#13	{#12} If the code submitted by the student contains bugs, do not correct them.
#14	{#13}, neither mention they are present! Simply make the execution provided by his code part of the answers.
#15, #16	Could you generate a difficult multiple-choice question with three answers to test the student's understanding of his own code, one of which is a valid/correct answer to the question? If the code submitted by the student contains bugs, do not correct them, neither mention they are present! Simply take this into account when creating the possible answers.

Table 4.5: The content of {`Request`} for each prompt to template exercises. Texts for #13 and #14 have substitutions for previous texts.

4.3 Proactive feedback

Finally, GPT was also explored as an agent to provide feedback proactively. The idea is that GPT will get periodic updates (which may be based on time, or based on IDE actions, such as clicking 'Run'). We explored two similar approaches: 1) asking GPT to provide a global feedback message, that would only be shown when it has sufficient priority; and 2) asking GPT to provide remarks labeled with a priority and an identifier, that could be shown through specific IDE features. Listing 4.3 provides the used template with the same semantics as in the previous sections. The placeholder {`StudentCode`} is in this case filled with a simple, buggy implementation of `is_prime`, except for requests that provided a code update.

````
A student got the following exercise:

> {ExerciseDescription}

{Status}

```py
{StudentCode}
```

{Request}

{Requirements?}
````

Listing 4.3: The prompt template used for generating proactive feedback.

Prioritized message Table 4.6 provides an overview of the text in placeholder {`Request`}. The requests #17 and #18 are insufficient, as GPT does not clearly understand what is expected and starts providing implementations for all functions (including the ones that have not been attempted yet). This help may be useful when the student is completely stuck, but it takes away from the learning effort when students receive the solution too early. Therefore, requests #19, #23, and #24 ask to additionally mark the whole message with a priority, provided the time a student is working on the exercise. Request #19 used code updates shown in Figure 4.1 in one GPT-session to retain context. We found that GPT marks its messages to remark a "small issue" of high priority either way, even when specifying that only 5 minutes have passed.

Further investigation is required to better understand why this is the case, and how to further leverage GPT's remarks.

#17	is the student doing well, or is he heading the wrong direction?
#18	{#17} Do not give away code or important clues, as the answers will be used to directly help the student. Try to give a summary or hints.
$\#19_{base}$	Could you write an answer to this student, and flag it with a priority label ("LOW", "MEDIUM", "HIGH") at the start? The answer you provide him will be directly shown, but as a tip if it's of low priority that he can click on, and a message being displayed right away if it's of high priority. Do not give away code or important clues, as he may find the solution by himself, and that's part of the learning process.
#19, #24	{$\#19_{base}$} He has only worked on it for 5 minutes, and an average student takes between 20 and 40 minutes for this assignment.
#19	{$\#19_{base}$} He has only worked on it for 11 minutes, and an average student takes between 20 and 40 minutes for this assignment.
#19	{$\#19_{base}$} He has only worked on it for 15 minutes, and an average student takes between 20 and 40 minutes for this assignment.
#19, #23	{$\#19_{base}$} He has only worked on it for 23 minutes, and an average student takes between 20 and 40 minutes for this assignment.

Table 4.6: The content of {`Request`} for each prompt to template exercises.

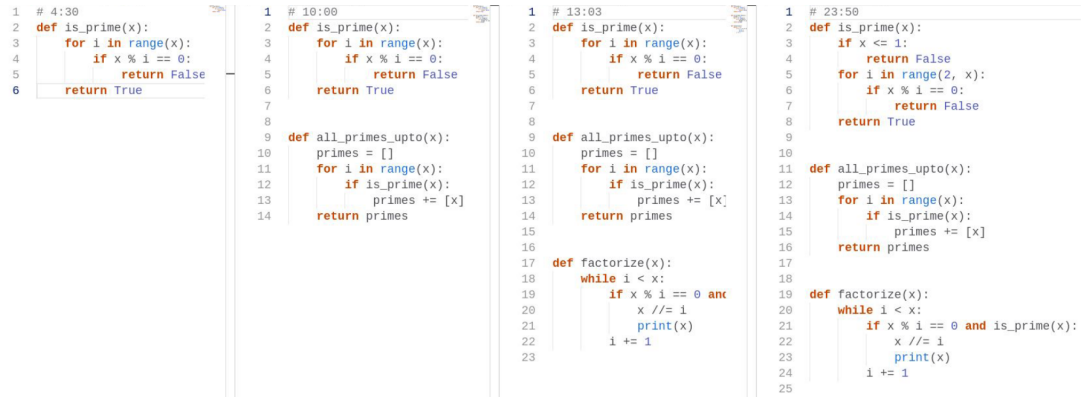


Figure 4.1: Synthetic snapshots of a student progressing on an exercise.

List of labeled messages Similar to the previous approach, we asked GPT to create a list of smaller remarks (in a sentence or two), associated with a priority. This way the information can be provided more cleverly to the student, other than constantly interrupting them. The idea is that these small messages can be shown in the IDE with 'info' messages hidden behind a counter of total (unread) messages, a light bulb shown next to a code line for hidden 'warning' messages, or a 'error' popup displayed right away to interrupt the student with sufficient priority.

Table 4.7 provides an overview of the text in placeholder {`Request`}. For both request #20 and #21, GPT provided 6 remarks of LOW and MEDIUM priority, respectively 2 and 3 of these remarks concern functions not yet implemented. For request #22, GPT provided 14 remarks, of which only one was of HIGH priority. The latter would have been useful if it was provided when the student was working past the provided expected duration. In each of the prompts, GPT is told that only 3 or 4 minutes have passed since the student started working on the current exercise. More importantly, it suggests fixing code that had not been provided yet, which may indicate that GPT is referencing other sessions and/or hallucinating. Ideally, GPT hints towards bugs rather than providing fixes for them. From these preliminary findings, it is

clear that this approach, requiring GPT to reason about when to provide what type of hint, is not optimal.

#20, #21, #22	Could you write a list of one-line comments/hints for each of the problems you see. Write these as if you are writing directly to the student, because these comments will be displayed to him (in different ways). Also, ad an ID to each of these comments so that I can keep track of them and label them with "LOW", "MEDIUM", "HIGH". An example: {Examples} The priority-labels should be decided upon as follows: {Rules} You should take into account the time taken by the student took so far (which you can find in the first comment of the file, e.g., '# minutes: 4') to write down what you can evaluate. If the student has only just started, the amount of comments should be low. You should compare this with the expected time for our students to reason how soon you should remark a certain problem for the first time, which I say is 20 to 40 minutes.

Table 4.7: The content of {`Request`} for each prompt to template exercises.

GPT was also asked to label these messages with an ID to keep track of them, as students may not have solved all of them, thus similar remarks that are phrased differently in consecutive updated lists would not be shown twice to the student. When continuing the same session for request #20, we found that this approach was effective, as all the old messages were reused in the updated list. The wording was surprisingly consistent over requests, but the priority labels changed.

The findings in this chapter indicate creativity, but require more prompt-engineering to enforce a format for automated extraction to mitigate hallucinations, and to limit additional artifacts. We must note that our testing is preliminary. For instance, we limited fine-tuning to 2 or 3 prompts, as we were doing a broad exploration. Furthermore, we only investigated one exercise, which is useful to better compare what does and does not work. Moreover, this exercise consists of multiple subtasks (defining functions) that are noise to GPT whenever we only focused on `is_prime`. We did not filter this noise to explore how GPT would handle more realistic situations. Finally, we refrained from tailoring the prompts to the specific exercise, but have not tested GPT's performance on other real-world exercises. Hence, further testing is necessary. For now, we can conclude that using GPT3.5 is not sufficient to provide reliable feedback. In the following chapter, we will focus on real submissions, so we provide in the chapter thereafter a different approach to personalizing the learning trajectory.

Chapter 5

Exploring student submissions

In order to provide an algorithmic solution to skill assessment, it is important to have a good understanding of student behavior when solving exercises. Making incorrect assumptions about the students' problem-solving may impede an agent's ability to provide effective recommendations. For this reason, we have gathered all the submissions (including intermediary submissions that have not passed unit tests) of seven peer students who enrolled in an introductory course in Python in 2019-2020, and all the submissions of one student who enrolled in the updated course in 2023-2024. The latter is a useful addition to the analysis, although some exercises may have been changed slightly, moved to another chapter, or have been omitted. If this was the case, the student's submissions were excluded from the analysis.

From the 43 exercises part of the original course, 7 were randomly selected among the first five chapters, of which all submissions were analyzed. We have selected a few exercises and submissions for discussion to demonstrate the variation within the submissions space. All the submissions shown have the variables renamed, and additional whitespace removed to make them consistent and easier to follow.

5.1 Summing coins

One of the first exercises looked into is part of the introduction chapter. It asks students to write an algorithm that prints out the total number of each type of coin to represent a certain amount of money, read from the terminal. Even though the submissions look very similar to each other, 69 of them were made by only 7 students. Important to note is that this number is inflated because one student made 30 submissions and two others made 13 and 12 submissions. In contrast, 14 submissions were made by the remaining 4 students. In their first attempts, several small bugs were present, which were squashed one submission at a time. We discuss a typical submission in Listing 5.1 that is close to a model solution, and also provide two submissions in Listings 5.2 and 5.3, in which students attempt to enforce the correct output in unintended ways.

Overall, the students struggled the most to print the total amount in fixed-point notation with two digital places. Most submissions contain a natural divide (`//`) like Listing 5.1, whereas a few use hacks to manually convert each digit to an `int` like Listing 5.2. A few submissions attempt to format a total result with `round` like Listing 5.3, and/or use other functions/constructs that are not intended for the introductory exercise (i.e., using `str.find` to separate a result string on `"."`, or using `"%.2f" % euro`) to enforce two decimal places. There are, however, a few commonalities among the submissions. For example, most of them convert the input right away to an integer. Although not as common, the input gets multiplied right away to the value of the coin read. The submissions end with a print statement, that contains `"You have"`, `"euro!"`, and `end=""`.

```
cents_1  = int(input("1 euro cents: "))
cents_2  = int(input("2 euro cents: "))
cents_5  = int(input("5 euro cents: "))
cents_10 = int(input("10 euro cents: "))
cents_20 = int(input("20 euro cents: "))
money = cents_1 + 2*cents_2 + 5*cents_5 + 10*cents_10 + 20*cents_20
euro = money // 100
cent = money % 10
cents_10 = (money % 100) // 10
print("You have ", euro, ".", cents_10, cent, " euro!", sep="")
```

Listing 5.1: An example of a typical submission.

```
cents_1  = int(input("1 euro cents: "))
cents_2  = int(input("2 euro cents: "))
cents_5  = int(input("5 euro cents: "))
cents_10 = int(input("10 euro cents: "))
cents_20 = int(input("20 euro cents: "))
total = (cents_1*1)+(cents_2*2)+(cents_5*5)+(cents_10*10)+(cents_20*20)
hundreds_digit = total % 10
tens_digit = total % 100
tens_digit_str = str(tens_digit)
euro_amount = (total-tens_digit)/100
euro_digit = str(tens_digit_str[0])
print("You have ", int(euro_amount), ".", euro_digit+str(hundreds_digit), \
    " euro!", sep="")
```

Listing 5.2: An example of a student using `int` to enforce the correct output.

```
cents_1  = float(input("1 euro cents: ")) * 0.01
cents_2  = float(input("2 euro cents: ")) * 0.02
cents_5  = float(input("5 euro cents: ")) * 0.05
cents_10 = float(input("10 euro cents: ")) * 0.10
cents_20 = float(input("20 euro cents: ")) * 0.20
total = cents_1 + cents_2 + cents_5 + cents_10 +cents_20
print("You have ", round(total, 2), " euro!", sep="")
```

Listing 5.3: An example of a student using `round` to enforce the correct output.

Using these similarities, we can conclude that basic automated token-based checks can be a viable solution to analyze these submissions. For instance, in the limited sample of submissions, it is sufficient to check whether the string part of the last print statement contains a `"."` to predict whether a submission follows the intended path regardless of it passing all unit tests.

5.2 Calculating a summation

In another exercise, students are asked to sum all numbers from 0 up to a number read from the terminal. This exercise is part of the second chapter, in which students have only seen `for` constructs. Surprisingly, this exercise only took 19 submissions for 7 students. It is important to note however that just two students have sent 11 submissions. We discuss two model like submissions in Listings 5.4 and 5.5, and a submission with an unintended solution in Listing 5.6.

```
x = int(input("x: "))
summation = x
for i in range(summation):
    summation = summation + i + 1
print(summation)
```

Listing 5.4: A submission with a `+1` addition fix.

```
x = int(input("x: "))
summation = 0
for i in range(x + 1):
    summation = summation + i
print(summation)
    " euro!", sep="")
```

Listing 5.5: A submission with a `+1` range fix.

```
x = int(input("x: "))
print(x * (1 + x)/2)
```

Listing 5.6: A submission completely ignoring `for`.

Most students use a loop as is intended by the exercise, however, it appears most students are executing a zero addition due to the chosen range. This is demonstrated in Listings 5.4 and 5.5, where `i` will be `0` for the first iteration, because the `range`'s start defaults to `0`. Usually, students started with a `range` of the correct length (`x`), but off by one value. In their consecutive correcting submission, they usually extended the range like Listing 5.5 or simply added '`+1`' in the assignment statement like in Listing 5.4, instead of shifting it. This is why zero additions are present in so many submissions. Only 3 of the 7 passed solutions match `range(1, x + 1)`. One of the students rightfully reasoned there is a more clever and efficient approach to calculate the summation than iteratively summing numbers. That is, the formula for triangular numbers used in Listing 5.6. By doing so, however, the student is not practicing `for` loops.

5.3 Reversing a string

In a third exercise, students are asked to reverse a string read from the terminal. This exercise was part of the third chapter in which students were taught while loops. For this exercise, 27 submissions were made in total by 8 students. One student submitted three different solutions. We discuss three submissions in Listings 5.7, 5.8 and 5.9 that represent common strategies and one that uses unintended solutions in Listing 5.10.

In Listings 5.7 and 5.8, a while loop is used, even though the number of iterations is known beforehand. This could be an indication that the student has not fully grasped the difference between definite iterations (`for`) and indefinite iterations (`while`) yet. In Listing 5.10, the student has experimented with unintended functions. As described in the previous section, simply checking for a range at runtime is an easy way to tell whether a for loop was used and whether its range is as expected.

```
word = input("word: ")
length = len(word) - 1
while length != -1:
    print(word[length], end="")
    length -= 1
# forgot: print("")
```

Listing 5.7: An incomplete submission printing character by character using `while` by decrementation.

```
word = str(input("word: "))
reversed_word = ""
length = len(word)
i = 0
while ((length - i) > 0):
    i += 1
    reversed_word += word[length - i]
print(reversed_word)
```

Listing 5.8: A submission printing the result using `while` by incrementation.

```
word = input("word: ")
length = len(word)
for x in range(length):
    print(word[length - x - 1], end="")
print("")
```

Listing 5.9: A submission printing character by character using `for` by incrementation.

```
string = input("word: ")
print(string[::-1])
```

Listing 5.10: A submission using an unintended solution.

In Listings 5.7 and 5.9, one character is printed at a time, whereas in Listing 5.8, the character is appended to a string and afterward the result is printed. The end result is the same, but it can be opinionated that one approach is better to promote than the other. The former, for example, requires knowledge on `print`'s `end=...` parameter and a second call to add a newline to the result that is often missed. For six out of seven students, the final, passing submission for their solution was preceded by one that had a missing print statement to print the last newline. The latter separates the output from the logic, but requires an additional variable, making the code more verbose.

5.4 Even numbers

In the first exercise of a chapter introducing functions, students are asked to define a simple, self-explanatory function `is_prime` with parameter `x`. In total, 8 students made 26 submissions. We provide three submissions in Listings 5.11, 5.12 and 5.13.

```
def is_even(x):
    if x % 2 == 0:
        return True
    elif x % 2 != 0:
        return False
```

Listing 5.11: An example of a submission proxying a boolean variable.

```
x = int(input("x: "))
even = True

def is_even(x):
    even = True
    if x % 2 == 1:
        even = False
    return even

print(is_even(x))
```

Listing 5.12: An example of a submission manipulating an additional variable.

```
def is_even(x):
    print(x % 2 == 0, end="")
```

Listing 5.13: An example of a submission showing that the student does not know what to do with the result. `return`.

Students usually had brief difficulty adapting to the new type of correctness assessment that does not expect the result to be printed, but instead a function to be defined that returns the correct result. The first attempts like Listings 5.12 and 5.13 usually contained print statements. A few submissions contain checks for truthiness to either return `True` or `False`, or sometimes store (and manipulate) the result right before returning, which is the case in Listings 5.11 and 5.12. This could indicate an incomplete understanding of either if statements (when the condition is not regarded as a boolean value) or return statements (when it may be assumed that `return` is always only followed by a calculated value – a variable or a constant).

It should be noted though that 6 out of 8 students eventually ended with a passing submission that matched the model solution (i.e., a function with '`return x % 2 == 0`' as its body). Sometimes, they had already submitted code that passed all tests, but they further experimented with the small piece of code until they discovered the model solution. This may indicate that for these small exercises, it is not as necessary to nudge the student to come up with solutions closer to the model.

5.5 Prime numbers

In a later exercise on functions, students were asked to implement the self-explanatory functions `is_prime`, `all_primes_up_to` and `factorize`. For this exercise 67 submissions were made by 8 students. One of the students made 33 submissions and another made 12, and the next made 8, so only 14 submissions were made by the remaining 5 students. We will focus on implementations of `factorize`, for which only 31 of them remain. Of these, only 12 are made by the same student, while the rest are more evenly distributed among the other 7 students. We discuss 5 distinct submissions in Listings 5.14, 5.15, 5.16, 5.17 and 5.18 passing all unit tests.

The majority (5 students) solved the exercise using a for loop approach, very similar to Listing 5.14. In this small sample, it appears to take more attempts to solve the exercise (i.e., pass all unit tests) for students that do so with a while loop than students that do so with a for loop. This makes sense, since the approach to iterate over all prime numbers may feel more intuitive at first. However, since these types of algorithms start from the smallest numbers, it is unnecessary to call `is_prime`, since prime numbers will precede their multiples and make it therefore impossible to naturally divide by a multiple afterward. This property was often overlooked, which can also be seen in the while loop approach of Listing 5.15 with the call to `is_prime`. In Listings 5.16 and 5.17, students figured out that evaluating the remainder is enough to decide whether `x` may be divided by `number`, but they still missed that all numbers below `number` would not be able to divide it anymore, and hence reset `number` to `2`. By removing this reset in Listing 5.16, a few unnecessary re-iterations could be prevented. This important in a course where the focus is on algorithmic thinking, in contrast to writing functions that happen to produce the correct results.

Aside to the practice on loops, it appears now and then a few mistakes against other patterns are present that are trained in previous chapters. For example, in Listing 5.17, the conditions in if statements do not take into account previous tests, the while condition is moved to an if statement for which a variable is kept, the same boolean variable is compared to `False`, etc. Yet, it too passed all unit tests. The same student submitted the code in Listing 5.18, in which a few of these remarks persist. This shows us that students are continuing chapters, while previous concepts are not mastered yet, making it harder for them to continue the current exercise.

```
def factorize(x):
    for number in range(2, x):
        if is_prime(number):
            while (x % number) == 0:
                x = x / number
                print(number)
```

Listing 5.14: An example of a typical submission using a `for-while` loop.

```
def factorize(x):
    number = 2
    while not is_prime(x) \
            and number < x:
        if is_prime(number) \
                and x % number == 0:
            x = int(x/number)
            print(number)
        else:
            number += 1
    print(x)
```

Listing 5.15: An example of a typical submission using a `while` loop.

```
def factorize(x):
   number = 2
   while x != 1:
       if x % number == 0:
           print(number)
           x = x // number
           number = 2
       else:
           number += 1
```

Listing 5.16: An example of a typical submission using a `while` loop.

```
def factorize(x):
   number = 2
   stop = False
   if is_prime(x) == True:
       print(x)
       stop == True
   while stop == False:
       if x % number == 0:
           x = (x // number)
           print(number)
           number = 2
       if x - 1 == number:
           print(x)
           stop = True
       if x % number != 0:
           number += 1
       if x == 1:
           stop = True
```

Listing 5.17: An example of a typical submission.

```
def factorize(x):
    if is_prime(x):
        print(x)
    else:
        while x > 1:
            i = 2
            stop = False
            while stop == False:
                if x % i == 0:
                    x /= i
                    print(i)
                    stop = True
                else:
                    i += 1
```

Listing 5.18: An example of a typical submission.

Chapter 6

PERSSISTANT: Concepts

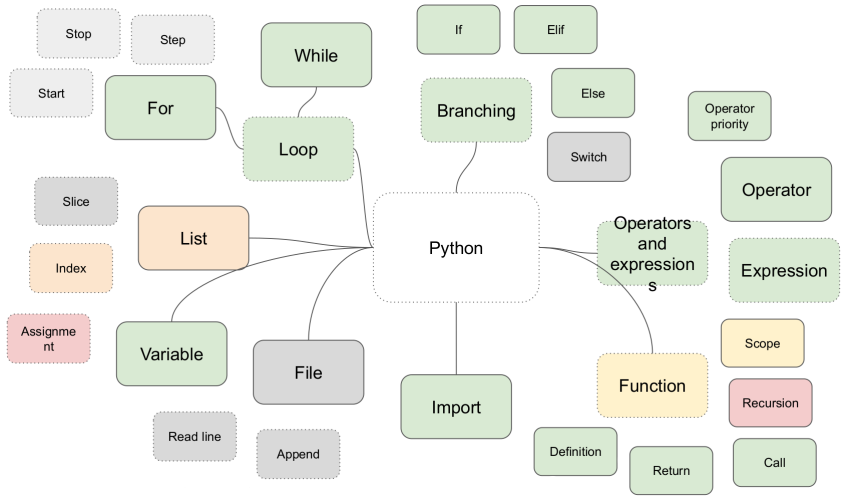
In this chapter, we describe the PERSSISTANT conceptually. The system will 1) monitor the progression students make each time a student submits code in a learning environment, 2) display this estimated progress, and finally, 3) use this estimated progress to provide the students recommendations, when an intervention is made. Unlike traditional ITS methods described in Section 2.3, PERSSISTANT is fully instructor driven.

In Section 6.1, we clarify how the knowledge domain is modeled as skills in PERSSISTANT. In Section 6.2, we overview the definition of progress. In Section 6.3, we describe how the application of a certain skill can be tested through several patterns. In Section 6.4, we describe how these pattern tests can be structured to grade a submission to an exercise. In Section 6.5, we describe how the assessed progress of a submission can be aggregated to provide higher level progress overviews, such as chapters, course, and skills. Finally, in Section 6.6 we describe how the current approach to structure-based skill assessment can be used to provide proactive feedback before a student makes an exercise, as well as during an exercise.

6.1 Modeling the knowledge domain

Inspired by the hierarchical knowledge-tree for the C programming language in the framework of *Knowledge Structure Tree based Exercise Recommendation (KSTER)* by Zheng et al. [Zhe+22]. We will similarly think of the Python course as a means to 'fill' the Python skill tree, where each skill and sub-skill has a mastery state. Figure 6.1a shows an example of such a tree, where every skill is related directly (or indirectly through children) to a Python construct. Additionally, the mastery states of a hypothetical student are represented using color, laid out in Figure 6.1b, but the system certainty is ignored. Note that this tree is hypothetical, as, unlike Zheng et al. [Zhe+22], we did not include experts in its design process.

Ideally, the skill assessment works with hierarchical structures, so students are more accurately assessed and the feedback can be more detailed, but this is not required. In the implementation described in 7, for instance, this is omitted. Furthermore, this knowledge tree alone does not bring any value if its skills are unrelated to the exercises they are part of (similar to KT described in Section 2.3.1). This mapping can be used to communicate that the student has not yet achieved the required mastery of certain skills (constructs) for an exercise upon initiation. This mapping is useful for making recommendations to other exercises in an attempt to make the learning more optimal (which is exactly what KSTER is for [Zhe+22]).



(a) The skill tree.

(b) The mastery states.

Figure 6.1: A synthetic student skill tree with mastery states.

6.2 Progress

In a traditional unit testing ITS, student progress can be defined in terms of whether an exercise has passed the tests (binary progress state). Using this state, progress per chapter can be defined in terms of the number of exercises passing, so that each exercise contributes equally to the total chapter progress. Likewise, exercises can be used directly to measure course level progress, or they can be grouped per chapter so that each chapter equally contributes to the total course progress.

As we previously noted in Chapter 5, students do not always use the intended constructs, or go through submission by trial-and-error. Hence, progress should not be defined solely on the number of tests passed. Similar to the traditional approach, skill-based progress can be defined at the aggregating levels, chapter-to-course, exercise-to-chapter. Unlike traditional progress, skill-based progress on the exercise level is not binary, but consists of an aggregation of all the skills that are present in that exercise. By this metric alone, students can make more informed decisions on what exercise to (re)do. Aggregating the skill performances of each exercise at the course level per skill further enables the provision of a detailed overview of each skill's progression.

Before we describe how we measure progress per exercise in detail, we briefly introduce our progress bar representation. Figure 6.2 shows one example of 50% progress made given a certain level of certainty in the system. Figure 6.2a represents this using a progress bar, and a binned, colored distribution directly underneath to indicate the most certain progression. This is the default representation we use throughout the remainder of the thesis. Figure 6.2b shows an example certainty distribution that could be mapped to the color bins. This detailed information is not always available, however. For example, BKT described in Section 2.3.1 uses one value to indicate mastery, not a full distribution. In our methods discussed in Section 6.3, we also do not use a distribution. Practically, only the binned certainty levels (blue) and a specific estimate (green) are needed, which is modelled in Figure 6.2c. Finally, this progress bar may be labeled, as shown in Figure 6.2d, in which case it refers to a certain skill.

(a) The default progress representation. **(b)** An example max-resolution distribution. **(c)** The minimal required distribution information. **(d)** The default progress associated to a skill.

Figure 6.2: A formal representation of 50% progression with system certainty.

6.3 Pattern-based skill assessment

In order to provide a comprehensive explanation of structure-based skill assessment at the level of an exercise, we first need to explain how the specific application of a skill (construct) is evaluated. For a specific application a_j^s of a skill $s^g \in S^g$ in course g, PERSSISTANT will evaluate each pattern test $p_i \in P_j$ associated to a_j^s, within a limited part of the code. These pattern-skill associations (tests) are created by the instructor in a declarative manner — similar to the static analysis provided by PMD[EKR17]. Each test p_i will result in a binary outcome o_i that is either passing or failing, and provides a bias value $b_i \in \mathbb{Z}$ associated with o_i. These bias values are then summed together in r_j. If $r_j > 0$, then a_j^s suggests mastery may be acquired, if $r_j <= 0$, it suggests mastery may be absent.

Two types of patterns are suggested: 1) those heavily inspired by XPath to check the relation between two nodes in a structure and 2) those that simply count tokens, which is useful to further confirm the bias, increasing the system certainty for the skill application test. By default, each pattern test provided must specify the biases `failcert` and `passcert` that will be interpreted as certainties. Furthermore, they allow specifying an expected range of occurrences `occrange` to pass, and fail otherwise.

The XPath-like patterns are summarized in Table 6.1, containing definitions for a few typical XPath selectors: `parent`, `ancestor`, `preceding-sibling`, `following-sibling`, `precedes` and `follows`. The patterns based on these selectors similarly work on two node *types* `a` and `b` – not specific instances. Also similar to XPath, one of these can be the `root` node selector – which *is* an instanced node. Additionally, the expected occurrence type `occtype` is required to specify whether `every` two nodes with the specified types match, or only `some` of them. The specification '`never`' can be achieved by assigning the `occrange` to $[0, 0]$.

name	parameters
`parent`, `ancestor`, `preceding-sibling`, `following-sibling`, `precedes`, `follows`	`ast`, `a`, `b`, `occtype`, `occrange`, `passcert`, `failcert`

Table 6.1: XPath-like patterns.

Another set of patterns are those that globally count (within the part of the code provided) the occurrence of certain tokens or condition tokens, summarized in Table 6.2. Pattern `count` will verify that a single `token` *type* occurs within `occrange`. Pattern `condcount` will similarly verify the occurrence of a condition token, but is constrained to be part of either an if or a while construction's condition. Finally, pattern `calls` is a specific type of `count` that additionally verifies if the token found by name (string) is actually a callable function.

name	parameters
`count`	`ast`, `token`, `occrange`, `passcert`, `failcert`
`condcount`	`ast`, `token`, `stmttoken`, `occrange`, `passcert`, `failcert`
`calls`	`ast`, `funcname`, `occrange`, `passcert`, `failcert`

Table 6.2: Counter patterns.

6.4 Structure-based skill assessment

Given that Section 6.3 explained how to evaluate a skill application, we can now explain how these influence the structure-based assessment of skills in a submission d^e for exercise e. For this, we look back at the exercise of calculating a summation from Section 5.2. Clearly, the model solution contains only one for loop. Any student deviating from that (i.e., by using multiple for-loops or none at all) is probably doing something "wrong" that does not lead to the intended learning gain. Through experience, a human instructor may quickly check whether a submission conforms to certain criteria, such as: 1) 'has exactly one for loop', 2) 'has no unnecessary pre-/post-iterations', 3) 'has expected range, so it does everything inside the loop', 4) 'directly converts the input', 5) 'uses only constructs seen up to this lesson where the exercise is provided', 6) 'uses only the necessary construct that a model solution would need', ... These checks necessitate that instructors mentally map high-level constraints to lower-level code constructs that may indicate a violation (or, conversely, a satisfaction) of that constraint.

Using a naive e-scoped metric $\hat{q}^s$ on a set $\hat{A}^{e,s}$ may however be insufficient for the automation of this constraint-like evaluation of d^e. (The formula below is an example that calculates the average skill progress based on each $\hat{a}_j^{e,s}$.) First, exercises may be composed of multiple subproblems (i.e., multiple functions that should be defined, such as in Section 5.5). The larger the exercise, the easier it is to count the wrong token or token pair through pattern tests. Second, given that multiple valid solutions may exist (e.g., Section 5.3), this basic approach may become unfeasible. Either the tests are set up to only expect one model solution, ignoring other valid (model) solutions, which will results in inaccurate assessed progress for the latter, or the tests are solution agnostic, sacrificing accuracy and certainty.

$$\hat{q}^s(d^e) = \frac{\sum j = 1|\hat{A}^{e,s}| f_{progress}(\hat{a}_j^s)}{|\hat{A}^{e,s}|}$$

Therefore, PERSSISTANT can assess d^e at a set of different structures T (parts of its AST), limiting the scope of patterns if desired, thereby preventing other structures from affecting the bias. A clear use-case of multiple structures is found with exercises that ask to implement multiple functions. Another use-case where structures may be useful is when the template already suggests (with comments) where certain parts of the code are expected, thereby limiting the solution space, or a dividing element (such as a call to `input`) is expected. To address the second problem of large solution spaces, a set of model solutions M_l can be defined per structure $t_l \in T$. A model solution $m_k \in M_l$ is associated with its own set of application tests A_k. Figure 6.3 provides a high-level overview of all the tests PERSSISTANT executes for d^e, in which 'P' represents $p_i \in P$, 'Skill' represents $a_j^s \in A$, 'Solution' represents $m_k \in M$, and 'Structure' represents $t_l \in T$.

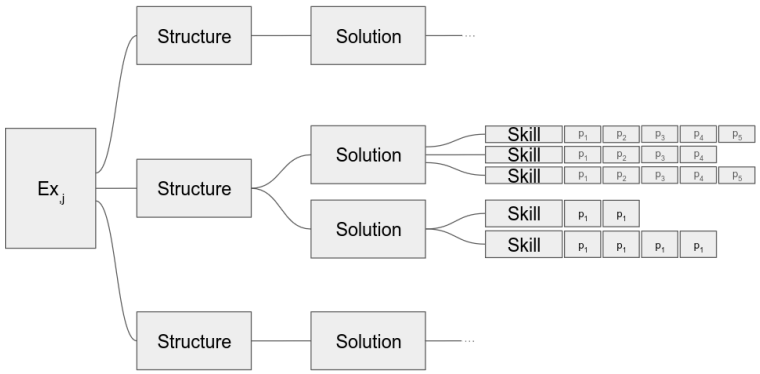


Figure 6.3: A high-level overview of the required data for structure-based assessment of an exercise.

Next, PERSSISTANT determines which $m_k \in M_l$ to pick for t_l by weights w_j associated to each $a_j^s \in A_k$ to influence a ranking function f_{rank} defined as:

$$f_{rank}(m) = \sum_{j=1}^{|A|} r_j \cdot w_j$$

The model m_l closest to the student's structure (and associated to A_l) is then found using argmax as shown below. Theoretically, the result of argmax is a set itself ($M_{l,max}$), so it does not make sense to assign it to m_l. Because it is possible that $f_{rank}(m_b) = f_{rank}(m_c)$ for two models $m_b, m_c \in M_l$, the cardinality may be higher than 1.

$$\begin{aligned} m_l &\in M_{l,max} \\ &= \underset{m_k \in M_l}{\operatorname{argmax}} f_{rank}(m_k) \\ &= \{m_k \in M_l : f_{rank}(m_i) \leq f_{rank}(m_k) \forall m_i \in M_l\}. \end{aligned}$$

Finally, the progress on an exercise skill is estimated using metric q^s defined similarly to $\hat{q}^s$ below, together with an analogous certainty metric c^s. Note that the ranking of the models influences whether $a_j \in A$ (with $0 \leq j < |A|$) is included in the calculation of both the progress metric and the certainty metric. Furthermore, it does not involve weights, as these are solely meant to influence the ranking of a model, not directly the progress.

$$q^s(d^e) = \frac{\sum_{l=1}^{|T|} \sum_{j=1}^{|A_l^s|} f_{progress}(a_j^s)}{|\{a_j : \exists t_l \in T.a_j \in A_l\}|}$$

$$c^s(d^e) = \frac{\sum_{l=1}^{|T|} \sum_{j=1}^{|A_l^s|} r_j}{|\{a_j : \exists t_l \in T.a_j \in A_l\}|}$$

And we define $f_{progress}$ as follows:

$$f_{\text{progress}}(a_j) = \begin{cases} 1, & \text{if } r_j > 0 \\ 0, & \text{if } r_j \leq 0 \end{cases}$$

Finally, submission d^e's total progress q^d and total certainty c^d aggregate this together by weighting each exercise skill s^e by its associated weight v^s.

$$q^d(d) = \sum_{s^e \in S^e} q^s(d^e) \cdot v^{s^e}$$

$$c^d(d) = \sum_{s^e \in S^e} c^s(d^e) \cdot v^{s^e}$$

The progress q^m and certainty c^m of an individual solution m_k can be calculated using the formulae below. These are used for progress visualization in Figure 6.4), but are not utilized in PERSSISTANT. Notice that in these formulae, exercise skills are not binned together. This is because a structure is not guaranteed to use all exercise skills.

$$q^m(m_k) = \sum_{a_j \in A_k} f_{progress}(a_j)$$

$$c^m(m_k) = \sum_{a_j \in A_k} r_j$$

Figure 6.4 provides an overview of the assessment of a synthetic submission, similar to Figure 6.3. The bias scores of each pattern 'P' are represented by dark red for strong negative bias, dark green for strong positive bias, and bright/white colors for small biases. As we noted before, the progress for each solution (and consequently its structure) does not affect the submission progress.

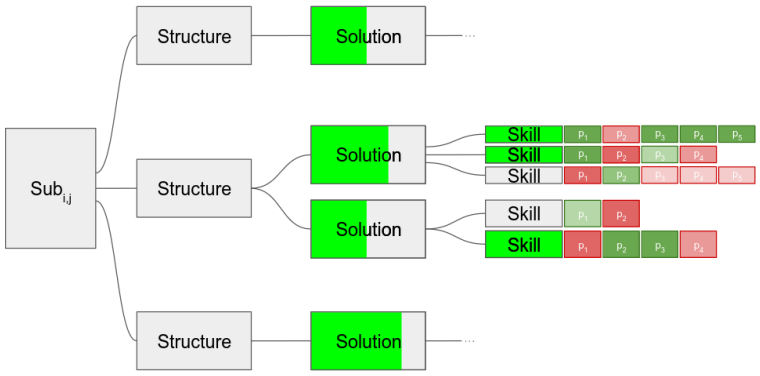


Figure 6.4: An overview of the assessment of a synthetic submission

6.5 Aggregating progress

As discussed briefly in Section 6.2, progress metrics can be aggregated together to provide clearer insights. First, to aggregate exercise e_y's progress q^e and certainty c^e from submission $d_x \in D_y$, the best submission can be selected using $argmax_d q(d)$. Next, as each exercise $e_y \in E_z$ is part of a chapter $ch_z \in C_g$, the chapter's progress q^c and certainty c^{ch} can be calculated with the formulae below.

$$q^c = \frac{\sum e_y \in E_z q^{e_y}}{|E_z|}$$

$$c^c = \frac{\sum e_y \in E_z c^{e_y}}{|E_z|}$$

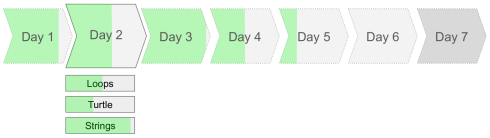


Figure 6.5: An example overview of the progress made on each chapter ('Day').

Finally, each chapter $ch_z \in C_g$ can similarly be aggregated to course g's progress q^g and certainty c^g.

$$q^c = \frac{\sum ch_z \in C_g q^{ch_z}}{|C_g|}$$

$$c^c = \frac{\sum ch_z \in C_g c^{ch_z}}{|C_g|}$$

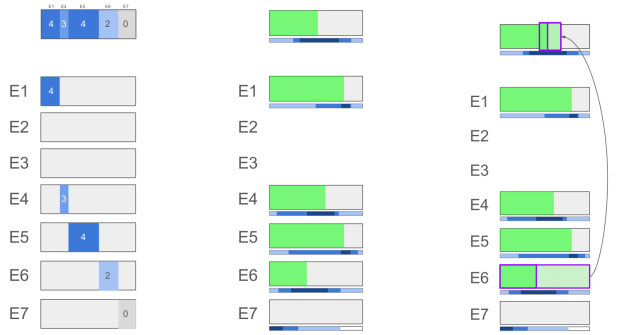
In addition to providing traditional-like aggregated progress overviews, the fine-grained skill assessment allows the dependency visualization between exercises and a skill at the chapter level (like in Figure 6.5) or at the course level. Ideally, an exercise $e_h^s \in E_g$ (with $0 \leq h < |E_g|$

if at course level, or $0 \leq h < |E_z|$ at chapter level) is associated to a weight value u_h, because exercises may require different levels of skill mastery.

$$q^{g|z} = \frac{\sum e_{g|z} \in E_{g|z} q^{e_{g|z}}}{|E_{g|z}|}$$

$$c^{g|z} = \frac{\sum e_{g|z} \in E_{g|z} c^{e_{g|z}}}{|E_{g|z}|}$$

Figure 6.6 provides an example overview of the contribution of each individual exercise to the chapter skill. Figure 6.6b shows an exercise's relative contribution to the course, whereas Figures 6.6a and 6.6c show the same contribution through progress bars. Figure 6.6c shows how a student can be informed of the expected contribution of an exercise.

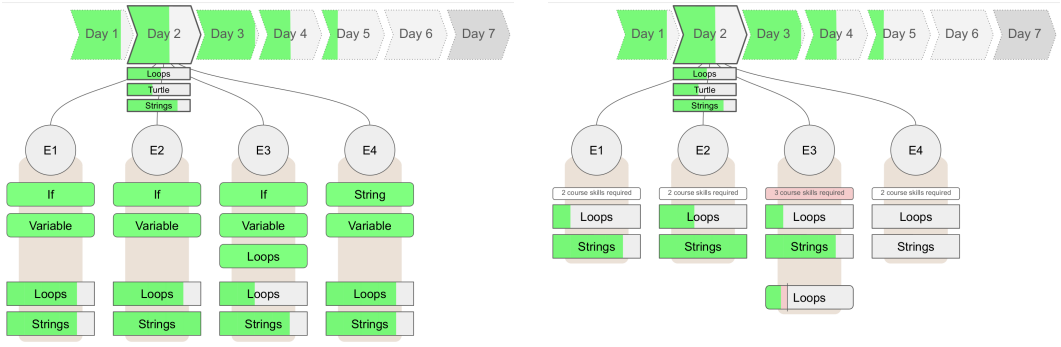


(a) Enter Caption **(b)** Enter Caption **(c)** Enter Caption

Figure 6.6: Enter Caption

6.6 Personalized, Proactive feedback

We conclude with a discussion on how these skill progressions can be used in two ways. First, they can be used to warn a student who is skipping important exercises or progressing through them poorly, whenever the student is about to start an exercise that will lead to inefficient trial-and-error behavior and perhaps affect their motivation negatively. This scenario is depicted in Figure 6.7, where one student has sufficiently progressed `E1` and `E2`, and made `E3` and `E4`. The other student, however, has low estimated progress on `Loops` (the topic of the chapter '`Day 2`'), and therefore is suggested to continue `E1` and `E2`, as their skill is not sufficiently mastered yet.



(a) Caption **(b)** Caption

Figure 6.7: Caption

While the student is solving an exercise, PERSSISTANT can also interrupt whenever the student takes longer than expected, as an indication of the student getting stuck as discussed in Section 3.2. Further building upon this concept, the student can be recommended exercises that train skills in which they seem to be lacking. We do not provide metrics and instead further discuss this approach in Chapter 8.

In the next chapter, we discuss PERSSISTANT's implementation.

Chapter 7

PERSSISTANT: Implementation

In the previous chapters, we have combined the concepts from Chapters 3 and 6 to create personalized learning trajectories in programming. In this chapter, we will describe PERSSISTANT's implementation, with a focus on the student's workflow. As mentioned before, intermediate exercises are not part of this prototype's DPP. In the next sections, we will detail the prototype's components in back-end and front-end.

7.1 Back-end

The back-end consists of four components, as depicted in Figure 7.1. An *HTTP server* hosts static *content* for the student and stores the test results created by *correctness assessment* and *structure-based skill assessment* upon receiving a submission. The first two components will be detailed in Section 7.1.1, the third in Section 7.1.2 and the fourth in Section 7.1.3.

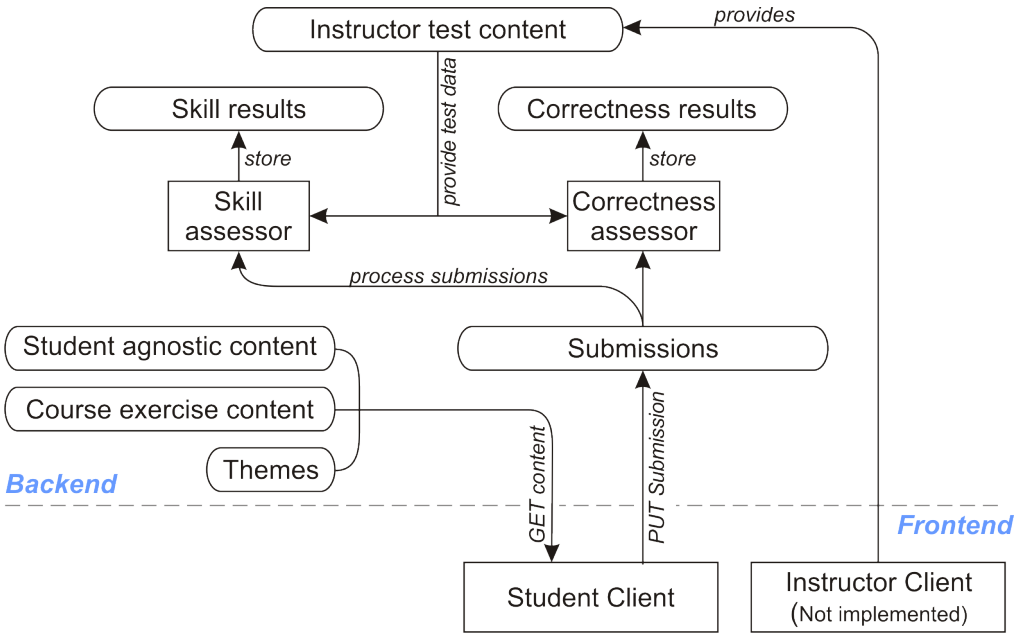


Figure 7.1: An overview of PERSSISTANT's backend, where the HTTP-server represent the dashed line between frontend and backend.

7.1.1 Serving static and dynamic content

To provide **static, representative course content** for the prototype, we have extracted exercise descriptions, working solutions and exercise templates from HTML dumps of exercise webpages. These pages were originally hosted for an introductory course on programming (2019) at our university, 'Programming and Algorithmic Thinking', which is the same course for which submissions were analyzed from Chapter 5. With some clever HTML-to-Markdown conversions, we turned these into readable exercise descriptions.

To effectively track progress throughout the course, it is essential to store **student submissions** (or some summary thereof) for future retrieval. For each exercise, PERSSISTANT saves all submissions in a JSON file (along with assessed properties, discussed later) inside a directory. Due to time constraints, the assessed progress of **skills** is not tracked globally in the course. Moreover, as we focused on tracking progress of one student, the saved data is student-agnostic.

The extracted descriptions and templates, along with submission test results and various programming themes, are provided through a custom **HTTP server** built using Python's HTTP modules. When responding to an uploaded submission, the server will provide a submission ID, so the student client can track the status by polling the server and fetch the results when it is ready. To not overload the server, the delay between each client's polling request will generally increase, from hundreds of milliseconds to several seconds at the end, until the client will give up – which is seldom the case. While many simple HTTP servers exist for serving static content from a file directory through GET requests, most do not support PUT requests (necessary to store new information). To cope with that limitation, and because of the need of an API end point for correctness assessment and structure-based assessment, we developed a custom server for PERSSISTANT.

7.1.2 Correctness assessment

In PERSSISTANT, two types of unit tests are supported to support assessment of all submissions from the course content used. The first type of test treats the submission as a script to be executed (without command line arguments, though trivially implementable), requiring instructors to supply command line input strings of the unit test(s) and an output string representing the expected result(s). The second type involves function calls, where instructors provide the arguments and expected return values. In the case of the latter type, the command line I/O can still be compared if provided (otherwise, it is disregarded). Each test will then be stored in a JSON file per exercise like Listings 7.1 and 7.2, ideally by an **instructor client**.

```
[
  { "type": "script",
    "tests": [
      { "inputs": [4, 2, 5, 1, 4],
        "expectedOutput":
        "You have 1.23 euro!\n"
      },
      { "inputs": [0, 0, 2, 1, 3],
        "expectedOutput":
        "You have 0.80 euro!\n"
      },
      { "Omitting": "Other tests" }
    ]
  }
]
```

Listing 7.1: Script unit test data for the exercise in Section 5.1.

```
[
  { "type": "function",
    "callable": "is_even",
    "tests": [
      { "arguments": [1],
        "expectedResult": false
      },
      { "arguments": [-4],
        "expectedResult": true
      },
      { "Omitting": "Other tests" }
    ]
  }
]
```

Listing 7.2: Call unit test data for the exercise in Section 5.4.

For execution of unit tests upon receiving a student submission, PERSSISTANT utilizes the `astinterp` Python interpreter module, written by Paul Sokolovsky, which is a single file and therefore easy to integrate and extend. As the module is based on Python's `ast` module itself, it traverses nodes through the `ast` visitor interface, which has the advantage of easily overriding certain visit calls that would interpret parts of the code. This is especially useful to intercept function calls such as `print`, to capture the output for later comparison, and `input` to provide custom input strings, as well as execute any function calls by name, which allows further experimentation discussed in Section 8.3. For a more technical description of unit testing using python, the work of Nguyen and Hoang [NH24] can be consulted, as they similarly fetch functions and execute parts of the AST.

However, using an interpreter in an interpreter comes at the cost of performance, especially when certain tests require a high number of operations and/or the submission itself has an inefficient implementation. For example, the submission in Listing 5.14 from Section 5.5 took 62.25 seconds to factorize 3003 by *astinterp*'s implementation, whereas the same code executed by the standard Python interpreter took 0.04 seconds. For less "extreme" tests, the execution time is reasonably low. To mitigate submissions running excessively long, instructors can add a maximum expected time, after which execution is interrupted and the test will fail. They can execute their model solutions with PERSSISTANT for each unit test to find an upper bound (which is by default 2s). This small limitation would make our non-optimized approach less suitable for production. Other Python interpreters, libraries, or command line based solutions exist that may perform faster, but they are often not designed to be used as a library or do not provide fine-grained access to parts of the interpretation.

7.1.3 Structure-based skill assessment

Finally, the structure-based skill assessment is executed when the unit tests have been completed. Just like the unit tests, their execution depends on content created by instructors. The declarative approach to detail all skill tests and their patterns from Figure 6.3 is captured in a JSON file with a compatible scheme. Listing 7.3 provides our assessment data for the exercise from Section 5.5.

Each pattern test provided will be executed as described in Section 6.3 to predict whether a skill application indicates mastery or not. For this, PERSSISTANT also uses Python's `ast` module to implement XPath-like tree selection to find all matches. Finally, the best matching model solution for each structure test are filtered as described in Section 6.4, and a submission score is binned from these as described in the same section. This score is however not stored directly for an exercise (but indirectly for each submission) due to time constraints. Therefore, PERSSISTANT does not execute the chapter, course, or skill level aggregations as described in Section 6.5.

7.2 Front-end

The student's front-end consists of three pages, namely the course overview/progress page and the exercise environment page, which are both heavily inspired by the webpages hosted by Dodona, from which the content was extracted. Furthermore, a skill progress page gives a basic overview of the course's skills and the progress made on them so far. In Section 7.2.1, we will highlight the features basic features, followed by the way in which structure-based skill assessment results are shown in Section 7.2.2. In Section 7.2.3, we will show the progress overviews, and finally in Section 7.2.4, we show or minimal implementation of proactive feedback.

The interface is developed using the web framework React, chosen due to our prior experience with it. We utilized the extensive array of customizable components offered by MUI to compose the interface on top of our React application. For HTTP requests to our server, we used Axios.

```
[
  { "root": "factorize",
    "solutions": [
      { "id": "1",
        "skills": [
          { "name": "loop", "weight": 0.5,
            "description": "Uses only one while loop", "patterns": [
              { "name": "token-count", "token": "[while]", "allowedRange":
                [1, 1], "passCertainty": 4, "failCertainty": -8
              },
              { "name": "token-count", "token": "[for]", "allowedRange":
                [0, 0], "passCertainty": 4, "failCertainty": -9
              }
            ]
          },
          { "Function tests": "Left out" },
          { "name": "loop", "weight": 0.3,
            "description": "No pre-branching", "patterns": [
              { "name": "ancestor", "occurrence": "every",
                "tokenA": "[:root:]", "tokenB": "[while]",
                "passCertainty": 6, "failCertainty": -1
              },
              { "name": "parent", "occurrence": "every",
                "tokenA": "[:root:]", "tokenB": "[while]",
                "passCertainty": 2, "failCertainty": -1
              },
              { "name": "parent", "occurrence": "every",
                "tokenA": "[if]", "tokenB": "[while]",
                "passCertainty": -3, "failCertainty": 0
              },
              { "name": "preceding-sibling", "occurrence": "some",
                "tokenA": "[if]", "tokenB": "[while]",
                "passCertainty": -3, "failCertainty": 0
              },
              { "name": "token-count", "token": "[if]", "allowedRange": [2, 2],
                "passCertainty": -3, "failCertainty": 0
              },
              { "name": "token-count", "token": "[if]", "allowedRange": [1, 1],
                "passCertainty": 1, "failCertainty": 0
              }
            ]
          },
          { "If test left out": "Left out" }
        ]
      }
    ]
  }
]
```

Listing 7.3: An example definition of one solution (modeling Listing 5.15) for the `factorize` structure.

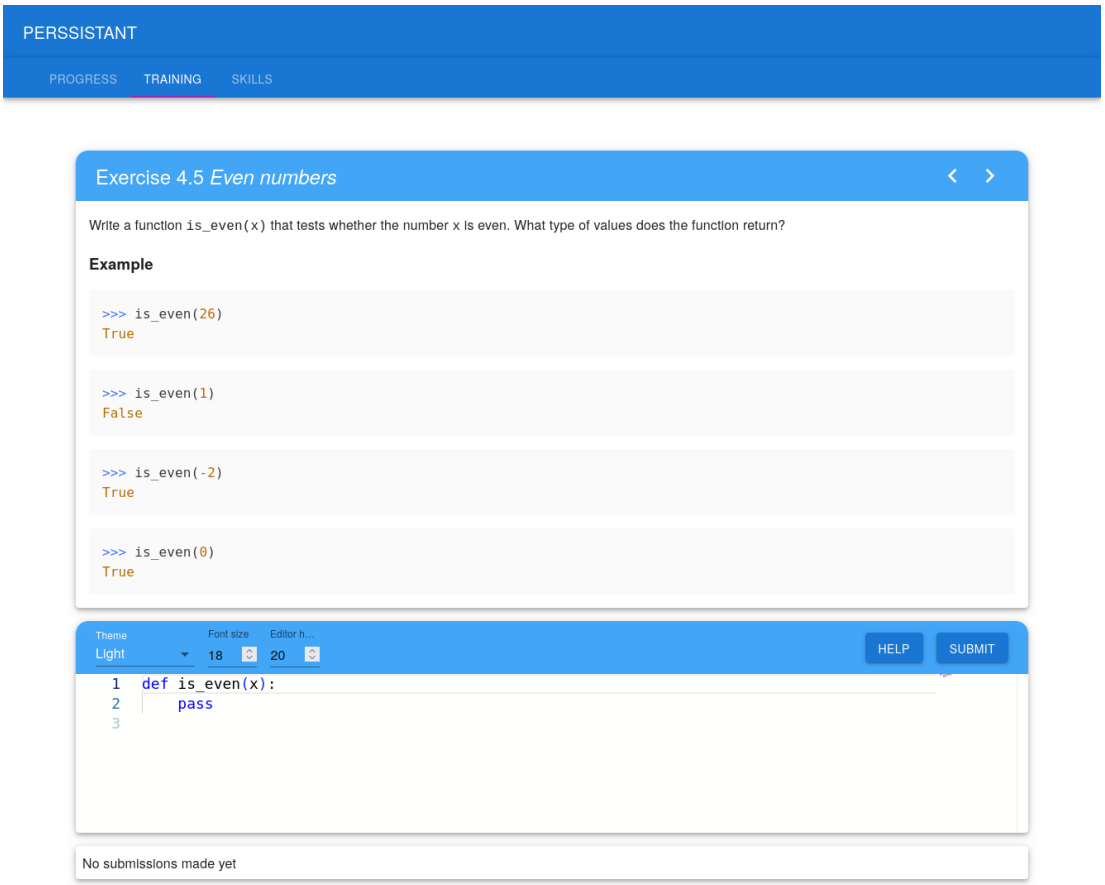


Figure 7.2: The initial view of the exercise from Section 5.5, with a template loaded in the IDE and an empty list of submissions.

7.2.1 Basic exercise IDE features

First, the prototype's front-end provides the basic features for solving an exercise that a programming ITS would provide in a separate "Training" tab, as shown in Figure 7.2. In the first box, the exercise's basic information is displayed to the student, such as the chapter number and exercise title, as well as the exercise's description. For the latter, PERSSISTANT uses a Markdown component with math and code-highlighting plugins to improve the readability of the extracted exercises. The left and right arrows at the top right are navigation controls that load the previous or next exercise.

Next, students write down their solutions within the interface's editor starting from the template solution which is automatically loaded for exercises without submissions. PERSSISTANT uses a port of the feature-rich Monaco editor, that encompasses the same shortcuts as well as syntax highlighting out of the box. As a small bonus accessibility feature, students can also choose among a wide range of themes, set the font sizes, and adjust the editor height relative to the browser window height, so they are not discouraged from using the provided editor.

By clicking "compile", code from the editor will be sent to the server for assessment, after which a new submission and its results will be added to the bottom box. This box is a list of expandable submissions, each allowing the student to inspect the submitted code, the correctness test results, the skill test results, and some meta information compatible with Judge's meta data (creation data, assessment data, server's exit code, etc.). Figure 7.3 shows three submissions, of which the top (last) submission has all these four sections collapsed.

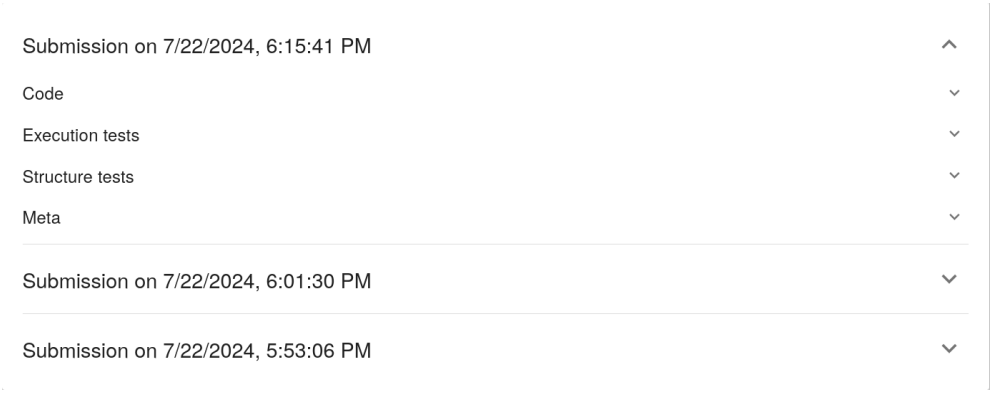


Figure 7.3: The an exercise's list of submissions, one of which is expanded, showing Code, Execution tests, Structure tests and Meta sections.

A few smart guesses are made at what section of a submission's results should be expanded after it is assessed. Figure 7.4 shows the section for correctness tests fully expanded for Listing 5.3 from Section 5.1. When the code produces the wrong output, it will be highlighted in red and shown next to the expected output for side-by-side comparison. Otherwise, only the output is shown in green. When a runtime error is caught, the error message for that test will be shown instead, as shown in Figure 7.5. Whenever a syntax error is found, the code cannot be interpreted and the meta section (collapsed by default) will be expanded containing the error message, as shown in Figure 7.6. When all correctness tests pass, its section will collapse, expanding the section of structure-based skill test results.

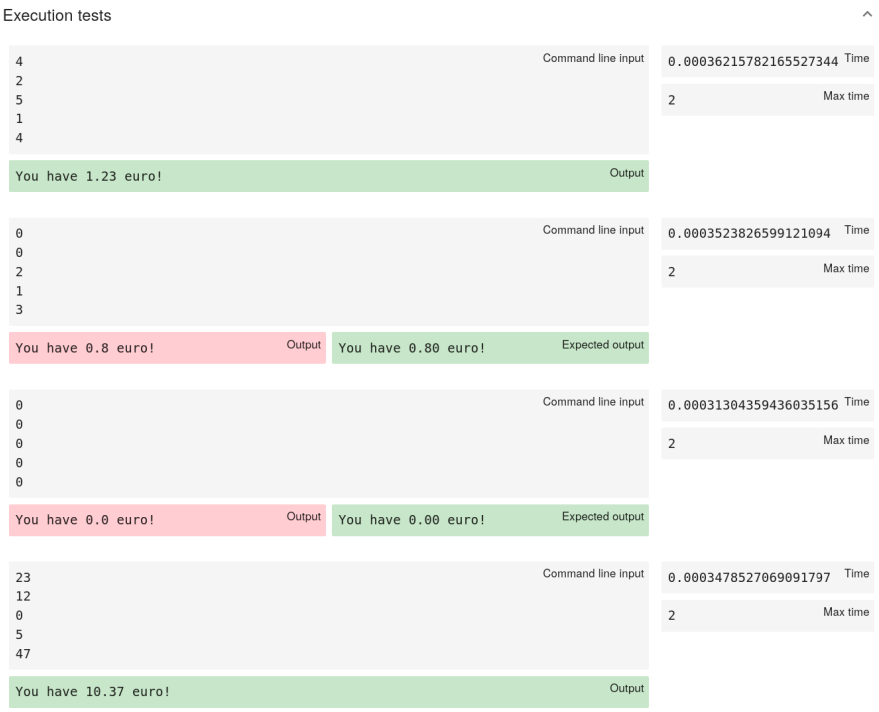


Figure 7.4: The unit test results of a submission containing the code from Listing 5.3.

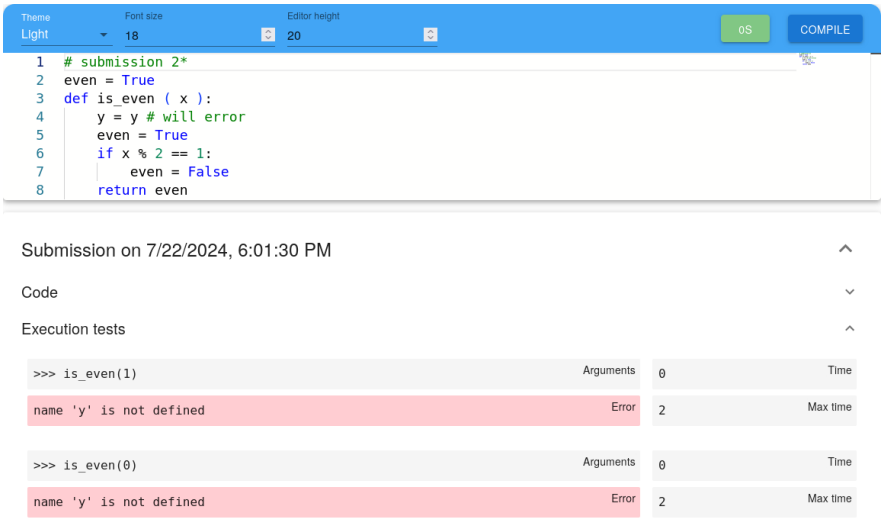


Figure 7.5: The unit test results of an incorrect submission for the exercise described in Section 5.4.

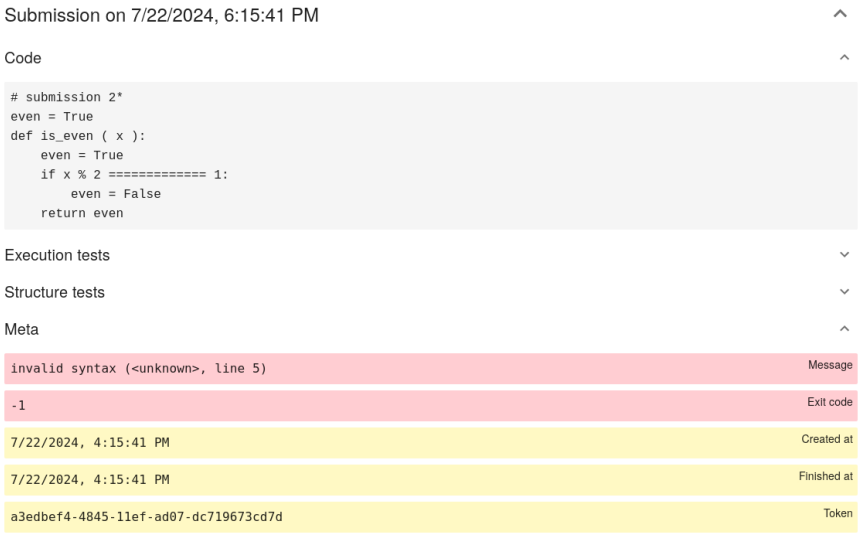


Figure 7.6: The code and meta sections expanded for a submission with a syntax error (due to character '**=**' being repeated).

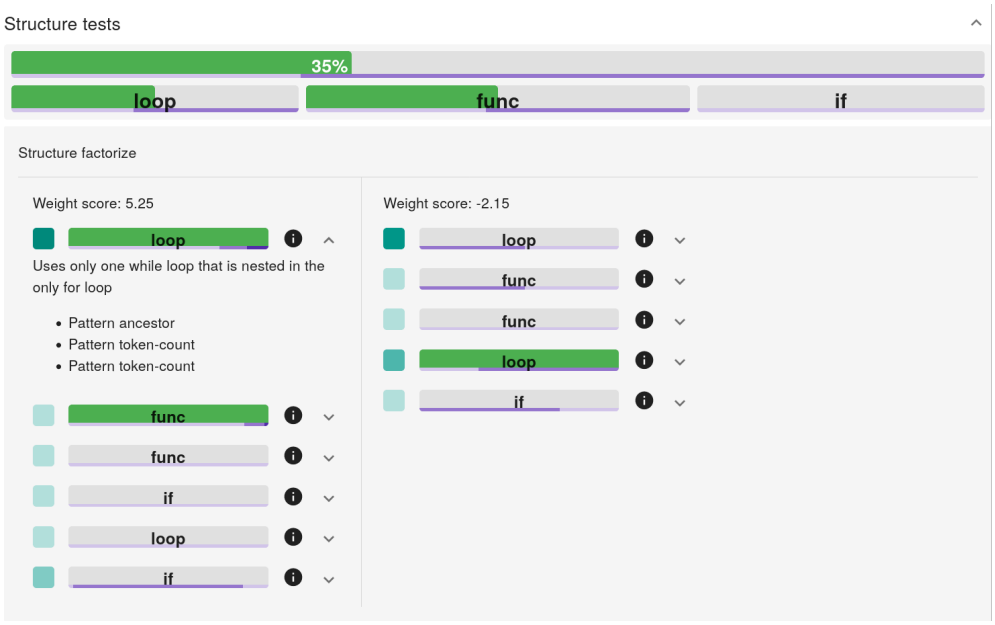


Figure 7.7: An overview of a submission's Structure test section, showing one structure test (row), consisting of two model tests.

7.2.2 Structure-based skill assessment to track exercise progress

In addition to the basic features, the training page also shows a structure section for each submission. When expanded, it will provide an overview of the total progress made through the submission, as well as the progress on the individual skill that adds up to the total progress. Below the submission's progress, a vertical list contains all structure tests, with each structure test represented as an ordered horizontal list of solutions compared against the submitted code. Figure 7.7 shows the progress and one such structure test for Listing 5.15 from Section 5.5, consisting of two model solution tests on `factorize`. The first solution test will check for `for`-based factorizing implementations, whereas the second will check for `while`-based implementations.

To improve the algorithm's scrutability, the interface shows several metrics, such as the overall weight score used to sort the solutions, as well as each skill test's weights (colored boxes), skill tests passing or failing, and allows further inspection by hovering certain elements or expanding the skill test (collapsed by default). Expanding the skill test will display the skill test's description and the patterns that are matched against. For example, Figure 7.7 shows that the first solution has the highest weight, because of the first skill test that passed, correctly identifying the implementation approach of Listing 5.15.

When a student hovers the cursor over a certain pattern, a pop-up appears showing all data associated with that pattern, as shown in Figure 7.8a. It includes the certainty score, the number of occurrences counted of that pattern, the tokens it is based on and the locations in the code between which all tokens are found. Furthermore, nested parts in the structure section and the associated code blocks/scopes will be hierarchically highlighted with a semantic color (yellow for structure, blue for skill, and red for patterns), for which we implemented our own HTML highlighting utilities to correctly place spanning highlights. The borders of hovered parts in the structure section will get the same hierarchical highlights to make it easier to reference the code and the structure test. Similar to pattern test results, a student can hover the cursor over a skill test's results as in Figure 7.8b, or the structure. Unfortunately, the interface does not have the flexibility to move these parts next to each other in the interface. This will obstruct easy referencing whenever the screen space (height) is small, especially when the structure is not listed at the top.

(a) Hovering a pattern test. **(b)** Hovering a skill test.

Figure 7.8: The highlighting associations shown for structure based test results.

7.2.3 Overview of course progress

Besides an environment for solving exercises, the user interface also provides two pages that help track progress, namely the 'PROGRESS' (overview) page and the 'SKILLS' page. The overview page shown in Figure 7.9 contains chapter progress "summaries" at the top, as well as a comprehensive overview of all exercises grouped by chapter. Each of these exercises has a progress bar on the right and an arrow to navigate to the training tab with the exercise loaded in. Clicking on one of the chapter progress summaries will scroll the page to the chapter's exercise panel. With these progress bars, this page provides an implementation of the visualization in Figure 6.5.

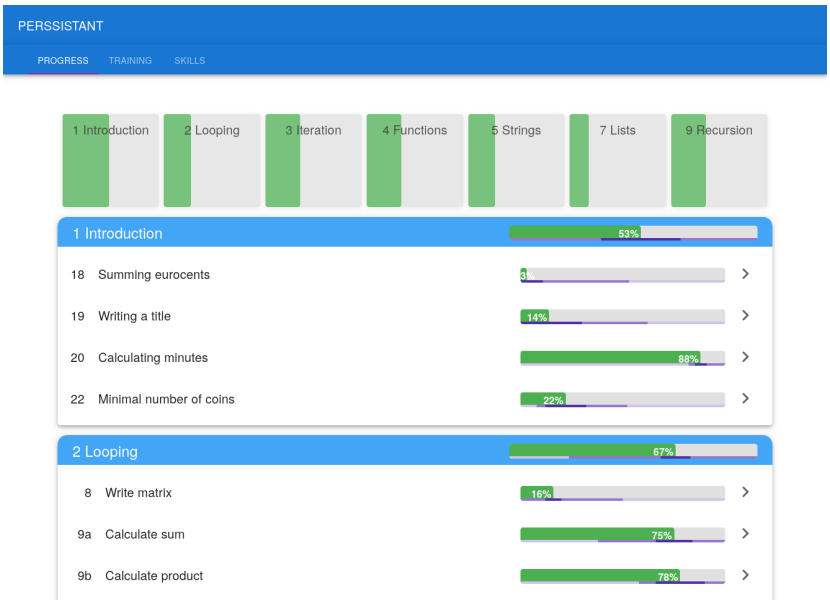


Figure 7.9: An overview of the progress made in the introductory course to programming in Python with random aggregations.

The skill page shown in Figure 7.10 provides more insights into the dependencies between skills and the exercises in which they are tested, which is an implementation of the visualization shown in Figure 6.6.

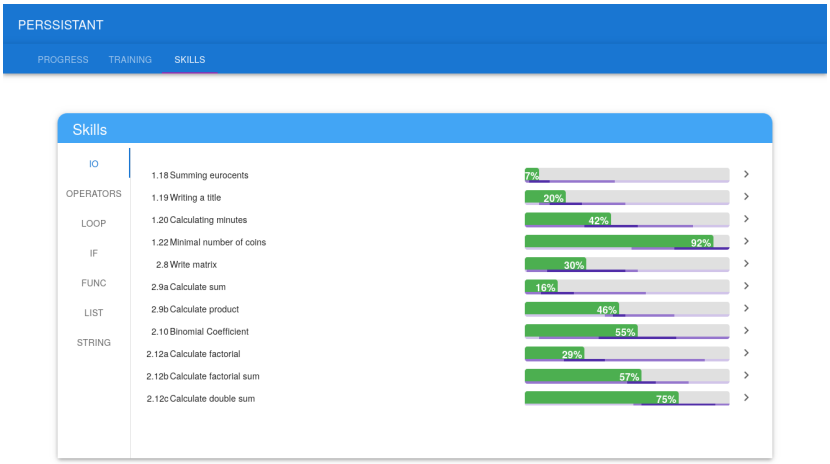


Figure 7.10: An overview of each skill's progress made in the introductory course to programming in Python.

Our custom progress bar component shown in Figure 7.11 was created to resemble the progress bar from Figure 6.2 with a lot of technical flexibility to support different use-cases. It can take variable heights/widths, display the certainty values inline or overflow height of the progress bar, display labels, display an exact progress value, a label, etc. Furthermore, upon hovering, the certainties will expand for better readability, as shown in Figure 7.11b, and hovering a certainty box will display its exact range to the left and the right of the box, as shown in Figure 7.11c.

(a) The default state. **(b)** A hovered state. **(c)** A hovered on certainty bin state.

Figure 7.11: The three states of information display of our progress bar component.

As mentioned before, the algorithm produces a pair of progress and certainty values, *not* distributions. To create a certainty bar that is quickly interpretable, the pair of progress and certainty values is converted to a 'pseudo' distribution. For a high certainty, the pair becomes a left or right skewed distribution when the progress value is extreme ('full' or 'empty') and a normal distribution for more centric progress values ('half'). For a low certainty, a distribution is produced analogously, but is much more flattened.

7.2.4 Proactive exercise recommendation

As noted in Section 3.2, several metrics can be used to decide when to interrupt the student. In PERSSISTANT, a per exercise maximum duration is assumed to be provided by the instructor. When a student starts working on a submission, the timer starts running, hidden by default through the 'HELP' button in Figure 7.2. As soon as the time tracked exceeds the maximum duration, a panel similar to the skill overview page will expand in between the editor's toolbar and the editor's content as shown in Figure 7.12 to display recommendations. If this duration is not present, no proactive recommendation will be done.

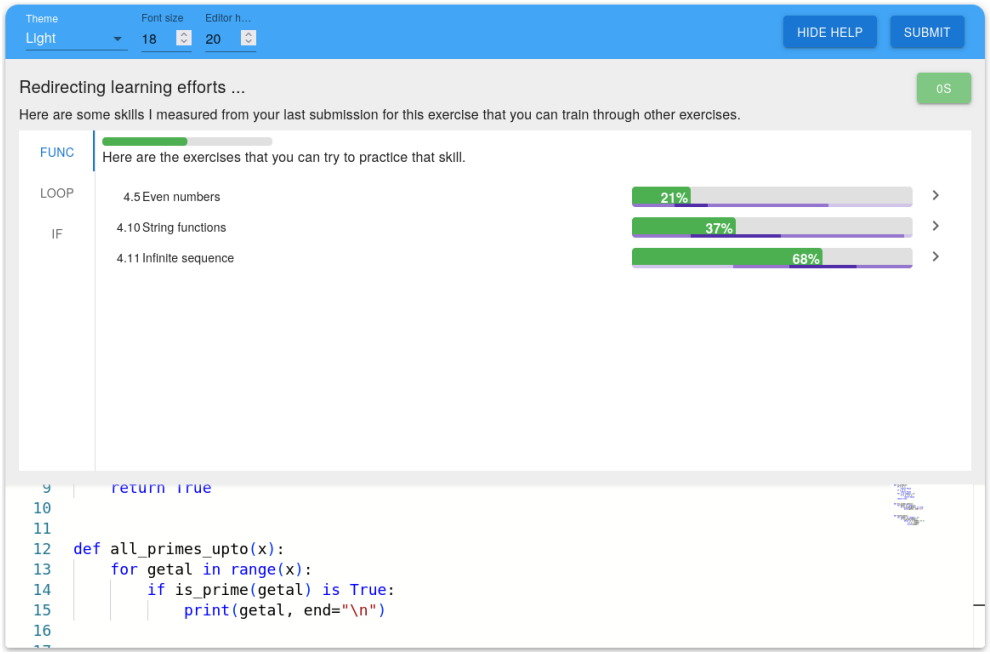


Figure 7.12: An overview of each skill's progress made in the introductory course to programming in Python.

PERSSISTANT also provides user initiative by allowing students to trigger this action, rather than having them wait for that time to have passed. This can be useful in the scenario described in Section 3.1, where the student's understanding of a concept was challenged, and therefore wanted to test his understanding with a new exercise. PERSSISTANT supports this scenario partially, since a student can verify the system's belief on the current progress and select an exercise, but cannot correct the system.

To provide recommendations, all exercises are listed that precede the current exercise in the standard trajectory of submissions. These are further ordered by the weakest skill not passing a 50% progression and has the most weight. Note however that the skill progressions assessed for exercises – rather their final submission – that have already been completed are not taken into account.

In the following chapter, we will discuss the concepts and implementation of PERSSISTANT.

Chapter 8

Discussion

In this chapter, we will provide a discussion on the concepts and implementation of this thesis. We will provide a critical self-reflection of our work and provide points of improvement for future work. In Section 8.1, we will further discuss applications of DPP. In Section 8.2, we will further discuss intermediate exercises. In Section 8.3, we will review our approach to assessment. In Section 8.4, we will review the usability of PERSSISTANT by the required preparation. In Section 8.5, we will take an educational perspective at proactive feedback provided in our concepts and PERSSISTANT. Finally, in Section 8.6, we will discuss the opportunities of monitoring progress.

8.1 Applicability of Dynamic Path Proposing

The concepts of DPP do not apply to just practical exercises, but, more generally, can be used in other fields requiring a specialized agent. When it comes to processing reading material, we envision that an LLM can be tailored to act like an interrogator to briefly verify whether students sufficiently understood what they have been reading after finishing a section. The intermediate exercises then become all sorts of assessments, not necessarily constrained to the debugging, template or extensions associated with programming exercises. For instance, students can be asked to review insufficient or wrong answers themselves, fill in specific details, or be asked follow-up questions (analog to DPP's intermediate exercises) in addition to open-ended questions, multiple-choice questions, look-up questions, etc. Given this assessment, students can be interrupted and redirected to other sections, since handbooks often contain multiple outgoing dependencies to other sections.

As noted before, DPP requires the agent to sufficiently estimate the student's understanding of the concepts they are training. In the case of programming, we used structure-based skill assessment in PERSSISTANT to enable DPP and could have used other approaches applicable to broader domains discussed in Section 2.3. However, we cannot make any claims about other domains that fall outside the range, nor did we demonstrate how KT or IRT can be used to support DPP, so this remains future work.

8.2 Supporting intermediate exercises

While the idea of providing intermediate exercises is enticing to support better personalization, in practice, it is not feasible for human instructors, as we noted briefly before. This is because the more exercises are created, the more time is needed to develop useful ones that account for the time students take to complete those before (or after) making traditional exercises, compared to when they were never interrupted at all.

Furthermore, it requires an ongoing effort to keep the exercises aligned with the course, which will (usually) get updated over time. For instance, instructors using Python in their introductory programming courses will likely want to update their exercises when updates are made to the language that may be useful to their students. The recent addition of the `match` statement is one such example. Updating intermediate exercises, in addition to the traditional ones, will likely discourage instructors from creating them. We also mentioned an approach where instructors can review LLM-generated intermediate exercises, but this is likely still too time-consuming to provide enough intermediate exercises, so the possibilities of personalization remain limited to work mainly with traditional exercises.

In our opinion, the only viable way to support intermediate exercises is by using an LLM to automatically create them. This removes the constraint of having only a limited number of exercises, at the cost of human review. In Chapter 4, we have attempted to generate intermediate programming exercises using GPT3.5. However, during the thesis, newer models have been released that are publically available, such as OpenAI's GPT4o. Furthermore, our exploration was limited. Future work should continue this exploration, which can be supported as an exercise generating extension to PERSSISTANT. The LLM can be provided information on the student's unmastered skills to come up with an exercise tailored to their needs.

Instructors will still have to adapt to students using this new form of practicing. Similar to students asking instructors for help with random exercises, they may start asking questions and help with their generated exercises, but on a daily basis – ignoring the possibility of the exercise being flawed or beyond their level. Therefore, experience with the traditional exercises may become less helpful to their instruction, which may cost them more time.

8.3 Skill assessment and feedback

In our work, we explored the use of ASTs to assess students' skills that directly relate to constructs. We attempted to provide a method that is sufficiently broad to cover real-world submissions. However, our testing is limited to the examples used in Chapter 6, and therefore we cannot yet justify our somewhat complicated approach, nor dismiss its potential. Future work should further create assessment data, so that our method can be compared with contemporary methods discussed in Chapter 2. Aside to the missing results, we describe two other types of tests hinted at in Chapter 5 that can further expand the PERSSISTANT's feedback to students.

Runtime-based checks In Section 5.2, we described how students used tolerable, but unnecessary hacks to result in a working solution. As part of our early exploration, we attached an interpreter from `astinterp` to a running submission to extract the start and stop values of the range. This was used to create warnings whenever either 1) there was no range at all, 2) the range was bigger than the number read from the terminal, or 3) zero is summed up. These tests would accurately warn cases like Listings 5.6, 5.5 and 5.4 respectively. Likewise, for the exercise in Section 5.3, we automatically created hints whenever a submission's call to `range` resulted in a different range size to the size of the input string to be reversed. We found with our limited tests that run-time based checks are cheaper to write down (once provided an interpreter) than checking the AST statically. We believe that providing additional feedback, in such situations, maybe helpful for students to better grasp the concepts, however, a declarative format is necessary for teachers to properly support this.

Token-based checks In Section 5.1, we already mentioned using specific tests (checking for a period strings) to verify if a student is using the intended functionality. Looking more generally at the set of tokens (`if`, `//`, `%`, etc.), students can be warned whenever they are using tokens that are not in the set of tokens they are expected to use up until that exercise. Furthermore, students can be nudged to fine-tune their solutions, whenever they are using constructs that are not needed by the model, which is helpful for exercises from Sections 5.4 or 5.5.

8.4 Course preparation and maintenance

In Section 6.4 and Section 6.3, we briefly mentioned that instructors are supposed to provide hierarchically structured testing data to automate the high-level assessment. There are a few problems with this requirement. First, providing all these tests already requires a lot of time, which defeats the argument of AI freeing up instructors' time to spend on more personalized instruction and does not motivate instructors who already find themselves constrained in time. Second, instructors are supposed to provide a negative and positive bias for each pattern test, a weight w_j for skill application test a_j, a weight v for each exercise skill s^e, and preferably a weight u_h for each skill aggregated per chapter or course. It takes a lot of fine-tuning on model solutions to get these weights and biases right. Third, while it may seem that this approach avoids the cold start problem because instructors only need their model solutions to construct the hierarchical testing data, this perspective overlooks the time required to fine-tune the data with real-world submissions (gathered over years). Fourth, instructors may find the best values over time, given enough submissions, but this is cumbersome. Using some ML algorithm to find the optimal values would defeat the argument of using our approach since, in that case, perhaps a sufficient number of submissions exist already to use other mature, well-tested methods as described in Section 2.3.

The focus of our work was however to investigate automated skill assessment, while better declarative tools are outside the scope. Future work should, therefore, investigate creating an interactive tool for instructors to create this testing data (unlike the current approach, which is laying all the declarative information down in JSON, which is too generic and does not provide convenience features such as auto-completion). Such a tool would speed up this process considerably, by providing a preview of all model solutions to an exercise (or structure) and its real-time assessment, based on the patterns provided. The preview may show which tokens will be selected, given a certain pattern, similar to how students can elicit explanations by hovering patterns, as described in Section 7.2.2. Using real-time progress visualizations would also help with defining weights and biases at the exercise level. This should also cause less friction to update these values when instructors want to update exercises (e.g., when a student came up with a valid model they had not thought of yet).

8.5 Proactive feedback and educational perspective

Aside from the quality of the feedback provided by PERSSISTANT, it is important to note that we did not receive input from educational experts regarding our approach to proactively interrupting the student's learning process. We believe the impact to be positive, but we should also be careful to make assumptions. Therefore, further research is needed to explore the implications of proactively interrupting students during their learning process. For instance, in addition to course concepts, students must also learn practical skills, such as how to deal with getting stuck, develop strategies to get 'unstuck', make plans for themselves, etc. A tool that keeps track of their progress may take away the opportunities for them to reflect on their performance and come up with new strategies.

Furthermore, it may promote negative (lazy) strategies. Students may learn through experience that an intervention will happen if they just wait long enough. Rather than attempting to solve the exercise, they may give up after a while, knowing that soon they will be assisted. Therefore, our simple instructor-provided time metric for PERSSISTANT may be particularly prone to students "gaming the assistance", as it does not take into account the number of times a student was helped in the past.

Finally, students who are motivated to work on exercises may benefit more from continuing to work on them even when they are progressing slowly, rather than being redirected to do different learning activities. The proposed alternative(s) may not be aligned with their interests, which could result in them disengaging from both the current exercise and the alternative(s). On the other hand, their interests for certain exercises may be used to introduce them to others.

8.6 Monitoring progress

Finally, monitoring progress opens up the opportunity for instructors to get real-time feedback on how students master knowledge. Future work should investigate the analysis of students' progressions to provide useful visualizations to instructors, so they can make informed decisions on where to improve their courses. Such tools can indicate which lectures, what reading material, or which exercises lead to the most progress increase. The ones that did not result in measurable progress may be improved upon.

Furthermore, having these visualizations in real-time enables instructors to get insights into the current progression students make. Such a tool may indicate how far behind students are, or whether a significant part of the student population seems to get stuck at a certain part, without asking for help. With this information, they can nudge them to ask questions related to these topics, or even prepare the next lecture to also reiterate the difficult concepts where students get stuck. Furthermore, if the tool is used over multiple consecutive courses, they can provide insights in the students' learned skills, and whether they are significantly lower than past years due to a shift of focus on certain topics in previous courses.

Such a monitoring tool should, however, respect students' privacy. Allowing instructors to inspect the progress of individual students could lead to situations where an instructor might say, "I noticed you have not attempted the previous exercises, so I will not assist you yet until you have made a visible effort." While students should make an effort, this approach could create a negative learning environment and discourage them from seeking help. To mitigate these concerns, while still providing useful feedback to instructors, future work should also look into the ways in which the progress of students can be anonymized or aggregated.

Chapter 9

Conclusion

9.1 Contribution

In our research, we focused on monitoring learning progress to personalizing the learning trajectory, applied in the domain of programming. We investigated three main questions:

- Q1: How can we effectively leverage AI to continuously monitor the progress of a single student in an introductory programming course, ensuring accurate tracking of their knowledge acquisition?
- Q2: What are the most effective methods for communicating real-time progress to students, so they can make informed decisions on the next learning unit?
- Q3: How can AI-driven progress monitoring be used to personalize students' learning trajectories?

From monitoring student progress, our research revealed several key insights. First, our structure-based approach to skill assessment requires considerable preparation time. Second, this approach is useful to support multiple student solutions, while still providing each of them detailed feedback tailored to their code. Third, our pattern-based approach initially showed to be sufficient to support small coding exercises. However, from the exploration of student submissions, it became clear that some (larger) exercises have different solutions that would require extensive modeling. To mitigate this issue, providing larger templates that constrain students to solve their exercises within the template can be beneficial.

Additionally, we explored various methods for communicating the monitored progress. First, we found that system certainty in conjunction with progress can be represented using a second bar for a binned distribution-like certainty indication. In our opinion, this visualization is easy to interpret and compare with other progress shown in an overview. Second, while we did not implement aggregation in our prototype at the chapter, course, or skill level, we did provide formal methods to aggregate these results. We found that aggregating progress using our methods is a non-trivial task. Third, we also made an initial attempt to make the algorithm more transparent. In the future, we would further like to support the student correcting the assessment. This can be necessary when the set of model solutions does not yet encompass a unique, valid one, or high-level predicates (skill application tests) are not mapped to sufficient low-level patterns.

Finally, our initial approach to providing proactive feedback was not explored extensively as the other research questions. In our early exploration, we found that it is difficult to keep track of whether students are solving exercises or perhaps taking a break. Furthermore, our approach to recommendation is yet too superficial, as it usually lists all preceding exercises in the trajectory – a trivial task a student can already do.

Looking ahead, future research should explore the long-term impacts of our AI-driven personalized learning approach. To continue this research, our first priority would be developing an instructor client that allows instructors to easily create assessment content. This paves the way for conducting measurable experiments, as more assessment data can be generated. In a broader scope, further investigations into the scalability of DPP across educational contexts are necessary to determine its effectiveness. Additionally, using LLMs to extend and further explore intermediate exercises could lead to more engaging and challenging learning experiences for students.

9.2 Reflection

Throughout the process of completing my master's thesis, I, Tijl Elens, have gained valuable insights into my personal learning journey and the challenges I faced. Reflecting on my experiences, I recognize areas where I excelled and others that require further improvement.

Reporting progress One key lesson learned was the importance of regular progress reporting, even when working independently. I found that I often procrastinated on reporting results due to the time required to prepare and bring clear progress presentations. At the time, I felt this time was better spent on further progression, rather than showing little progress frequently. This made it difficult, however, to gauge if I was moving in the right direction or spending too much time on less relevant tasks. While I was engaged with various aspects of the project, I sometimes overlooked essential tasks, such as working on effective proactive interruption or further prototype testing, which are crucial for overall progress.

Planning Planning is an important point of improvement. In my bachelor's thesis, I learned that with big projects, I can focus too deeply on certain parts. For the master's thesis, I tried to follow the 80-20% rule, given that the work usually needed review anyway. However, this approach is not compatible with my tendency to work on tasks that are not always as essential to the overall progress. As a result, I was left with many unfinished tasks alongside the crucial work that needed to be completed close to the deadline.

Writing Even though the text has not reached the quality I initially laid forward, I believe this experience has been a valuable practice for doing research long-term. Bringing concepts clearly and processing feedback are important researcher skills. Compared with the bachelor's thesis, I was able to produce more text of reasonable quality in half the number of months writing it. However, I recognize that I missed the opportunity to elevate the thesis text to its full potential, as I was unable to process all valuable feedback in a timely manner. Specifically, I would have liked to incorporate an additional exploration of the recently released GPT4o, using the same prompts, however, I decided not to as including the results would require time I had not left. Additionally, I requested all submissions directly to the professor of the course, rather than bypassing by asking peers. At the time, I was not sure yet about the usefulness, until I moved on from analysis to continue working on the prototype. Finally, I acknowledge that the concepts described take many shortcuts in explanation, which renders them inaccessible to a reader not familiar with the methods. On a positive note, I tackled writing the related work from scratch with a new overviewing method that – I believe – will help me significantly keeping track of the papers in the future.

Prototyping Regarding the prototype, I decided on a file-server rather than using a proper SQL-database or something alike. The choice for the former definitely helped in quickly showing an initial prototype, but impeded the ability to easily extend new data structures or relations between exercises, submissions, and skills. If I were to redo this project, I would definitely investigate setting up a proper database.

Workflow Despite these challenges mentioned in previous paragraphs, I am proud of my ability to maintain a sustainable workflow throughout the year-long thesis period, in contrast to the unhealthy amount of time I dedicated to my bachelor's thesis. Even though I encountered macroscale fluctuations in motivation (as described by Tetzlaff et al. [TSB21]), I was able to persist working on the thesis and always continued to make progress. This demonstrates growth in my time management and self-care skills.

In conclusion, my master's thesis journey has been a transformative experience, filled with both challenges and successes. I have gained valuable insights into my communication style, decision-making abilities, and time management skills. While there are areas for improvement, I am confident that the lessons learned will continue to shape my growth as a researcher and individual.

Bibliography

[AWN23] Ghodai Abdelrahman, Qing Wang, and Bernardo Nunes. "Knowledge Tracing: A Survey". In: *ACM Comput. Surv.* 55.11 (Feb. 2023). ISSN: 0360-0300. DOI: 10.1145/3569576. URL: https://doi.org/10.1145/3569576.

[Bec+23] Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. "Programming is hard-or at least it used to be: Educational opportunities and challenges of ai code generation". In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 2023, pp. 500–506.

[BO23] David Baidoo-anu and Leticia Owusu Ansah. "Education in the Era of Generative Artificial Intelligence (AI): Understanding the Potential Benefits of ChatGPT in Promoting Teaching and Learning". In: *Journal of AI* 7.1 (2023), pp. 52–62. DOI: 10.61969/jai.1337500.

[BSH96] Joseph Beck, Mia Stern, and Erik Haugsjaa. "Applications of AI in Education". In: *XRDS: Crossroads, The ACM Magazine for Students* 3.1 (1996), pp. 11–15.

[BWP21] Anirudhan Badrinath, Frederic Wang, and Zachary Pardos. "pyBKT: An Accessible Python Library of Bayesian Knowledge Tracing Models". In: *Proceedings of the 14th International Conference on Educational Data Mining*. 2021, pp. 468–474.

[CA94] Albert T Corbett and John R Anderson. "Knowledge tracing: Modeling the acquisition of procedural knowledge". In: *User modeling and user-adapted interaction* 4 (Dec. 1994), pp. 253–278. DOI: 10.1007/BF01099821.

[CCL20] Lijia Chen, Pingping Chen, and Zhijian Lin. "Artificial Intelligence in Education: A Review". In: *IEEE Access* 8 (2020), pp. 75264–75278. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2988510.

[Dij+22] Ramon Dijkstra, Zülküf Genç, Subhradeep Kayal, Jaap Kamps, et al. *Reading Comprehension Quiz Generation using Generative Pre-trained Transformers*. 2022.

[EKR17] Stephen H. Edwards, Nischel Kandru, and Mukund B.M. Rajagopal. "Investigating Static Analysis Errors in Student Java Programs". In: *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ICER '17. Tacoma, Washington, USA: Association for Computing Machinery, 2017, pp. 65–73. ISBN: 9781450349680. DOI: 10.1145/3105726.3106182. URL: https://doi.org/10.1145/3105726.3106182.

[Fos+15] Davide Fossati, Barbara Di Eugenio, STELLAN Ohlsson, Christopher Brown, and Lin Chen. "Data driven automatic feedback generation in the iList intelligent tutoring system". In: *Technology, Instruction, Cognition and Learning* 10.1 (2015), pp. 5–26.

[Fre+23] Adrian de Freitas, Joel Coffman, Michelle de Freitas, Justin Wilson, and Troy Weingart. "FalconCode: A Multiyear Dataset of Python Code Samples from an Introductory Computer Science Course". In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. SIGCSE 2023. ¡conf-loc¿, ¡city¿Toronto ON¡/city¿, ¡country¿Canada¡/country¿, ¡/conf-loc¿: Association for Computing Machinery, 2023, pp. 938–944. ISBN: 9781450394314. DOI: 10.1145/3545945.3569822. URL: https://doi.org/10.1145/3545945.3569822.

[GKS19] Rahul Gupta, Aditya Kanade, and Shirish Shevade. "Neural Attribution for Semantic Bug-Localization in Student Programs". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/f29a179746902e331572c483c45e5086-Paper.pdf.

[Guo15] Philip J Guo. "Codeopticon: Real-time, one-to-many human tutoring for computer programming". In: *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. 2015, pp. 599–608.

[Hob19] Sebastian Hobert. "Say hello to 'coding tutor'! design and evaluation of a chatbot-based learning system supporting students to learn to program". In: (2019).

[Hov+16] David Hovemeyer, Arto Hellas, Andrew Petersen, and Jaime Spacco. "Control-Flow-Only Abstract Syntax Trees for Analyzing Students' Programming Progress". In: *Proceedings of the 2016 ACM Conference on International Computing Education Research*. ICER '16. Melbourne, VIC, Australia: Association for Computing Machinery, 2016, pp. 63–72. ISBN: 9781450344494. DOI: 10.1145/2960310.2960326. URL: https://doi.org/10.1145/2960310.2960326.

[Kas+23] Enkelejda Kasneci et al. "ChatGPT for good? On opportunities and challenges of large language models for education". In: *Learning and Individual Differences* 103 (2023), p. 102274. ISSN: 1041-6080. DOI: https://doi.org/10.1016/j.lindif.2023.102274. URL: https://www.sciencedirect.com/science/article/pii/S1041608023000195.

[Kou22] Charles Koutcheme. "Towards Open Natural Language Feedback Generation for Novice Programmers using Large Language Models". In: *Proceedings of the 22nd Koli Calling International Conference on Computing Education Research*. Koli Calling '22. Koli, Finland: Association for Computing Machinery, 2022. ISBN: 9781450396165. DOI: 10.1145/3564721.3565955. URL: https://doi.org/10.1145/3564721.3565955.

[Kun+23] Tiffany H. Kung et al. "Performance of ChatGPT on USMLE: Potential for AI-assisted medical education using large language models". In: *PLOS Digital Health* 2.2 (Feb. 2023), pp. 1–12. DOI: 10.1371/journal.pdig.0000198. URL: https://doi.org/10.1371/journal.pdig.0000198.

[Laa+23] Samuli Laato, Benedikt Morschheuser, Juho Hamari, and Jari Björne. "AI-assisted Learning with ChatGPT and Large Language Models: Implications for Higher Education". In: July 2023.

[Liu+19] Xiao Liu, Shuai Wang, Pei Wang, and Dinghao Wu. "Automatic Grading of Programming Assignments: An Approach Based on Formal Semantics". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. 2019, pp. 126–137. DOI: 10.1109/ICSE-SEET.2019.00022.

[Liu+22] Naiming Liu, Zichao Wang, Richard Baraniuk, and Andrew Lan. "Open-ended knowledge tracing for computer science education". In: *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. 2022.

[Mag+21] Setareh Maghsudi, Andrew Lan, Jie Xu, and Mihaela van der Schaar. "Personalized Education in the Artificial Intelligence Era: What to Expect Next". In: *IEEE Signal Processing Magazine* 38.3 (2021), pp. 37–50. DOI: 10.1109/MSP.2021.3055032.

[MDP20] Samiha Marwan, Anay Dombe, and Thomas W Price. "Unproductive help-seeking in programming: What it is and how to address it". In: *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. 2020, pp. 54–60.

[MG23] Sruti Mallik and Ahana Gangopadhyay. "Proactive and reactive engagement of artificial intelligence methods for education: a review". In: *Frontiers in Artificial Intelligence* 6 (2023). ISSN: 2624-8212. DOI: 10.3389/frai.2023.1151391. URL: https://www.frontiersin.org/articles/10.3389/frai.2023.1151391.

[MHP23] Samim Mirhosseini, Austin Z Henley, and Chris Parnin. "What is your biggest pain point? an investigation of cs instructor obstacles, workarounds, and desires". In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 2023, pp. 291–297.

[Moo+22] Delaney Moore, John Edwards, Hamid Karimi, Rajiv Khadka, and Paul Bodily. "Temporal Abstract Syntax Trees for Understanding Student Coding Thought Process". In: *2022 Intermountain Engineering, Technology and Computing (IETC)*. 2022, pp. 1–6. DOI: 10.1109/IETC54973.2022.9796943.

[NH24] Anh-Tu Phuong Nguyen and Van-Dung Hoang. "Development of Code Evaluation System based on Abstract Syntax Tree". In: *Journal of Technical Education Science* 19.Special Issue 01 (Feb. 2024), pp. 15–24. DOI: 10.54644/jte.2024.1514. URL: https://jte.edu.vn/index.php/jte/article/view/1514.

[NMS22] Bharatwaja Namatherdhala, Noman Mazher, and Gopal Krishna Sriram. "A comprehensive overview of artificial intelligence tends in education". In: *International Research Journal of Modernization in Engineering Technology and Science* 4.7 (2022).

[Nwa90] HyacinthS. Nwana. "Intelligent tutoring systems: an overview". In: *Artificial Intelligence Review* 4.4 (1990). ISSN: 1573-7462. DOI: 10.1007/bf00168958. URL: http://dx.doi.org/10.1007/BF00168958.

[Pel17] Radek Pelánek. "Bayesian knowledge tracing, logistic models, and beyond: an overview of learner modeling techniques". In: *User Modeling and User-Adapted Interaction* 27 (2017), pp. 313–350.

[Pie+15] Chris Piech, Jonathan Bassen, Jonathan Huang, Surya Ganguli, Mehran Sahami, Leonidas J Guibas, and Jascha Sohl-Dickstein. "Deep Knowledge Tracing". In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett. Vol. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper_files/paper/2015/file/bac9162b47c56fc8a4d2a519803d51b3-Paper.pdf.

[PK17] Stefan Popenici and Sharon Kerr. "Exploring the impact of artificial intelligence on teaching and learning in higher education". In: *Research and Practice in Technology Enhanced Learning* 12.1 (Nov. 2017). DOI: 10.1186/s41039-017-0062-8. URL: https://doi.org/10.1186/s41039-017-0062-8.

[RAM18] Ljupcho Rechkoski, Vangel V. Ajanovski, and Marija Mihova. "Evaluation of grade prediction using model-based collaborative filtering methods". In: *2018 IEEE Global Engineering Education Conference (EDUCON)*. 2018, pp. 1096–1103. DOI: 10.1109/EDUCON.2018.8363352.

[RCH18] Rolf Reber, Elizabeth A. Canning, and Judith M. Harackiewicz. "Personalized Education to Increase Interest". In: *Current Directions in Psychological Science* 27.6 (2018), pp. 449–454. DOI: 10.1177/0963721418793140. eprint: https://doi.org/10.1177/0963721418793140. URL: https://doi.org/10.1177/0963721418793140.

[ROS10] Danijel Radošević, Tihomir Orehovački, and Zlatko Stapić. "Automatic on-line generation of student's exercises in teaching programming". In: *Radošević, D., Orehovački, T., Stapić, Z:" Automatic On-line Generation of Students Exercises in Teaching Programming", Central European Conference on Information and Intelligent Systems, CECIIS*. 2010.

[Ros22] Max Roser. "The brief history of artificial intelligence: The world has changed fast – what might be next?" In: *Our World in Data* (2022). https://ourworldindata.org/brief-history-of-ai.

[RW16] Ido Roll and Ruth Wylie. "Evolution and revolution in artificial intelligence in education". In: *International Journal of Artificial Intelligence in Education* 26.2 (Feb. 2016), pp. 582–599. DOI: 10.1007/s40593-016-0110-3. URL: https://doi.org/10.1007/s40593-016-0110-3.

[Sal20] Pedro Salazar Paredes. "Comparing python programs using abstract syntax trees". In: (2020).

[SGA08] Silvia Schiaffino, Patricio Garcia, and Analia Amandi. "eTeacher: Providing personalized assistance to e-learning students". In: *Computers & Education* 51.4 (2008), pp. 1744–1754. ISSN: 0360-1315. DOI: https://doi.org/10.1016/j.compedu.2008.05.008. URL: https://www.sciencedirect.com/science/article/pii/S036013150800078X.

[Shi+22] Yang Shi, Min Chi, Tiffany Barnes, and Thomas Price. "Code-DKT: A Code-based Knowledge Tracing Model for Programming Tasks". In: (2022). DOI: https://doi.org/10.48550/arXiv.2206.03545. arXiv: 2206.03545 [cs.SE].

[SS14] Piyanuch Silapachote and Ananta Srisuphab. "Gaining and maintaining student attention through competitive activities in cooperative learning A well-received experience in an undergraduate introductory Artificial Intelligence course". In: *2014 IEEE Global Engineering Education Conference (EDUCON)*. Apr. 2014, pp. 295–298. DOI: 10.1109/EDUCON.2014.6826106.

[TSB21] Leonard Tetzlaff, Florian Schmiedek, and Garvin Brod. "Developing personalized education: A dynamic framework". In: *Educational Psychology Review* 33 (Sept. 2021), pp. 863–882. DOI: 10.1007/s10648-020-09570-w.

[Wan+17] Lisa Wang, Angela Sy, Larry Liu, and Chris Piech. "Learning to Represent Student Knowledge on Programming Exercises Using Deep Learning." In: *International Educational Data Mining Society* (2017).

[Wan+23] Tianjia Wang, Daniel Vargas-Diaz, Chris Brown, and Yan Chen. *Towards Adapting Computer Science Courses to AI Assistants' Capabilities*. 2023. arXiv: 2306.03289 [cs.HC].

[WMB24] Ben Williamson, Alex Molnar, and Faith Boninger. *Time for a Pause: Without Effective Public Oversight, AI in Schools Will Do More Harm Than Good*. 2024.

[Zaf+22] Fabiana Zaffalon, André Prisco, Ricardo De Souza, Davi Teixeira, Wanderson Paes, Paulo Evald, Neilor Tonin, Sam Devincenzi, and Silvia Botelho. "A Recommender System of Computer Programming Exercises based on Student's Multiple Abilities and Skills Model". In: *2022 IEEE Frontiers in Education Conference (FIE)*. 2022, pp. 1–8. DOI: 10.1109/FIE56618.2022.9962646.

[Zhe+22] Wei Xing Zheng, Qing Du, Yongjian Fan, Lijuan Tan, Chuanlin Xia, and Fengyu Yang. "A personalized programming exercise recommendation algorithm based on knowledge structure tree". In: *Journal of Intelligent and Fuzzy Systems* 42.3 (Feb. 2022), pp. 2169–2180. DOI: 10.3233/jifs-211499. URL: https://doi.org/10.3233/jifs-211499.