UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Exploring the pedagogic aspects of programming in the framework of computer science education

Robin Weynjes

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

PROMOTOR :

Prof. dr. Frank NEVEN

2024
2025

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

Exploring the pedagogic aspects of programming in the framework of computer science education

Robin Weynjes

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

PROMOTOR :

Prof. dr. Frank NEVEN

Acknowledgements

This master's thesis is the culmination of everything I have learned during my time at Hasselt University. Starting out as someone with no prior programming experience, I have seen myself grow year after year as I learned more knowledge and skills, eventually resulting in this thesis. This would not have been possible without some people, whom I am very thankful for.

First and foremost, I would like to thank my promoter, prof. dr. Frank Neven. His expertise, guidance, and feedback have been invaluable and have helped me countless times during my thesis. By having frequent meetings, I was always able to report my progress and have immediate feedback, for which I am very grateful.

I would also like to thank all the professors and teaching assistants I have received lessons and guidance from throughout the years. It is thanks to their lessons and explanations that I have learned the skills I have today. As described in this thesis, teaching computer science is far from a trivial task, and they have done an amazing job with it.

Finally, I would also like to thank my parents and my friends for their never-ending support and reassurance whenever I was facing difficulties. The process of creating this thesis has not always been a smooth ride, and I could always rely on them when I needed it.

Nederlandse Samenvatting

Deze thesis bestaat uit twee grote delen. Het eerste deel beschrijft de belangrijkste literatuur over de educatieve aspecten van de informatica. Hoofdstuk twee beschrijft de belangrijkste moeilijkheden die beginnende studenten informatica ondervinden. Leren programmeren is een erg moeilijke opgave, en er zijn veel zaken waar een beginnende student tegenaan kan lopen. Leren programmeren is meer dan alleen syntax en semantiek, men moet ook weten hoe men precies de functionaliteit van een taal kan gebruiken. Zonder deze kennis kan men wel weten hoe programma's werken, maar zelf programma's schrijven is erg moeilijk als deze ontbreekt. Er zijn misconcepties die op verschillende gebieden kunnen optreden, van programmeertalen die anders werken met getallen tot de semantiek van sleutelwoorden die in contrast staan tot de betekenis van deze waarden in normaal taalgebruik.

Hoofdstuk drie beschrijft verschillende manieren en soorten oefeningen die gebruikt kunnen worden voor formatieve evaluatie. Dit is evaluatie waar het belangrijkste doel is om ervoor te zorgen dat de student bijleert en om te toetsen of de student mee is met de leerstof, zonder hier punten of een strikte evaluatie op te plakken. Er bestaan verschillende soorten manieren om les te geven, en een van de dominantste stromingen hierin momenteel is actief leren, waarbij de student actief betrokkken wordt en niet passief kennis vergaat zoals bij klassiek lesgeven.

Hoofdstuk vier beschrijf verschillende modellen om een probleem waarvoor geen triviale oplossing bestaat aan te pakken. Dit is een creatief proces dat moeilijk is voor beginnende studenten. Het laat zich niet vangen in strikte regels en het kan moeilijk zijn om een oplossing te vinden voor een bepaald probleem, zeker voor beginnende studenten. Gelukkig zijn er een aantal methoden die verschillende fases afbakenen en voor iedere fase taken definiëeren die uitgevoerd moeten worden om tot een oplossing te komen. Het aanleren van zo een stappenplan bevordert het vertrouwen en het probleemoplossend vermogen van studenten.

Tot zover het eerste deel van deze thesis, het tweede deel gaat over het verrichte onderzoek en een bijdrage aan een universitair vak.

Hoofdstuk vijf gaat over een bevraging die afgenomen werd van studenten informatica die pas begonnen zijn aan de opleiding. De bevraging polst naar hun achtergrond in programmeren, wiskunde, en hun opleidingsniveau en onderzoekt mogelijke verbanden tussen hun achtergrond en de behaalde punten op het eerste programmeervak. Uit deze analyse komt naar voren dat geen van de onderzochte achtergrondskenmerken statistisch significant zijn in relatie met de behaalde punten, maar de steekproef is ook relatief klein wat het moeilijk maakt om definitief uit te sluiten dat er geen invloed is. Vooral de voorafgaande programmeerervaring zat relatief dicht bij het significantieniveau. Verder polste de bevraging ook naar hoe studenten hun bekwaamheid inschatten in verschillende domeinen van de informatica zoals probleemoplossend denken. Ook hun zelfbeoordeling op deze domeinen werd geanalyseerd, en als hun zelfbeoordeling hoger was, dan hadden ze ook algemeen hogere punten. Dit was vooral het geval bij probleemoplossend denken, wat er mogelijks op wijst dat dit de belangrijkste vaardigheid is voor een student om te leren. Het verschil met andere vaardigheden is wel niet gigantisch, alle vaardigheden zijn belangrijk.

Hoofdstuk zes gaat over een methode die gebruikt wordt om de prestaties van toekomstige

studenten voorspellen, voornamelijk om te voorspellen of ze zullen slagen op het introductievak tot programmeren. Hiervoor werd de data op het leerplatform Dodona geanalyseerd, en werden verschillende karakteristieken uit de data gehaald die het gedrag van de studenten samenvatten. Deze karakteristieken werden dan gebruik als invoer om een machine learning model te trainen, en dit model voorspelt dan de slaagkansen van toekomstige studenten op basis van hun submissies op het leerplatform. Maar de resultaten van deze aanpak waren teleurstellend, deze aanpak is al toegepast in een aantal studies en vergeleken met deze studies waren de resultaten erg slecht. Deze aanpak is al toegepast in verschillende contexten, maar hier waren de resultaten erg slecht. De voornaamste reden hiervoor lijkt het lagere aantal studenten te zijn. Het aantal studenten in de voorgaande studies was verschillende keren groter, waardoor zij ook veel meer data hadden om hun model te trainen. De belangrijkste conclusie hier is dat het aantal studenten in deze context te weinig is om de aanpak succesvol toe te passen.

Hoofdstuk zeven gaat over een aantal aanvullingen en de verbetering van een universitair vak dat dient als introductie tot het programmeren voor de richting informatica. De verbeteringen van dit vak zijn gebaseerd op de verzamelde literatuur. De oefeningen voor dit vak waren reeds divers, maar een ontbrak een type oefeningen dat door de literatuur als belangrijk wordt omschreven, namelijk parsons puzzels. Dit zijn puzzels waarbij regels code gegeven worden, maar in een willekeurige volgorde. Een student moet dan deze code terug in de juiste volgorde zetten. Dit is vooral nuttig voor in het begin van een vak omdat het de moeilijkheid wegneemt dat studenten anders zelf code van nul moeten schrijven. Tegelijk traint het toch algoritmisch denken en maakt het hun meer vertrouwd met code. Ook werden er een aantal worked examples toegevoegd die het proces beschrijven om een probleem van start tot finish op te lossen. Hieruit kunnen studenten leren hoe zij nieuwe problemen kunnen aanpakken op een gestructureerde manier. Als laatste is er ook een document toegevoegd dat verschillende rollen beschrijft die een variabele kan aannemen, met codevoorbeelden erbij waarin een voorbeeld zit van een variabele die op de beschreven manier gebruikt wordt.

Summary

This thesis consists of two main parts. The first part describes the key literature on the educational aspects of computer science. Chapter Two outlines the main difficulties faced by beginning computer science students. Learning to program is a very challenging task, and there are many obstacles that a beginner may encounter. Programming involves more than just learning syntax and semantics; one must also understand how to effectively use the functionality of a language. Without this knowledge, a person might understand how programs work, but writing programs themselves becomes very difficult. Misconceptions can arise in various areas, ranging from the way programming languages handle numbers differently to the semantics of keywords that contrast with their meanings in everyday language.

Chapter Three discusses different methods and types of exercises that can be used for formative assessment. This type of assessment aims primarily to ensure that students are learning and to check whether they are keeping up with the material, without attaching grades or strict evaluation. There are various teaching methods, and one of the most dominant trends today is active learning, in which the student is actively engaged rather than passively receiving knowledge as in traditional teaching.

Chapter Four presents different models for tackling problems that have no trivial solution. This is a creative process that is difficult for beginners. It cannot be captured by strict rules, and finding a solution to a given problem can be quite challenging, especially for novices. Fortunately, there are several methods that define distinct phases and tasks for each phase that need to be carried out in order to arrive at a solution. Learning such a structured approach enhances students' confidence and their problem-solving abilities.

That concludes the first part of this thesis. The second part concerns the research that was conducted and a contribution to a university course.

Chapter Five discusses a survey conducted among computer science students who had just started their studies. The survey inquired about their backgrounds in programming and mathematics, as well as their educational level, and examined possible correlations between these backgrounds and their scores in the first programming course. The analysis revealed that none of the background characteristics examined were statistically significant in relation to the grades achieved. However, the sample size was relatively small, making it difficult to definitively rule out any influence. Prior programming experience was particularly close to the significance threshold. The survey also asked students to assess their own competence in various domains of computer science, such as problem-solving. These self-assessments were analyzed as well, and it was found that students who rated themselves higher in these areas generally achieved higher grades. This was especially true for problem-solving, suggesting that it may be the most crucial skill for students to develop. However, the differences between skills were not dramatic—each of them plays an important role.

Chapter Six discusses a method used to predict the performance of future students, mainly to forecast whether they will pass the introductory programming course. For this purpose, data from the learning platform Dodona was analyzed, and various characteristics were extracted to summarize student behavior. These characteristics were then used as input to train a machine

learning model, which predicts future students' chances of success based on their submissions on the platform. However, the results of this approach were disappointing. Although this method has been applied in several studies, the results in this case were significantly worse. The main reason seems to be the smaller number of students. Previous studies had much larger sample sizes, and thus more data to train their models. The key conclusion is that the number of students in this context is too low to successfully apply this approach.

Chapter Seven covers several additions and improvements to a university course that serves as an introduction to programming for computer science students. The improvements are based on the literature reviewed. The exercises in the course were already diverse, but one type of exercise described as important in the literature was missing: Parsons puzzles. These are puzzles in which lines of code are presented in a random order, and the student must rearrange them into the correct sequence. This is especially useful at the beginning of a course because it removes the challenge of writing code from scratch, while still training algorithmic thinking and increasing familiarity with code. In addition, several worked examples were added that describe the process of solving a problem from start to finish. These allow students to learn how to approach new problems in a structured manner. Lastly, a document was added that describes different roles a variable can take on, along with code examples that illustrate how variables are used in those ways.

Contents

1 Introduction **9**

I Literature survey **11**

2 Struggles of students **12**
2.1 Introduction . . . 12
2.2 Misconceptions . . . 12
2.2.1 Cognitive load theory . . . 13
2.3 Difficulties of novices . . . 14
2.3.1 Learning barriers . . . 14
2.3.2 Types of errors . . . 15
2.3.3 Strategic knowledge . . . 16
2.3.4 Notional machine and mental model . . . 17

3 Assessment and teaching **18**
3.1 Introduction . . . 18
3.2 Program comprehension teaching methods . . . 18
3.3 Teaching strategies . . . 22
3.3.1 Active learning . . . 22
3.3.2 The PRIMM method . . . 22
3.4 Formative Assessment . . . 23
3.4.1 Common assessment methods . . . 23
3.4.2 Social assessment . . . 25
3.4.3 Tracing assessment . . . 25
3.4.4 Code assessment . . . 26
3.4.5 Assessment using diagrams . . . 27
3.4.6 Matching problems . . . 28
3.4.7 Higher order thinking . . . 28
3.4.8 Worked examples . . . 28
3.4.9 Conclusion . . . 29
3.4.10 Discussion . . . 29

4 Problem solving **30**
4.1 Introduction . . . 30
4.2 Problem solving model and application . . . 30
4.2.1 Problem solving model . . . 30
4.2.2 Problem solving study . . . 31
4.3 Problem solving heuristics . . . 34
4.3.1 Stepwise refinement . . . 34
4.3.2 Different roles of variables in novice programs . . . 35
4.4 Problem solving course . . . 36

II Research and personal contribution **37**

5 Survey **38**
5.1 Introduction . . . 38
5.2 Survey . . . 38
5.2.1 Subject context . . . 38
5.2.2 Survey contents . . . 39
5.2.3 Results . . . 40
5.2.4 Interpretation . . . 44
5.3 Limitations . . . 51
5.4 Conclusions . . . 52

6 Educational data analysis **53**
6.1 Introduction . . . 53
6.2 Analysing data of Dodona . . . 53
6.2.1 Dodona . . . 53
6.2.2 Classification . . . 53
6.2.3 Accuracy metrics . . . 54
6.2.4 Approach to analyse the data . . . 55
6.3 Context . . . 57
6.4 Results . . . 57
6.4.1 Course A . . . 57
6.4.2 Course B . . . 58
6.4.3 Interpretation of the coefficients . . . 61
6.4.4 Predicting future years . . . 68
6.5 Discussion . . . 69

7 Course additions **71**
7.1 Introduction . . . 71
7.2 Course setup . . . 71
7.3 Parsons puzzles . . . 72
7.4 Worked examples . . . 73
7.4.1 Trilateration . . . 74
7.4.2 Sum of squares . . . 74
7.4.3 Game of life . . . 75
7.4.4 Intersecting rectangles . . . 76
7.5 Roles of variables . . . 79

8 Conclusions **81**

Chapter 1

Introduction

Learning to program is an extremely challenging task for beginning students. There are many different aspects of programming in which students can face difficulties [MRF19a]. Small misconceptions in the understanding of programming concepts can cause bugs that are very hard to debug due to an incorrect understanding of programming. These misconceptions are far from the only hindrance students face when learning to program.

Students can make errors at different levels [QL18]. Misconceptions happen at the syntax and semantic levels of knowledge and consist of incorrect information. But programming is more than knowledge alone; it is an active skill that not only depends on static knowledge but also on the ability to write code and develop programs that can solve a given problem. The skillset that allows a programmer to tackle a new problem and develop a program that solves this problem is essential, but is far from trivial for a beginning student [Sol86]. Novices, in particular, lack experience, while experts can rely on problems they have previously encountered and reuse solution strategies that they have used before.

Perhaps because of the high initial difficulty, the computer science major has the highest dropout rate of any major [Win25]. Around ten percent of all students who attempt the major drop out eventually. In comparison, the other four majors with the highest dropout rate for a major are around seven percent. So, computer science has a large number of students who do not complete the major.

This is not without importance, as our society becomes more and more integrated with technology. The role of programmer remains an important job, with 1,897,100 software-related jobs existing in the United States in 2023, and the projection for 2033 estimates that this number will increase to 2,225,000 [25]. In order to prepare enough programmers, educational institutions need to reevaluate their approach to teaching computer science. It is important to minimize the dropout rate and educate as many programmers as possible.

This thesis aims to investigate the computer science education landscape to discover the best practices for teaching computer science. The research questions for this thesis are the following:

- What difficulties do students face when learning to code?
- What are the obstacles that prevent students from starting an exercise?
- How does one go from a problem statement to a working program? Is there a generic approach that can be followed?
- What does the literature surrounding computer science education recommend about best practices to teach programming?
- How can educational data be used to support the learning process of students?

These research questions will be answered in the following sections. This thesis consists of two major parts. The first part is an analysis of the literature surrounding computer science education. Chapter Two, "the struggles of students", discusses the various difficulties that beginning students face when getting familiar with computer science and learning to program. These difficulties are numerous and can be tricky to overcome. Chapter Three, ´´Assessment and teaching strategies", discusses numerous formative assessment methods and teaching strategies. Formative assessment serves as the backbone of a curriculum, testing if students have acquired the skills and knowledge that are expected of them. Chapter Four, "Problem solving" discusses problem-solving in detail. Problem-solving is one of the most difficult skills to learn for beginning programmers, yet it is one of the most crucial skills. Without proper problem-solving skills, tackling new problems is extremely difficult, and the solving process will consist of trial & error and guessing.

The second part of this thesis is about research done and the contributions to a course. Chapter Five, "Survey", discusses a survey that was administered to beginning computer science students. Several aspects of their background that could potentially have an influence on their marks were analyzed, and they were asked to self-assess their proficiency in multiple domains of programming. This self-assessment was then correlated with their marks to determine the most important skill to acquire for beginning students. Chapter Six, "Educational data analysis", discusses an approach that applies machine learning to predict the performance of future students. Several features to be used as input for the machine learning model are computed from the submissions of exercises, and the data from one or more years is used to predict the performance of future years. Chapter Seven, "Course additions", describes the improvement of a course by adding Parsons puzzles and worked examples to the course. In addition, a document describing the different roles a variable can take is appended to the course as well.

Part I

Literature survey

Chapter 2

Struggles of students

2.1 Introduction

This chapter provides an introduction to the literature surrounding the problems students face when learning to program. The different sources of misconceptions and areas of programming where they are most prominent will be discussed, as well as possible solutions for students to overcome misconceptions.

Apart from misconceptions, there exist several other difficulties students are likely to encounter. Technical knowledge, like syntax and semantics, has to be learned, but strategic knowledge also needs to be acquired, although it is much more difficult to teach. Problem-solving skills, starting an exercise, and writing the first code to solve a problem are all common difficulties.

Finally, the differences between novices and experts are outlined. The major advantages experts have are experience with many different types of problems, the ability to group lines of code together based on their function, the ability to reason at a higher level, and the ability to recognize a familiar problem statement where a known code pattern can be applied.

2.2 Misconceptions

A misconception is an incorrect understanding or belief about a specific concept or topic. Research indicates that physics teachers adept at identifying common student misconceptions are generally more successful in facilitating effective learning than those who cannot [Wil18].

In programming education, misconceptions about programming languages can complicate the debugging process and lead to frustration among students. These misunderstandings often arise from various factors, including prior knowledge or the high cognitive load arising from working simultaneously with multiple new concepts. The works [Sen+23], [Wil18], and [QL18] identify several key areas where common programming misconceptions tend to arise.

- Numerical Operations: Computers process numbers and mathematical operations differently than humans. For instance, a beginner might expect that 99/100 would yield 0.99 but not understand that certain programming languages might treat this as an integer division, resulting in 0.
- Keyword Semantics: Programming keywords often have meanings that differ significantly from their interpretations in natural language. A common misconception is that the condition in a while loop is continuously re-evaluated, as students might assume this based on the everyday meaning of the word “while.“
- Analogies as Teaching Tools: Although analogies can aid in teaching by helping students draw parallels, they can also lead to misunderstandings when the analogy does not fully

align with the concept it aims to represent. Because of this, it is important to clarify the discrepancies between analogies and the concepts that they parallel. The box analogy is often used to illustrate variables, as both can store items. In this analogy, assigning the value of a variable to another variable resembles transferring items between boxes. However, unlike boxes, the original variable retains its value after the assignment.

- Syntax and Structure Confusion: Students may confuse the purpose and structure of certain program statements. For example, a common programming statement such as `a = a + 1` is frequently misinterpreted because it appears very often and can be viewed as an atomic statement because of this.
- Overestimating Computer Capabilities (The "Superbug" Misconception): Many students assume that computers can infer their intentions and automatically adapt to their needs. This overestimation leads to misunderstandings about a computer's actual capabilities.
- Abstract Concepts in Programming: Certain abstract programming concepts, such as memory allocation, reference management, and control flow, can be challenging to grasp because they are not directly visible in the code.

Misconceptions are more likely to happen when the cognitive load on the students is high [QL18]. Novices are unfamiliar with programming and are introduced to many new concepts. They have to learn the syntax and semantics of a programming language as well as develop the skill set for writing programs. Complex tasks can add to the cognitive load, making it more difficult for students.

2.2.1 Cognitive load theory

Cognitive load theory states that there are multiple types of cognitive load [Wil18]. Intrinsic load is the information people have to keep in mind to learn. In the context of programming, an example of this is learning what a variable is. Intrinsic load is critical to the learning process and cannot be reduced unless the amount of content being taught is reduced. Germane load is the mental effort required to link new information to already existing information. This is a desirable mental effort, just like cognitive load, as it forms relations between concepts. An example of this is remembering that a loop variable is assigned a new value each time the loop executes. This forms a connection between the loop variable and the loop itself.

All other sources of cognitive load are called extraneous load. These sources do not aid the learning process and only distract from knowledge being absorbed and connections being formed. An example of this is a flashy background in a presentation that distracts from the content being taught.

Cognitive load theory states that we have a fixed amount of memory that can be split across these different loads. The goal of instructors is to reduce the amount of extraneous load as much as possible so students can spend the maximum amount on intrinsic and germane load.

The book [Sen+23] also highlights several underlying causes of these misconceptions. In the absence of an explicit execution model, students may develop their own implicit models, which are often inaccurate. Introducing a clear model of program execution can help mitigate these issues. Additionally, presenting only a narrow range of examples may reinforce faulty patterns; providing a wider variety of examples can reduce the likelihood of these misinterpretations. Misconceptions can vary in their impact and be difficult to correct [Sen+23]. Generally, they fall into three main categories:

- Factual Misconceptions: These involve incorrect knowledge that can usually be corrected by providing accurate information.
- Broken Model Misconceptions: These involve flawed mental models of concepts and their relationships. They are harder to correct and may require students to reason through examples that expose contradictions within their current understanding.

- Deep-Rooted Beliefs: Some misconceptions are tied to fundamental beliefs, often influenced by cultural or social identity. These are also “broken models“ but are particularly resistant to correction, as they can conflict with the student's worldview.

There are multiple ways to deal with misconceptions in students. Exercises that can serve as formative assessment and be used to overcome misconceptions will be presented in Chapter 3.

Students tend to learn a lot from examples. Showing sufficiently different example programs to students can prevent incorrect patterns from being detected by students [Wil18].

2.3 Difficulties of novices

2.3.1 Learning barriers

The paper [KMA04] identifies six learning barriers for students when working with programming languages. They identified these learning barriers from a study in the context of an introductory programming course for Visual Basic.NET. They researched the problems students faced when learning to program and determined overarching themes in their issues. Six learning barriers were identified this way.

In this context, a learning barrier is defined as any significant difficulty encountered when writing programs for a novice programmer. In order to overcome such barriers, a simplifying assumption is made that removes the barrier. Progress is made if the simplifying assumption is correct. However, errors are very likely to occur if the simplifying assumption is incorrect. Barriers can be insurmountable, which in this context means that the barrier could not be overcome despite considerable effort. Next, we present different types of barriers:

- Design barrier: an inherent cognitive difficulty is designing the program and specifying what the program needs to do. This is completely separate from the underlying implementation and is more of a barrier on the conceptual level. An example of a design barrier is not knowing how to create an algorithm that solves a certain problem. Writing a working program is next to impossible without a notion of this algorithm.

- Selection barrier: the difficulty of selecting appropriate programming constructs to implement an algorithm. Contrary to design barriers, this barrier is related to knowledge of the programming language and its constructs. An example of a selection barrier is knowing what algorithm to use but not knowing that lists are needed to implement this algorithm.

- Use barrier: these barriers are relevant when it's known that a certain construct must be utilized, but not known in what way. This can manifest in different ways: it can be unclear in what ways the construct can be used, unclear how to use it, or unclear what effect the use will have. An example of a use barrier is knowing that a list must be used to implement an algorithm, but not knowing how the list can be used to achieve the desired behavior.

- Coordination barrier: Complex behavior can be achieved by combining a multitude of programming constructs. Coordinating and making several programming elements work together is no easy task for a novice programmer. For a coordination barrier to exist, a student must generally know what constructs they want to use, but not how they can interact with each other in order to execute an algorithm. An example of a combination barrier is knowing that you must use a for loop and a condition to check if a list contains a certain element, but not understanding how to make the condition and for loop work together.

- Understanding barrier: This barrier is related to understanding and interpreting a program's output, particularly in detecting bugs. The majority of the understanding barriers were insurmountable. Compile-time errors were insurmountable when learners could not

identify which parts of their code were deemed right or wrong by their compiler. Runtime errors and other unexpected behavior presented insurmountable barriers when they obscured what did or did not happen at runtime. An example of an understanding barrier is getting a runtime error and not being able to locate the source of the bug based on this error.

- Information barrier: This barrier is also related to debugging programs but concerns the internal state of a program. Reading out the internal value of a variable or the stack trace of function calls is an example of the internal state of a program. Information barriers arise when learners have a hypothesis about their program but don't have the means to test it because they have difficulty reading the internal state. Without being able to confirm or reject a theory that explains the way their program behaves, students can start guessing which parts are responsible, which can potentially lead to inducing errors. An example of an information barrier is being unable to use a debugger to check the internal state of a program at a certain point in execution.

The six barriers aren't isolated from each other. An invalid assumption to overcome one barrier can lead to the next barrier. The paper [KMA04] identifies the common transitions between different types of barriers. A transition graph summarizing the results of their research is shown in Figure 2.1. Note that the edges don't add to 100% because edges smaller than 10% were removed from the graph, as well as invalid assumptions leading to insurmountable barriers.

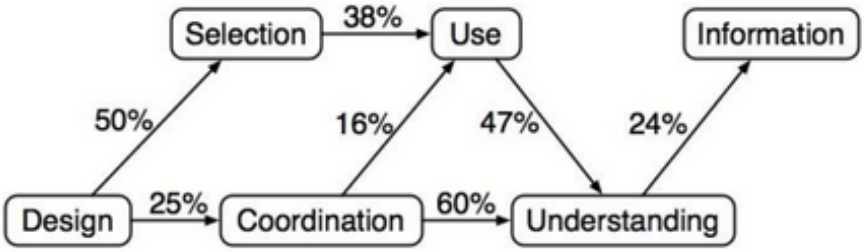

Figure 2.1: The progression of learning barriers
Source: [KMA04]

These barriers are mostly relevant to programming novices, but some can apply to experts as well. Experts usually have less trouble with selection, coordination, and use barriers because of their experience with writing programs. Understanding and information barriers arise for experts as well as novices.

These six barriers don't necessarily encompass all possible types of barriers students can face. They serve as an indication of the most prominent difficulties students encounter while learning to program. It is possible that more types of barriers exist than the ones known thus far.

2.3.2 Types of errors

There are different types of errors that students can make [QL18]. The most common type of error is syntactic errors by far. Examples of these errors are mismatched parentheses, missing semicolons, and using the assignment operator `=` instead of the equality operator `==` (e.g. using `if (a = b)` instead of `if (a == b)`). Fortunately, this type of error is easy to fix as modern compilers and integrated development environments (IDES) highlight these errors.

A more severe type of error is a conceptual error. An alternative name for this type of error is a misconception. These errors aren't related to the code produced by a student, but are related to the student's mental model of execution and computer knowledge. A conceptual error occurs when a student's acquired knowledge of a concept does not match reality. For

```
initialize sum at 0
initialize counter at 0
for element in list
    add element to sum
    increment counter
if counter not 0
    output running total divided by counter
else
    output 0
```

Figure 2.2: An example of a plan that computes the average of a list.

example, students may fail to understand that variables can only hold one value at a time. There are many areas of programming in which these conceptual errors can arise. A more detailed discussion of conceptual errors is presented in the following subsection.

The last type of difficulty students can have is related to the strategic knowledge of students. Strategic knowledge refers to expert-level programming knowledge and the skills required to plan, create, and debug a program using syntactical and conceptual knowledge. Novice students lack strategic knowledge and have a very hard time writing programs[QL18][RRa03][Sol86].

2.3.3 Strategic knowledge

Programming experts are usually not aware of the knowledge and strategies they employ when solving a problem. They usually describe the source of their inspiration and the know-how of when to apply a certain strategy or pattern as intuition or gut feeling. This type of knowledge is usually referred to as "tacit knowledge"[Sol86]. Only experts possess this kind of knowledge, and it is something novices lack. Another name for this type of knowledge is "Strategic knowledge"[QL18]. Strategic knowledge is a skillset combining planning, writing, and debugging programs using the acquired syntactic and semantic knowledge. Because strategic knowledge relies on syntactic and semantic knowledge, if this foundation is incomplete or insufficient, strategic knowledge cannot be applied. Students must first master these two types of knowledge before they can begin to build strategic knowledge.

Strategic knowledge can be applied to any type of problem and allows an expert to see patterns previously encountered, even in completely new problems. Expert programmers are able to chunk information together in a meaningful way. This allows them to reason about multiple lines of code as one block, for example, and provides structure and overview. Experts possess a multitude of these chunks that contain a pattern for a solution. These types of chunks are called "plans", and each plan achieves a certain "goal" [Sol86]. A plan consists of a high-level overview of multiple steps that are required to achieve a goal. It is important to note that this goal doesn't have to be the complete solution to a problem. A plan can also be used to achieve a subgoal for the problem. An example of a plan is shown in Figure 2.2. The goal of this plan is to compute the average for a list.

Notice that this plan is language independent and does not specify implementation details. An expert programmer can instantiate this plan in different programming languages. Expert programmers have a multitude of plans at their disposal, and the ability to recognize (sub)goals where a known plan can be used to achieve that (sub)goal. The knowledge of these plans and the ability to apply them in novel contexts is referred to as strategic knowledge as a whole.

Strategic knowledge is something novices severely lack. They do not have the experience acquired by solving many problems, and as such, they do not possess many plans that can be utilized to solve problems. Novices have a difficult time chunking information together like experts do. They often get lost in the individual statements without seeing the global forest that is the program [Sol86]. This makes planning, writing, and debugging programs particu-

larly challenging for novices. Even if they possess the syntactic and semantic knowledge that is required to solve a certain problem, actually writing the program and knowing the algorithm to use is another task on its own. Novices have difficulty understanding the task given to them, and decomposing the problem is yet another challenging task for them.

Fortunately, there are some heuristics and problem-solving models that can serve as a foundation for novices when attempting to solve a novel, non-trivial problem. These techniques are presented in Chapter 4 in the context of problem solving.

2.3.4 Notional machine and mental model

A mental model is an internal, conceptual model that a student has of a concept. They are particularly relevant for programming, where the code itself is tangible but execution of code is not visible. Mental models are required to understand the behavior of the code. A mental model consists of two pieces: its structure (the parts) and its behavior (the state changes). Students have more trouble with the state changes [MP24]. The parts of a program are usually directly visible in the code, but the behavior is much more difficult to see. An adequate mental model of programs is needed to understand and predict the behavior of code.

Such a mental model of the behavior of code is called the notional machine [RRa03] [MP24]. This model isn't necessarily true to reality. Few mental models will correctly view the execution of code in its entirety, including the physical memory instructions in the hardware. A notional machine is an abstract of the computers behavior, and only the relevant details to understand the behavior need to be included in this model. The book [Sen+23] recommends that learners be taught an explicit model of the notional machine, otherwise they will construct their own model of execution, which is likely to contain errors.

Chapter 3

Assessment and teaching

3.1 Introduction

Assessment refers to the process of evaluating or measuring an individual's skills, abilities, knowledge, or performance. There are two types of assessment: formative assessment and summative assessment. The primary purpose of formative assessment is to monitor the learner's progress, provide constructive feedback, and enhance their understanding of the subject being taught. Unlike summative assessments, formative assessments do not involve pass-or-fail outcomes. For maximum effectiveness, these assessments should be conducted swiftly to minimize disruption to the lesson flow. They should also yield clear, actionable results. An essential objective of formative assessment in programming education is to identify and address students' misconceptions, particularly those related to the challenges they face when learning to program. In contrast, summative assessment evaluates students' final performance, typically assigning grades or marks. The goal is to determine whether students have achieved the learning objectives outlined in the curriculum.

There exist many types of formative assessment activities. Some assessment activities that reinforce program comprehension skills are presented. These activities are shown in the context of the block model, which serves as a classification mechanism for the different types of program comprehension that exist. The block model's primary goal is to serve as the foundation on which course material can be built.

Although formative assessment is very important, it doesn't replace good teaching strategies. The teaching strategy using the PRIMM method is discussed, and the paradigm of active learning, supported by the constructivist learning theory, is also discussed. The active learning paradigm states that students don't learn efficiently from regular lectures and that instead, students must be immersed in the course material and actively work with it instead of passively absorbing the content of a lecture.

Finally, different types of exercises that serve as formative assessment methods are presented. Each type of exercise has its own strengths and weaknesses, and must be appropriately combined to create a balanced curriculum that strengthens the students' proficiency in the different aspects of programming.

3.2 Program comprehension teaching methods

Program comprehension is an important part of computer science. Program comprehension is usually conceptualized as a process in which an individual constructs their mental model of a program [Izu+19]. Without proper program comprehension, several tasks, such as extending a program, debugging, and rewriting code, cannot be adequately accomplished and are reduced

	Architecture/structure		Intention
	Text surface	Program execution	Relevance/intention
Macro structures	Understanding the overall structure of the program text	Understanding the algorithm underlying a program	Understanding the goal/purpose of the program in the current context
Relationships	Relationships between blocks	Sequence of function calls, object sequence diagrams	Understanding how subgoals are related to goals
Blocks	Regions of interest that build a unit (syntactically or semantically)	Operation of a block or a function	Understanding the function of a block of code
Atoms	Language elements	Operation of a statement	Function of a statement

Figure 3.1: The block model consisting of three dimensions and four hierarchical levels through which code comprehension can be looked at.
Source: [Sch08]

to simple trial and error. There is also a strong motivation present to teach students code comprehension skills [Sch08]. Reading code is an important skill because much of preexisting knowledge, such as standardized solutions to programming problems and good programming styles, can be captured by programs, and learning to read those programs helps to teach this knowledge. Writing code also involves reading code. For example, looking back upon your code after a break or not touching it for a while requires you to read the code to get familiar with it again. The block model [Sch08] serves as a support for program comprehension activities and the different categories of activities. The block model is presented in Figure 3.2. The slightly edited version by [Sen+23] of the block model is used here.

The block model highlights code comprehension at four different hierarchical levels. An atom is an individual statement given by a line of code. Blocks are made out of multiple atoms/lines of code and are grouped together because they work together towards a common goal or because the lines of code are similar. Relationships exist between different blocks of code. If a block of code is a function, that function may be called by other blocks of code. Finally, the macro structure is at the top level of the hierarchy. It defines global cohesion, why relationships between blocks are necessary, the program's goal, and the algorithm used to achieve that goal.

These four hierarchical levels can be looked at from different dimensions. The first dimension only looks at the textual component, how the program is structured, and the different syntactical components that make up the program. The translation from text to program execution at runtime is the focus of the second dimension, the different sequences of function calls, and the operation of the different lines of code. These two dimensions focus on the architecture and structure of the program. Opposed to this, looking at the intention and relevance of the program is the final dimension. The goal of blocks of code is looked at rather than the structure or implementation of the blocks. These three dimensions are ordered in levels of abstraction. Only looking at the text scratches just the surface of the program, while reasoning about the goal of the program and the different sub-goals is the highest level of abstraction. The text surface and program execution are concerned with the structure of a program and the execution through which it achieves its goal, while the intention specifies what this goal consists of.

Extracting knowledge from text can be done at two distinct levels, denotation and connotation [Izu+19]. For example, when looking at the statement `i = i + 1`, one can look at the individual components of this expression in isolation. The variable `i` is assigned a new value. This new value comprises the sum of the variable `i` and the literal `1`. Denotation refers to the element's meaning in a context-free sense, like looking up a word in a dictionary. Its literal meaning is

clear and defined, but the relevance in the sentence is ignored. Likewise, the statement's goal is not touched upon when only looking at the denotation. In contrast to denotation, connotation concerns the meaning in the concrete context in which it is used. With regard to the variable `i`, this can be to understand its role as a stepper. The program comprehension tasks that align with the program's textual dimension focus on the program's denotational meaning. Understanding the execution of the program is done using the notional machine, which introduces connotation as well. The execution of a program depends on the context established by additional elements. The new value of the variable `i` depends on its old value. These two levels of meaning are related to the program itself.

For the third dimension, relevance/intention, the program has to be linked with an external concept. For example, if the variable `t` is interpreted as a temperature measure in Celsius, the statement `1.8 * t + 32` converts this interpretation from Celsius to Fahrenheit. By only considering the program itself, this meaning cannot be uncovered. The external concepts of temperature, Celsius, and Fahrenheit must be considered to link the statement with its purpose. The external meaning is referred to as goals, and the program's method of achieving these goals is referred to as plans by [Sol86]. The literature concerning this topic sometimes uses different but equivalent terms such as "Structure" and "Function".

Each individual block of the block model combines one of the four hierarchical levels with one of the three dimensions. For example, the block at the level of relationships and through the dimension of the program execution is concerned with how functions relate to each other, the sequence in which they call each other, and the different objects present in the code and their relationships, like composition and inheritance.

The purpose of the block model is to provide an overview of the different areas concerning program comprehension and aid computer science teachers in teaching students to understand programs. It does not provide a path or the best way to begin this process. It is up to the teacher to choose which blocks to tackle and the order in which they are tackled. Not all blocks have to always be taken into account. When novices start learning computer science and are introduced to program comprehension, they usually start at the bottom of the block model with the most fundamental component, the atoms [Sch08]. They gradually work their way up to the top parts of the block model. This can be done across multiple dimensions simultaneously or while only looking at one dimension.

Building further upon the block table, the paper [Izu+19] identifies several program comprehension activities for each block of the block model. A slightly edited version by [Sen+23] of these activities is shown in Figure 3.2.

These code comprehension activities are concrete implementations of the intentions behind the block model. For example, for the dimension "relevance/intention" at the block level of code, a possible activity is summarizing the goal of the block of code in a short sentence. Not all tasks for the block model identified by [Izu+19] are shown here due to space considerations. Interested readers are encouraged to read the original working group report.

The text surface tasks focus on the perceivable appearance of the program. The comprehension process starts with reading and discerning between elements. The tasks are focused on statically determinable properties, such as syntax and static typing of variables.

The program execution tasks are all about the translation of the text to an executable program, which is sometimes referred to as the operational semantics. The information provided by the program text is insufficient to understand the program execution and must be supplemented with a concept of machine state, providing the context in which the program executes. A notion of machine state, establishing the context in which the program runs, is necessary. The construct of a mental model of a notional machine, an abstraction of a computer's behavior, is at the core of this dimension.

Finally, while the previous 2 categories of tasks focus on the program itself, the function or purpose tasks introduce properties extrinsic to the program. Separating functions related to

	Architecture/structure		Intention
	Text surface	Program execution	Relevance/intention
Macro structures	• Describe the overall program block structure by drawing nested boxes • Represent the overall program structure by drawing a tree of function/procedure dependencies	• Verify if a program statement or block is ever reachable during program execution • Identify a comprehensive set of inputs to check all possible computation flows of a program	• Choose an appropriate name for a program • Create meaningful test cases for the allowed inputs and expected outputs
Relationships	• Link each occurrence of a variable with its declaration • Identify where a particular function is called	• Trace the program execution for a given input, where the program includes calls to procedural units • Verify whether some branches of a switch/case statement are redundant, that is, can never be executed • Identify the scope of a variable	• Choose an appropriate name for a variable • Solve a Parson's puzzle for a given code purpose by reordering simple blocks
Blocks	• Draw a box around the code of each conditional construct • Draw a box round the body of each method/ procedure/function	• Identify recurring instrumental blocks such as that for swapping the values of two variables • Identify the block(s) implementing some specific program pattern	• Summarize in a short sentence what the block goal is • Identify the program block(s) with a given function, described in problem-domain terms
Atoms	• List all integer variables • Draw a box around the headers of all procedures/functions in a piece of code	• Determine the program output for given input data, again where the program does not include procedural units • Determine the value of an expression for given values of the involved variables	• Identify the purpose of an expression or a simple statement, in connection with the problem domain • Rename a constant with an appropriate name from the problem

Figure 3.2: Several code comprehension activities mapped to each block of the block model

Source: [Izu+19]

code execution and functions related to the purpose is not always straightforward.

3.3 Teaching strategies

3.3.1 Active learning

Active learning[BF21] is a teaching strategy that focuses on the way students gain knowledge. The active learning paradigm states that students construct knowledge based on personal experiences and existing knowledge. This is a stark contrast with the way normal lectures are performed, in which a student is expected to listen and passively absorb information.

Some of the most popular active learning instruction methods are flipped classroom, project-based learning, and peer instruction. Peer instruction will be discussed in more detail in section 3.4.2. With the flipped classroom method of teaching, students are expected to study the content ahead of time before the actual lecture. The time that would be used for a lecture can then be used for various other activities that involve the students more than a passive lecture.

Project-based learning is a method of teaching that involves projects the students have to work on, which involves them more and gives them the space required to get familiar with the coding concepts learned. By engaging the students with authentic, real-world problems, they also become more familiar with the development process behind projects.

Active learning is strongly related to the constructivist theory [Ben98a]. The constructivist theory is a theory on how knowledge is acquired that states that knowledge isn't passively gathered, but actively constructed based on a student's previous knowledge. Knowledge cannot exist on its own and is always part of the knowledge structure present.

3.3.2 The PRIMM method

There are different approaches to tackling the problem of teaching students to program. Program comprehension tasks are easier than program writing tasks and are often used as a starting point before students are tasked with writing code of their own. A structured approach for teaching programming is the PRIMM (Predict, Run, Investigate, Modify, Make) method [SWK19].

- Predict: students are given a program and are tasked with discussing and predicting the behavior as well as the output of said program. The prediction can be made individually or in groups of two. It is important that the code is only observed and not executed. The teacher discusses the different answers together with the class.
- Run: students are given an executable version of the program to confirm or deny their predictions, after which the results are discussed with the class.
- Investigate: a range of activities relating to the code are provided, such as tracing, explaining, annotating pieces of code, debugging, and so on. The block model presented in Figure 2.X can serve as a map for these activities and their goal.
- Modify: a series of challenges with increasing difficulty, such as adding functionality to or rewriting parts of the given code, are presented to the students. The program will gradually change, and ownership of the program will shift towards the students. They were first presented with unfamiliar code that belonged to someone else, but eventually, they gained partial ownership of the code as it got extended with new functionality.
- Make: the students are tasked with a new programming problem where they can practice the skills and knowledge acquired by performing the previous steps.

Students gradually progress through each of these stages and start by getting more familiar with the code presented to them. They eventually move on to editing the code, which practices code

writing skills without having to start writing a program from scratch. The final step is for the students to combine the skills and knowledge they learned and write a new program based on a problem statement. The major advantage of the PRIMM method is that it provides a structured approach to supporting the teacher in assisting students with their learning process.

The PRIMM method was evaluated in the context of several schools with students aged 12-14. A baseline test and a posttest were taken, as well as control groups at several of the schools. The results were analysed, and the marks of the experimental group on the posttest were higher in a statistically significant way.

Another teaching method similar to the PRIMM approach is the use-modify-create method [Lee+11].

- Use: At first, students are mere users/consumers of the creation of someone else. They use preexisting code someone else made without changing anything about it at first. They are mere observers of the code.
- Modify: Over time, students make gradual changes to the code. This can start with very simple changes, such as formatting output in a different way or changing the color of an object in a visualization. Students eventually move towards more complex changes in the code, such as altering the way the program behaves and adding more functionality to the code. Successfully modifying the code in this way requires a deeper understanding of the code and the underlying concepts. The code was previously someone else's code, but as the changes increase, the ownership partially shifts towards the student.
- Create: During the modify stage, students may gain inspiration for a new program. This program is one they have to create from scratch, using the skills and knowledge learned by modifying the code.

Use-modify-create defines the progression of students with three stages of engagement. It is based on the premise that scaffolding increasingly deep interaction will help acquire and develop computational thinking skills in students. The progression between these stages isn't always linear. There is no clean break point between modifying and creating, and students can transition between the stages.

It's important that the progression of students maintains a level of challenge at all points without becoming overwhelming for students. Increasingly more difficult tasks should be presented as their skills and knowledge increase, to continuously challenge the students. Tasks that seem daunting at one point become achievable later with the appropriate and incrementally challenging experiences. Without a sufficient level of difficulty, the students will eventually lose interest and get bored with the tasks as their growing skills refrain from being challenged. On the other hand, a too steep learning curve with the difficulty of the tasks beyond the capabilities of the students will not allow for good progression and can induce anxiety in the students.

3.4 Formative Assessment

3.4.1 Common assessment methods

The works [Wil18] and [Sen+23] discuss various options to perform formative assessment.

Asking multiple choice questions (MCQs) [Wil18][Sen+23] is a widely used method of formative assessment, particularly effective when the answer options are designed to target specific misconceptions. These misleading but plausible options, known as plausible distractors, help to uncover misunderstandings, giving MCQs diagnostic value.

Administering MCQs in class allows instructors to gauge students' comprehension and identify the prevalence of misconceptions. Tools such as online response systems, like Kahoot, enhance the process by providing instant grading and displaying answer distributions, which further inform the instructor about student understanding. An example of an MCQ is presented in

What is 37 + 15?
A) 52
B) 42
C) 412
D) 43

Figure 3.3: An example of a Multiple choice question for an addition.
Source: [Wil18]

total = 0
if v > 0
total += v
for v in values

Figure 3.4: An example of a Parsons problem.
Source: [Wil18]

Figure 3.3. The correct answer is, of course, 52, but each possible answer can indicate the source of the misconception. If a student chooses 42, they are performing the addition correctly but are neglecting the carry. If the answer is 412, they are treating each column of numbers as a separate problem unrelated to the neighboring numbers. Finally, if they choose 43, they know they must carry a number but are carrying it into the wrong column. Based on the answer, the appropriate misconception is identified and can be remedied.

Code and run [Wil18] is another common assessment method where the student is prompted with a problem and is expected to write a program solving this problem. The difficulty of this exercise can vary drastically depending on the needs of the instructor. If they are to be used in class, they should be brief and only have 1 or 2 plausible solutions. The larger the exercise becomes, the more important planning ahead and problem-solving techniques become. Section 4 discusses different problem-solving techniques and plans that can be followed to solve an exercise.

Programming is a crucial skill when learning computer science. Code and run exercises help students learn these skills, but they can be hard to assess. There are often multiple correct ways to write a program, which makes it difficult to check directly for code correctness. If their code is rejected because it doesn't follow the exact same structure, students can be demoralized. Assessing the output of their programs mitigates this problem, but introduces a new one: it is impossible to give feedback on the coding style based solely on the output alone.

Parsons puzzles [Wil18][Sen+23] are another formative assessment technique where students are presented with lines of code that make up a complete solution, but the lines are provided in a jumbled order. The student's task is to reconstruct the solution by arranging the lines correctly. For example, Figure 3.4 illustrates a Parsons puzzle where the goal is to sum all the positive values in an array.

For more advanced learners, Parsons puzzles can include distractor lines. These are extra lines of code that are not part of the solution. Students must identify the correct lines and arrange them appropriately, significantly increasing the difficulty of the exercise. This variation is better suited for experienced students due to its higher complexity.

Parsons puzzles offer several benefits for students. Unlike traditional coding exercises, they provide a predefined starting point, sparing students from the challenge of beginning with an empty screen and having to write code from scratch. This allows learners to concentrate on understanding and organizing the flow of control within the code rather than on syntax or implementation details.

	Unacceptable 0-3	Poor 4-6	Good 7-8	Excellent 9-10
Solution	An incomplete solution is implemented on the required platform. It does not compile and/or run.	Runs, but has logical errors. Apply poor if the program does not use a 2D array or has multiple incorrect results.	A complete solution is tested and runs but does not meet all the specifications and/or work for all test data. Apply good if program misses one data entry line.	A completed solution runs without errors. It meets all the specifications and works for all test data.

Figure 3.5: An example of a Rubric used for grading a programming assignment.
Source: [ESH16]

3.4.2 Social assessment

Assessment can also involve students evaluating either their peers or themselves, a process known as self-assessment or peer assessment [Wil18][Sen+23]. These methods have the advantage of promoting active student involvement and fostering deeper engagement with the material. However, potential disadvantages include students being overly lenient with themselves or their friends and the possibility of inaccurate evaluations. These issues become particularly problematic when such assessments contribute to grades, as the results may not reliably reflect actual performance.

An alternative to mitigate these drawbacks is calibrated peer review [Wil18]. In this approach, students first assess an example provided by the instructor, who has already supplied a benchmark evaluation. Only when the student's assessment aligns with the instructor's can they proceed to evaluate their peers. A rubric may be provided to guide the process and enhance reliability. A rubric is defined as a list of several degrees of quality and the criteria needed to meet a certain degree of quality. An example of a rubric is presented in Figure 3.5. The degrees of quality are defined as "unacceptable", "poor", "good", and "excellent". Only one row with the category "solution" is shown here, but the full rubric includes four additional rows with the categories modular program design, code readability, user interface, and turned-in. A submission is graded based on each category and the requirements the submission fulfills. The main advantage of a rubric is that the requirements are strictly defined, making grading a submission or peers more consistent.

Another collaborative assessment method is peer instruction[Wil18], which combines social interaction with conceptual learning. First, an introduction is given to a topic; this can be done with a traditional lecture or by having the students read or watch an educational video in advance. Then, an MCQ designed to expose misconceptions will be posed in class. Should the MCQ reveal that many students have varying misconceptions, they are encouraged to discuss the reasoning behind their answers in small groups.

This process requires students to externalize their thought processes, debate their interpretations, and potentially correct one another. Peer instruction is highly scalable and can be effectively used in classes of any size. To measure progress, the same MCQ can be posed again after the discussion to determine whether students' understanding of the concept has improved. Peer instruction is well-researched and has been shown to improve student motivation,

3.4.3 Tracing assessment

Tracing programs [Wil18] is a category of assessment that evaluates students' ability to follow and understand the execution of code. This skill is critical for debugging and serves as

```
A) vals = [-1, 0, 1]
B) inverse_sum = 0
    try:
        for v in vals:
C) inverse_sum += 1/v
       except:
D)         pass
```

Figure 3.6: An example of a tracing problem where the execution of a program is being traced.

Source: [Wil18]

```
left = 24
right = 6
while right:
    left, right = right, left % right
```

Figure 3.7: An example of a tracing problem where the values of the variables are being traced.

Source: [Wil18]

an indicator of students' comprehension of core programming concepts. Effective program tracing requires a solid understanding of constructs such as function calls, loops, and conditionals.

One approach involves having students label the steps in a program and provide the sequence of these labels, representing the execution trace. For example, the correct trace for the program depicted in Figure 3.6 is ABCCD. Another method focuses on tracing variables by tracking their values throughout the program's execution. Students might be tasked with recording the value of a specific variable at each line number to demonstrate their understanding of the program's behavior. Figure 3.7 shows an example of such an exercise, and Figure 3.8 shows a possible solution for this exercise.

Reverse execution[Wil18] is the opposite of traditional program tracing. Rather than starting at the beginning of a program and following its execution, the objective is to determine the input that would lead to a specific output. When the output is an error, reverse execution closely resembles debugging, as it involves identifying the conditions that cause the program to fail.

3.4.4 Code assessment

Other assessment methods exist that are similar to tracing but focus more on changing the code. Minimal fix [Wil18] is a method that strongly resembles debugging. Students are given a program containing a bug and are expected to find and remedy it while changing the program as little as possible. Figure 3.9 shows an example of such a problem. The program's goal is to determine if a given point is part of an interval defined by `[lower, higher]`. The given code attempts this but has a bug in the first condition. The condition `point <=` lower should be

left	right
24	/
24	6
6	0

Figure 3.8: A possible solution for the tracing exercise presented in Figure 3.7

```
def inside(point, lower, higher):
    if (point <= lower):
        return false
    elif (point <= higher):
        return false
    else:
        return true
```

Figure 3.9: An example of a minimal fix problem.
Source: [Wil18]

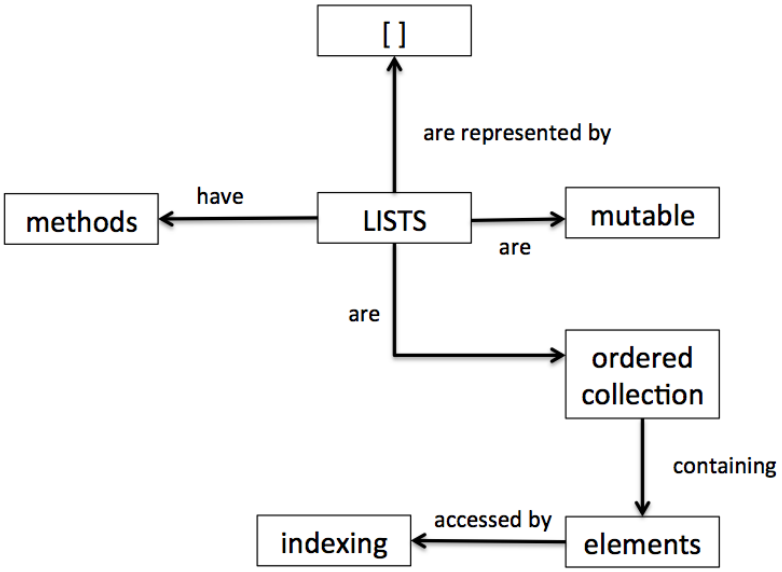

Figure 3.10: An example of a concept map describing lists in Python.
Source: http://swcarpentry.github.io/training-course/2012/10/round-2-1-concept-map-and-examples-of-lists-in-python/

`point >=` lower to determine if the point is part of the interval.

A similar method is theme and variation, which asks students to make a small change to a program that changes the program's behavior or its output. This is the opposite of refactoring exercises, where the goal is to restructure the code without changing the output. Refactoring code is a very useful real-world skill that is also suited for teaching students. Tweaking an existing program that already works is one of the fastest ways to produce a result.

3.4.5 Assessment using diagrams

Another category of assessment revolves around the use of visual aids such as diagrams. Concept maps [Wil18][Sen+23] are one such type. Figure 3.10 shows an example of a concept map Creating a concept map about a certain topic externalizes cognition and shows the internal mental model. This provides opportunities to correct this mental model should misconceptions be present. They can also be used to assess the level of understanding present regarding the visualized concept. The downside of letting students create concept maps is that they have to be manually corrected, which can take a significant amount of time.

A quicker assessment method is to let students label existing diagrams. The labels can be provided

Option	Label
?	Start of line
*	Zero or more occurrences
+	End of line
$	One or more occurrences
^	Zero or more occurrences

Figure 3.11: An example of a matching problem for regular expression operators.
Source: [Wil18]

to the students, or they can fill them in themselves. This method also reveals cognition but doesn't require human intervention to correct the diagrams, which means it scales better to larger groups. A similar option is splitting a diagram into multiple parts and having students put the diagram back together. This is similar to a Parsons problem in the sense that multiple parts have to be put back together in the right order. A smaller or larger skeleton can be provided to aid in the reconstruction of the diagram.

3.4.6 Matching problems

Matching problems [Wil18] can be thought of as a special case of labeling diagrams where the label is on one side, and the diagram, which in this case is a column of text, is on the other side. The labels have to be matched with the corresponding items. An example of such an exercise is one where the column of text is a code block, and the labels are possible outputs of the code block. One-to-one matching problems have only 1 combination of label and item, but variations exist where different matching types are possible.

Many-to-many type matching problems can have multiple matches for one label, and some items may not have corresponding matches at all. This variation is a lot more difficult compared to one-to-one matching problems. Both require the student to perform higher-order thinking, but many-to-many type matching has a higher search space. All labels remain possible options to match with, even if they have already been matched with different items. One-to-one matching allows for reducing the number of possible labels by matching the easy combinations first.

A special case of matching is ranking, where different items must be sorted based on a property. As an example, a list of algorithms can be provided, which must be sorted from fastest to slowest algorithm.

3.4.7 Higher order thinking

Most of the previous assessment methods discussed so far have focused on the technical capabilities of students. Assessment methods that train the higher-order thinking of students also exist. An example of such a method is code review [Wil18], where students are presented with code that is functionally correct, but violates programming conventions, like using inappropriate variable names, for example. It can even be graded automatically if a list of potential errors is presented and errors have to be matched with lines of code.

Another method that focuses on higher-order thinking is summarization. A summarization exercise consists of a piece of code that is presented to the students, and they are asked to describe the goal of the piece of code in a short sentence. This trains students to think on a higher level about the code they are writing, and to group multiple lines of code based on their function.

3.4.8 Worked examples

While not a typical form of assessment, worked examples [Sen+23] are examples of a completely solved exercise. Contrary to a regular solution, the full process of solving the exercise is documented, including any possible dead ends, misconceptions about the exercise, and potential bugs in the code. They serve to illustrate how similar exercises can be solved and to showcase the process that someone goes through when solving an exercise. Students can then learn from the method used to solve this exercise and apply this knowledge in later exercises.

3.4.9 Conclusion

While many types of formative assessment techniques and exercises are available, it is essential to implement them thoughtfully in a course. Selecting only a few methods ensures that students are not overwhelmed. Since the cognitive load students can handle is limited, introducing too many new assessment methods may unnecessarily increase this load, potentially hindering their ability to focus on the subject matter itself.

3.4.10 Discussion

Each of the provided assessment methods has its own advantages and disadvantages. Programming is ultimately about creating a working program, but a good development process is important to deliver quality code on time. Combining product assessment with process assessment or assessment by interview combines the evaluation of the final product with insight into the development process. Combining multiple assessment methods can help overcome some of the disadvantages that some of the assessment methods would have when used on their own.

Chapter 4

Problem solving

4.1 Introduction

Learning problem-solving skills is one of the most common difficulties beginning computer students face [MRF19a]. It is also one of the most important ones, requiring skills and knowledge. Solving an exercise becomes painstakingly difficult without the proper method to start tackling the problem in an appropriate way. The process of problem solving requires knowledge of the problem domain, proficiency in programming, and knowledge of programming concepts. Unfortunately, explicit instruction on problem-solving is not always taught. Teaching problem solving is different from teaching most programming concepts. A clear definition exists for most of these concepts, but there is no definition of problem solving, nor an exact algorithm or guide on how to do it. It is inherently a creative process, and each problem has its own quirks and defining features. But this doesn't mean two problems are completely unique. Many problems share similarities, be it in the way they are structured or in the way their solution is implemented. Being able to recognize that a new problem is similar to a previous, older problem that has already been solved allows one to take inspiration from the existing solution for that problem.

In this chapter, a generic approach to solving a problem, starting from the problem statement and ending with the solution, is presented. This solution contains 6 different stages that have to be completed. Understanding the problem, devising an algorithm to solve the problem, implementing this algorithm, testing the code produced, and delivering & documenting the final solution are all part of this process. To show that explicitly teaching a problem-solving process helps students become better problem solvers, another study is presented where a very similar problem-solving process is taught to one group of students, while another group of students is not introduced to this process.

4.2 Problem solving model and application

4.2.1 Problem solving model

The paper [DTM99] introduces a model for problem solving based on the different models presented thus far in the literature surrounding this topic. This model aggregates the important stages that are shared across multiple models, and the different tasks that are required to be completed before one can move onto the next stage. The different stages of the model are:

- Formulating the problem: The start of solving any problem is defining the problem that is to be solved and defining what it means to have solved the problem. Additionally, given information, unknowns, and possible relations can be defined between different aspects of the problem. Refining the problem statement can be done by asking questions that explore the problem in more detail, restating the problem, gathering new information, introducing notation, drawing relevant diagrams/visualizations of the problem,... Domain knowledge, problem modeling, and communication skills are required for this stage.
- Planning the solution: After exploring and defining the problem, a solution must be sought that solves this problem. This includes two important parts: Identifying possible solutions for the

problem, as well as devising a plan. Alternative solutions can be found by looking at a simplified version of the problem, restating the problem, or looking at similar, previously solved problems. Once several solutions have been considered, a suitable solution is selected and further refined. A plan must be devised by outlining the solution and breaking it into subparts. A high-level plan is sufficient in this stage to keep a general overview without getting lost in implementation details, which are premature at this stage. The plan is then split into subparts to make it more manageable. The goal is split into several subgoals, and the tasks required to achieve each subgoal are defined. Domain and problem-specific knowledge, as well as strategic knowledge, are required for this stage.

- Designing the solution: Once a high-level plan for the solution has been constructed, this solution will have to eventually be implemented. Before that, the plan will need to be refined again in order to prepare for this implementation. The high-level solution will be refined to a carefully specified solution with a sufficient level of detail. Existing subgoals, which represent various components of the solution, are reevaluated to see if further decomposition is needed. The subcomponents must be transformed into modules for which the functions are defined, the data used by the module is formally represented, and the algorithms used are explicitly stated. As for planning the solution, domain and problem-specific knowledge, and strategic knowledge are required for this stage.
- Translating the solution: After specifying the solution in full detail and preparing the solution for implementation, the solution has to be converted into code. This is the first stage that includes actual development of the program, and it is heavily reliant on the programming language of choice and other technical details. Translating the plan requires extensive knowledge of the programming language, the ability to compose programs consistent with the defined solution, and the ability to comprehend programs that can potentially be reused in this new context. This stage is usually carried out in parallel with the next stage, testing of the solution. The ability to debug programs is also an important skill to have in this context. In addition to previously required skills, this stage also requires syntactical, semantic, and pragmatic skills to translate the plan into a working program.
- Testing the solution: This stage is usually performed in parallel with the previous stage. The developed solution has to be verified. The verification procedure includes checking the agreement of the code with the specified solution in the previous stage, as well as the correctness of the code itself. The correctness of the code is typically verified by checking the accuracy of the results. Test data is first developed, after which the data is used to test the program and to evaluate the results produced by the program on this data. The verification of the correctness of the program can take place at any point during development, but comprehensive post-development testing is an essential part of this process. Apart from correctness, other criteria such as efficiency, readability, and reliability can be taken into consideration as well during the testing process. This stage requires the same skills as the translation stage.
- Delivering the solution: The final stage of the problem-solving process ends with delivering the results and producing documentation for the deliverable. This includes documenting the program, the solution strategy taken, and the results of the performed tests. This is essential to allow comprehension and/or modification of the code later on. The produced documentation doesn't have to be limited to internal documentation, such as code comments. It can also include external documentation developed prior to the code, such as the problem statement, information about the problem domain, solution planning, specifications and algorithms, and so on. Communication skills are essential to this stage to communicate all the relevant information about the production process.

An overview of the different stages, the related tasks, and the skills required for this stage is presented in Figure 4.1.

4.2.2 Problem solving study

The previous section talked about a problem-solving model that establishes the different stages of problem-solving. Explicitly making students aware of a problem-solving model increases understanding and makes them better problem-solvers[Lok+16]. This study was performed in the context of a two-week web development summer camp in which 48 high school students participated. The summer camp taught the basics of HTML, CSS, and JavaScript to the students with a focus on the React framework. The goal was for the campers to become familiar with web development and feel like they were capable

	Problem formulating stage	Solution planning stage	Solution design stage
Tasks	• Create problem description • Use inquiry questions to refine problem description • Extract facts from refined problem description	• Identify alternatives/select solution strategy • Breakdown problem into components • Organize/associate facts with components	• Refine components into sub-components, sequence and organize • Specify data and module functions • Specify algorithimic logic
Knowledge and skills	• Domain knowledge • Problem modeling skills • Communication skills	• Domain and problem knowledge • Strategic skills	• Domain and problem knowledge • Strategic skills
	Solution translation stage	Solution testing stage	Solution delivery stage
Tasks	• Compose code from algorithmic specification • Comprehend, reuse and integrate existing code • Debug code	• Develop test data • Test solution for correctness • Modify solution or prior stages	• Document solution, strategy and results Present results
Knowledge and skills	• Domain and problem knowledge • Strategic skills • Syntax, semantic and pragmatic skills	• Domain and problem knowledge • Strategic skills • Syntax, semantic and pragmatic skills	• Communications skills

Figure 4.1: A table showing the different stages of the problem-solving model, the tasks that are to be performed for each stage, and the skills required for each stage.
Source: [DTM99]

of learning more about these technologies; they need not necessarily be capable of developing a full-fledged website themselves. The students were split into two groups: one experimental group and one control group. The control group consisted of 23 students, while the experimental group consisted of 25 students. There were no significant differences between the two groups in terms of prior programming experience, gender distribution, grade levels, and class attendance rate. The camp consisted of ten 3-hour sessions distributed across two weeks. The experimental group had sessions from 9 to 1,2, and the control group had sessions from 13 to 16.

The experimental group received additional instructions on the problem-solving process. The first session contained a one-hour lecture about problem-solving. The problem-solving stages that were presented are summarized as follows:

- Reinterpret problem prompt: Programming typically starts with a description of a problem, which serves as the stepping stone for a programmer to begin the problem-solving process. This statement must be understood, interpreted, and possibly clarified by the programmer.
- Search for analogous problems: Programmers can draw upon related problems they have previously solved to gain inspiration for a solution strategy. Related problems may have similar solutions, which simplifies the task of finding a solution.
- Search for solutions: With some understanding of a problem, programmers seek solutions that will solve the problem. These solutions can appear from previously solved problems, be created by adapting a known solution to fit the problem, or be born from a creative approach to a problem.
- Evaluate a potential solution: With a potential solution for the problem in mind, this potential solution must be evaluated. This evaluation is concerned with the suitability of this solution for the problem, as well as practical details such as implementation. Typically, a prototype or mental model of how the solution will function is created in this stage.
- Implement a solution: When an acceptable solution has been found, this solution must be converted into a working program that solves the problem.
- Evaluate implemented solution: After implementing the solution, the solution must be evaluated to see how well the solution solves the problem. This stage involves testing and debugging the code, as well as rewriting the code to fix any bugs found.

Note that this problem-solving process is slightly different from the process described in the previous subsection. The solution delivery stage has been removed, and the solution planning stage has been spread across the “Search for analogous problems“ stage and the “Search for solutions“ stage. The first stages, “Reinterpret problem prompt“ and “Problem formulating stage“, have different names but serve the same function. Both intend to explore the problem and gain familiarity with the problem domain. The stages “Evaluate a potential solution“ and “Solution design stage“ serve the same purpose as well, just like the stages “Implement a solution“ and “Translating the solution“, which both have the goal of converting the chosen solution into a functional program. The stages “Evaluate implemented solution“ and “Solution testing stage“ are also identical, as they both aim to test the solution to make sure it is working properly.

The experimental group was made explicitly aware of the different stages of problem solving. When the students requested help from the teaching staff, the teaching staff encouraged them to reflect on their current stage in the problem-solving process. The students were also handed a physical handout that details these stages and encouraged them to keep track of the stages as they progressed through them.

At the end of each session, campers filled in an end-of-day survey that probed different aspects of their experience. The campers were asked to reflect on difficult tasks and the way in which they tackled these problems. They were also asked about their general self-efficacy in relation to web development. Finally, they were also asked about their aptitude for programming in general.

The conclusions of the research were that the experiment group who received an introduction to problem-solving was more confident and had a higher self-initiated task productivity. There were more students who added additional functionality and the students were more confident in the fact that they could grow to become better programmers, while the control group had more of a fixed mindset and believed aptitude for programming to be static.

```
sum is set to 0
for each element in list:
    add element to sum
if length list is not zero:
    set mean to sum divided by length of list
else:
    set mean to zero
```

Figure 4.2: A template for the problem of calculating the average of a list.

```
max is set to negative limit
for each element in list:
    if element larger than max:
        set max to element
```

Figure 4.3: A template for the problem of calculating the largest value of a list.

4.3 Problem solving heuristics

The problem-solving processes defined so far are abstract plans for solving problems. They define the high-level steps that need to be taken, but don't specify detailed tasks that need to be performed or the exact order in which they have to be executed. In particular, the stages that revolve around identifying and devising a solution strategy for a problem are only touched upon at a very high level. This makes sense because there are many different types of problems that exist, and there doesn't exist a single method of devising a solution. Programmers can look back at previous problems that they have solved, and possibly identify similar problems for which the solutions can serve as inspiration, but such a problem isn't guaranteed to exist. Luckily, there exist some heuristics and guidelines that aid the creative process of establishing a new solution for an unseen problem.

4.3.1 Stepwise refinement

Stepwise refinement [Sol86] is a strategy that can be applied when starting with solving a problem if the problem is too large to solve in one step. The goal is to provide a structured approach to the problem that splits the problem into subproblems in a recursive way. An assumption made here is that the student has several canned solutions for simple problems. These problems are common in programming, which is why there exist canned solutions for them. The splitting into subproblems ends when a canned solution exists for the subproblem, and this solution can be implemented in a straightforward way. An example of such a problem is computing the average of a list. A possible template for this task is presented in Figure 4.2.

This template is language-independent and provides a generic way to calculate the mean of a list. An expert programmer is able to recognize which plan can be applied in a certain situation. Expert programmers have a multitude of plans for different problems.

Suppose that the program requires the mean and maximum to be computed for the same list. An expert programmer has two templates for both of these problems. These plans seem different at first, but they have an almost isomorphic structure. They both start with initializing a variable with a default value. Afterwards, a loop is used to go over each element of the list. For each element, the variable defined earlier is (possibly) updated based on the value of the element. These plans are very similar and can be merged together in order to use only one loop instead of two. This merged plan is shown in Figure 4.4.

Stepwise refinement requires splitting a problem into multiple subproblems. However, the solutions to these subproblems have to be merged together to form a complete program. In general, there are four different ways in which two solutions can be merged together.

- Abutment: the two plans are executed sequentially and completely independent of each other. An example of this is if the two templates shown previously were to be executed after each other instead of merging them.

```
sum is set to 0
max is set to negative limit
for each element in list:
    add element to sum
    if element larger than max:
        set max to element
if length list is not zero:
    set mean to sum divided by length of list
else:
    set mean to zero
```

Figure 4.4: The merged version of the template for calculating the max value and the template for calculating the mean value of a list.

- Nesting: one plan is completely surrounded by another plan. An example of this is present in the template for calculating the largest value in a list. The if test in that plan can be viewed as a plan of its own for updating a variable based on a condition. This plan is completely contained in the plan for calculating the largest value.
- Merging: two plans are interleaved to form a new plan that is composed of the parts of the original plans. An example of this is the merged plan shown in Figure 4.4.
- Tailoring: sometimes a canned solution has the potential to be used for a problem, but isn't an exact fit for the problem. It must be slightly edited before it can be used.

By recursively splitting a problem into smaller subproblems, eventually a subproblem so small will be found that it can be trivially solved. The splitting of the problem is a top-down approach that will result in many small, trivial subproblems. These subproblems can then be combined using these four methods for merging subproblems. The merging happens bottom-up, and the end result is a global solution for the problem. By utilizing the strategy of stepwise refinement, a more structured approach can be employed to find the solution to a problem.

4.3.2 Different roles of variables in novice programs

A variable can be defined in many different ways, have many different names, and have many different values. But the ways in which a variable can be used in a program are rather limited [BS06]. A study identified ten roles that variables can assume that account for 99% of the variable usage in novice programs. By identifying the variables and their respective roles that are needed to solve a problem, the solution becomes clearer and easier to find.

- Fixed value: a variable which is initialized and whose value never changes afterwards, like a constant.
- Stepper: a variable stepping through a predictable succession of values, such as a loop variable.
- Follower: a variable that takes the last value of another variable and is always one step behind that other variable. It is following the other variable.
- Most recent holder: a variable holding the latest value that was encountered by going through a succession of values.
- Most wanted holder: a variable that holds the most suitable value encountered thus far. The criteria to determine the most suitable value can be chosen in any arbitrary way.
- Gatherer: a variable that accumulates the values encountered by stepping through a succession of values. The gatherer variable holds an accumulation of these values.
- Transformation: a variable that always holds a transformed value of another variable. The transformation variable entirely depends on the value of the other variable and the transformation that is applied.
- One-way flag: a variable that can only ever take two possible values. It is initialized with one of these two values, and once it takes the other value, it can never again regain its original value.
- Temporary: a variable that holds a value for a very short time only.

- Organizer: a variable holding an array that is only used to reorganize the values in that array.

This is relevant for problem-solving because these roles of variables account for 99% of all the variable usage in novice programs. Therefore, being familiar with these roles can aid students in their code writing and problem-solving process. When students need to devise an algorithm or convert an algorithm into an implementation, these roles can play a supporting role by offering templates in which variables can be used.

4.4 Problem solving course

At Hasselt University, there exists a course called "Problem solving" that teaches some fundamentals of problem-solving and heuristics that can be applied to go from a problem statement to a solution. This is not done in the context of programming, the course is more focused on mathematical puzzles and riddles. The general principles of problem-solving still apply here. A problem statement is presented, and a solution must be found or it must be proven that no solution can exist for most of the exercises. This course also follows a model for problem-solving, but it is focused on general problems and not only on programming.

The model consists of four steps, and for each step a series of heuristics and questions one can ask themselves are defined.

- Understanding the problem: the first task of any problem is to understand the problem. Examples of heuristics for this stage are: can I explain the problem in my own words? What information has been given to me? What is asked in this problem? Can I reorganize the problem so it becomes easier to understand? Can I introduce a useful notation? There exist many more heuristics introduced, not all of them can are applicable for every problem but they serve as a fallback in case one gets stuck.
- Devising a plan: the second step is to start planning and trying to find a solution. There is no step by step process to solve a problem, a certain amount of creativity is required. With more training and experience this step becomes easier over time. Some heuristics for this stage are: Do I know a related problem? Can I reformulate the problem? Have I used all the available information? Is there a subproblem I can solve? Can I manually solve an example? Once again, not all of these are relevant for every problem but they are useful to keep in mind.
- Carrying out the plan: this is an easier step compared to the previous two. Once a plan has been identified, it has to be fully developed and all details have to be taken care of.
- Looking back: after the plan has been carried out, it is time to take a step back and look how the solving of the problem went. Important questions that can be asked here are: Which steps in the previous stages worked and which ones were dead ends? Can the result be achieved with a different method? Can the solution method be used for different problems as well?

This problem-solving model is similar the the ones presented in the previous sections, but is also different because it is more generic and works for any type of mathematical problem.

Part II

Research and personal contribution

Chapter 5

Survey

5.1 Introduction

There are a wide variety of possible issues students can face while learning to program and obtaining the skillset to solve problems. Each of these struggles can be a hindrance to students and stand in the way of making progress. Several struggles that students face when being made familiar with computer science and programming have been documented in Chapter 2.

Not all types of difficulties students face are equally prominent in the learning process. Some of them may appear more often or form a more significant obstacle for students to overcome.

In order to document and investigate the struggles students face when being introduced to programming, a survey was used to research the effect of the struggles students face. The survey serves a dual purpose: it intends to measure the degree to which each category of problems students face impacts their overall score, as well as investigate the effect of certain aspects of a student's background, such as previous programming experience, on these problems and the marks of a student. The survey was administered in the context of a subject that serves as an introduction to programming at a university level. The answers to this survey were then analyzed to compute the correlation between self-efficacy and the marks of students.

5.2 Survey

5.2.1 Subject context

This subject is at a CS1 level and teaches the fundamentals, such as variables, types, conditions, loops, and functions. A CS1-level course is the first programming subject for a course that teaches the very essentials of programming. A CS1-level course is typically given at a higher education institution, such as a university. It requires no previous programming experience and teaches the very basics of programming, including algorithmic thinking and problem-solving. The programming language chosen for this introduction is Python, using the procedural paradigm, also known as the imperative paradigm. The imperative paradigm focuses on functions as the primary components of a program. Python also supports the object-oriented paradigm, but this is outside the scope of this subject.

It is a mandatory part of the computer science course, and it is one of the first subjects for this course, as well as the first subject that touches upon programming. It can be followed as an optional subject by the Master of Commercial Engineering in Business Informatics and the educational Master's in the Sciences, focusing on computer science. However, the vast majority of the students are part of the Computer Science course. The subject is split into two parts, each taking around 8 weeks. The two parts combined form an entire semester. Each part is treated as a separate subject that stands on its own. The marks for both parts are split across an exam that counts for 90% of the marks and several mini-projects that account for the remaining 10% of the marks.

A subject day is organized as follows: first, a lecture is given in the morning. This lecture usually takes around 1 hour and 30 minutes to complete. Afterward, students are given exercises and reading

material, which they have the entire day to complete. A teaching assistant is always present in the classroom for the students who are in need of guidance or have questions about the subject material. At the end of the day, the teaching assistant will recap and discuss the assigned exercises for that day and possible solutions for them.

5.2.2 Survey contents

The answers to the survey were completely anonymous. The survey contains questions about computer science knowledge and skills as well as the student's background. There is also a question asking the student to give their final mark for the subject. The final mark is used to find patterns in the data. The survey aims to discover which parts of computer science and background characteristics impact students' progress the most if students struggle with them. A student's final mark is used as an indicator of their total progress throughout the subject.

The background characteristics measured are the amount of previous programming experience, the number of weekly mathematics hours in the last 2 years of secondary school, and the level of the course taken in secondary school. These are the characteristics that make it plausible that they can influence the proficiency of students. Prior programming experience before starting the computer science course gives students a head start, which could provide an advantage over students without experience. It is also possible that this head start isn't significant enough to influence the marks in a noticeable way or that prior experience can be a disadvantage if previous programming content was taught or interpreted in an incorrect way, leading to faulty knowledge. The number of mathematics lessons received is also a possible indicator of future performance. Having been trained in mathematics can provide an advantage because of the similarities to programming. Both are abstract forms of thinking that require problem-solving skills to tackle new problems for which no trivial solution exists. Being proficient in either one teaches skills that may transfer over to the other domain, making it easier for students to orient themselves when given an introduction to the new domain.

The level of education is also a factor that can influence the performance of students, and as such, it is inquired about in the survey. There are 4 possible levels of education in secondary school in Belgium. General education (ASO) has a strong theoretical aspect and prepares students for higher education. Technical education (TSO) also has a theoretical component, but also offers some electives that are more concrete and technical than ASO. Art education (KSO) is the same as TSO, but the electives focus on the arts instead of technical subjects. These are background characteristics that can influence computer science students' performance. Vocational education (BSO) directly prepares students for the workforce, and students aren't expected to pursue higher education with a BSO degree. Because the goal of the education level "ASO" is to prepare for higher education, it can be expected that students with this level of education will perform better than students with a different level of education.

The students were asked about their background in relation to these characteristics to measure their influence. The number of weekly mathematics hours in the last two years before starting higher education was asked to gauge the students' mathematics proficiency. An important note to make here is that the amount of mathematics hours can be the same for multiple types of education, but the content can be more or less extensive and contain different mathematical topics. This means that students with the same number of mathematics hours can still have different experiences with mathematics. On top of that, the amount of weekly mathematics hours doesn't give an indication of how good the student was at mathematics, and only shows the amount of experience they have. The students were asked about their level of education as well as the amount of their prior programming experience. They could choose between "no experience", "a little bit", "medium", and "a lot" as categories for their experience.

The common problems students face and their influence on their marks were also investigated. Several statements were presented to the students, and the students were asked to self-assess their abilities in the context of the statement. An example of a statement is "I know how to begin an exercise if I don't immediately see the path to a solution". The responses to these statements were measured on a 10-point Likert scale. Should a student be very confident about their capabilities when it comes to starting an exercise, they will rank themselves towards the higher end of the scale, and if they don't believe they are skilled at starting an exercise, they will rank themselves towards the lower end of the scale. The following statements were presented to the students:

- I know how to begin an exercise if I don't immediately see the path to a solution.
- I can devise an algorithm that solves a given problem.

Amount of exp.	Avg.	Median	Median conf. intvl.	Std. dev.	Nr. of students
No experience	7.4545	7.0	[2.0, 9.0]	6.5324	11
A little bit	14.5556	15.0	[11.0, 19.0]	4.6128	9
Medium	13.7143	19.0	[8.0, 19.0]	7.4322	7
A lot	11.6000	15.0	[4.0, 20.0]	7.2319	5

Figure 5.1: A table showing the average, standard deviation, median, median confidence interval, and number of students for each category of experience.

- I can choose the appropriate data structures (=lists, strings,...) and variables to implement an algorithm.
- If I get an error after executing a program, I can debug this and locate the bug.
- If I get the wrong output after executing a program, I can debug this and locate the bug.
- If I find the source of a bug, I can remedy the bug.
- I am motivated to solve exercises.
- If I struggle with an exercise, I give up quickly.
- I have enough time to solve my exercises.
- I immediately begin programming when I start an exercise.
- I see the benefit of solving exercises to understand the learning material and to pass the exam.

The survey intends to gauge students' difficulty with the following domains: problem-solving, algorithmic thinking, debugging, motivation, behavior, and time management. The first two questions relate to proficiency in problem-solving. The second and third questions measure proficiency with algorithmic thinking. Questions 4-6 are about debugging skills, specifically the skills to find and remedy a bug. Finally, the last 5 questions are about the student's behavior. They gauge their motivation, time management skills, and behavior when working on exercises.

These problem domains are identified by [MRF19a] by performing a systematic literature review on the literature surrounding introductory programming.

5.2.3 Results

The survey was administered after the first part of the subject, meaning 8 weeks of study and practice, as well as an exam, had passed for the students. A total of 33 students answered the survey, but one student filled in the survey but did not give their final mark. The answers of the remaining 32 students were analyzed. To place this number in context, a total of 53 students started the subject. The survey was conducted during the first lecture of the subject's second part to maximize the responses.

The influence of background characteristics is measured by computing each possible category's average and standard deviation and comparing these values to determine if some categories influence the marks more. The amount of experience students have in relation to programming before taking the subject is considered first. The average, standard deviation, median, median confidence interval, and number of students for each category are shown in Figure 5.1. Most noticeably, the average mark for students with no previous experience in programming is significantly lower than that for students with at least a little experience. It is difficult to draw conclusions based on only this information, as the average and standard deviation don't tell us everything about the data. A boxplot is typically used to understand a given dataset fully by visualizing statistical information about the data distribution based on a five-number summary. The five numbers, labeled Q_0 to Q_4 are defined in the following way:

- Q_0: the minimum value present in the dataset, excluding any outliers.
- Q_1: the median of the lower half of the dataset, which can be interpreted as the value observed after passing through 25% of the data. This is also known as the first quartile.
- Q_2: the median of the full dataset, this is the value that splits the dataset into two equal parts.
- Q_3: the median of the upper half of the dataset. This is also known as the third quartile.
- Q_4: the maximum value present in the dataset, excluding any outliers.

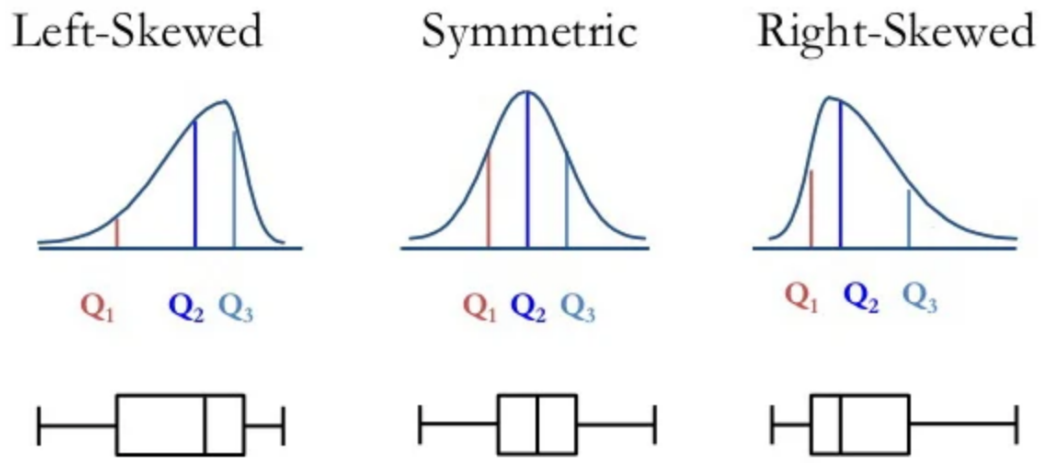

Figure 5.2: Boxplots showing the distribution of data and their corresponding boxplots for a left-skewed, symmetric, and right-skewed distribution.
Source: `https://www.simplypsychology.org/boxplots.html`

Outliers are classified using the interquartile range (IQR), which is defined as the distance between Q_1 and Q_3, IQR = Q_3 - Q_1. An outlier is defined as any data point for which the value is either larger than Q_3 + 1.5 * IQR or smaller than Q_1 - 1.5 * IQR. Outliers are shown as individual points in a boxplot. Outliers are given that name because they differ significantly from the other points in the dataset. Significantly different points could be anomalies and noticeably shift the maximal or minimal value. For that reason, they are displayed as individual points.

A boxplot is visualized by drawing the interval between Q_1 and Q_3 as a box and highlighting the median Q_2 with a horizontal line. Whiskers extend from the box to the minimal Q_0 and the maximal value Q_4. The whiskers extend up to 1.5 times the IQR at most, but are only drawn up until the minimum and maximum. Their length can be different because of this. There is always 25% of the data located in the interval between Q_n and Q_{n+1}. This means a quarter of the data is located between the median line and the bottom, as well as the top of the box. The remaining two quarters are spanned by both whiskers above and below the box.

The position of the median, the size of the box, and the length of the whiskers indicate the distribution of the data. If the data is densely distributed, the box will be smaller, and the whiskers will be shorter. As an example of this, Figure 5.3 shows different kinds of distributions. The distribution for “No experience“ is more densely compacted than the distribution for ”Medium”. For the latter, the bottom whisker is also relatively long. The median is at the top of the whisker. Half of the data is below the median, and the other half is above it. This indicates that there are a lot of data points at 18 and above and that the bottom half of the data is spread out very loosely between 1 and 18.

The position of the median can indicate if the data is left-skewed, right-skewed, or symmetric. If the median is positioned around the middle of the box, the interval between Q_1 and Q_2 and the interval between Q_2 and Q_3 will be of equal length, indicating that the distribution is symmetric. On the other hand, if the median is close to the top of the box and the whisker on top of the box is on the short end, the data is right-skewed. On the other hand, if the data is left-skewed, the median will be positioned close to the bottom of the boxplot, and the whisker at the bottom will be shorter. Examples of different types of skew and their corresponding boxplots are shown in Figure 5.2.

Figure 5.3 shows boxplots that visualize the distribution of data for the 4 categories of programming experience: none, a little, medium, and a lot. This is the same data for which the average, standard deviation, and number of students were shown in Figure 5.3. The distribution shows that the marks for the category ”a little bit” are the highest and most consistent. Only a single student had a mark below 10. All the other students scored higher and thus passed the subject. This consistency is also visible in the fact that the standard deviation for this category is significantly smaller than the other categories. The marks for the categories ”medium” and ”a lot” are more spread out and less consistent. The distribution for ”medium” is right-skewed since the median is at the top of the box and the top

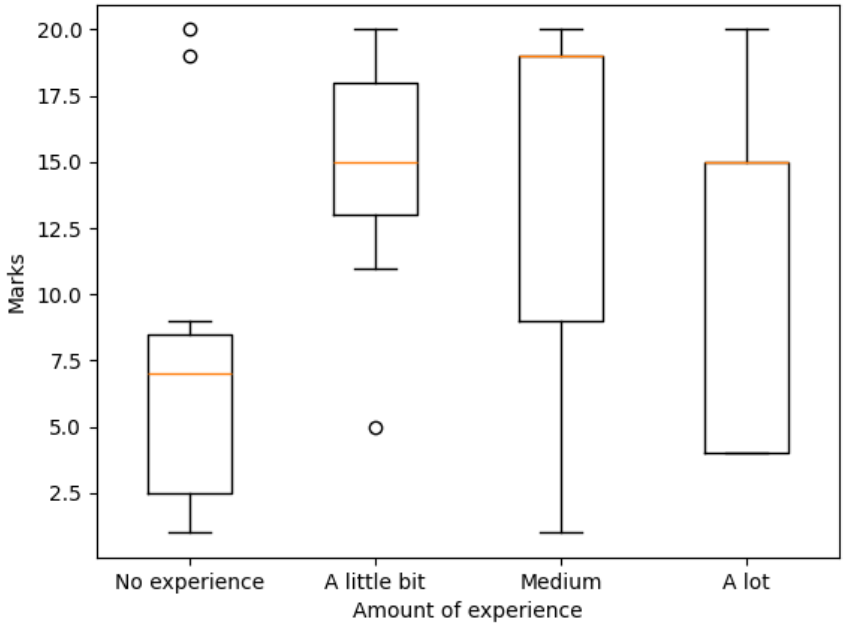

Figure 5.3: Boxplots showing the distribution of marks for all categories of programming experience.

Math hrs.	Avg.	Median	Median conf. intvl.	Std. dev.	Nr. of students
2h	1.0	1.0	/	0.0000	1
4h	9.5	9.0	[4.0, 15.0]	5.8310	8
5h	15.0	15.0	/	0.0000	1
6h	12.3077	14.0	[7.0, 18.0]	6.3821	13
7h	10.5	10.5	[1.0, 20.0]	10.4083	4
8h	14.6	19.0	[7.0, 20.0]	6.5038	5

Figure 5.4: A table showing the average, standard deviation, median, median confidence interval, and number of students for the different amounts of mathematics hours.

whisker is very short. The median for "a lot" is also at the top of the box, but the bottom whisker is nonexistent. This means that the data is very dense at this position, and a significant number of students scored around 5. An important side note for these boxplots is that each category's data is rather limited, as is visible in Figure 5.1. For example, the category "a lot" only has 5 students. Because there are only a few data points, the position of the five numbers is highly dependent on these points. For example, two of these students scored a 4 as their mark. Two out of five is more than 25% of the data, which means Q_0 and Q_1 have the same position. This causes the bottom whisker to collapse.

The other background characteristics are looked at using the same methods. Figure 5.4 shows the information for the different amounts of mathematics hours. There was only one student who had two weekly hours of mathematics and one student who had five weekly hours of mathematics. There is not enough information to say anything meaningful about these groups. The group with eight weekly hours of mathematics had the best scores on average, followed by the group with six weekly hours. The group with seven mathematics hours has a very high standard deviation compared to the others. The confidence interval for the median is very large because of the limited number of students per category. Figure 5.5 shows boxplots for each of the different amounts of weekly mathematics hours.

The groups for two and five weekly mathematics hours only contain one student, meaning Q_0, Q_1, Q_2,

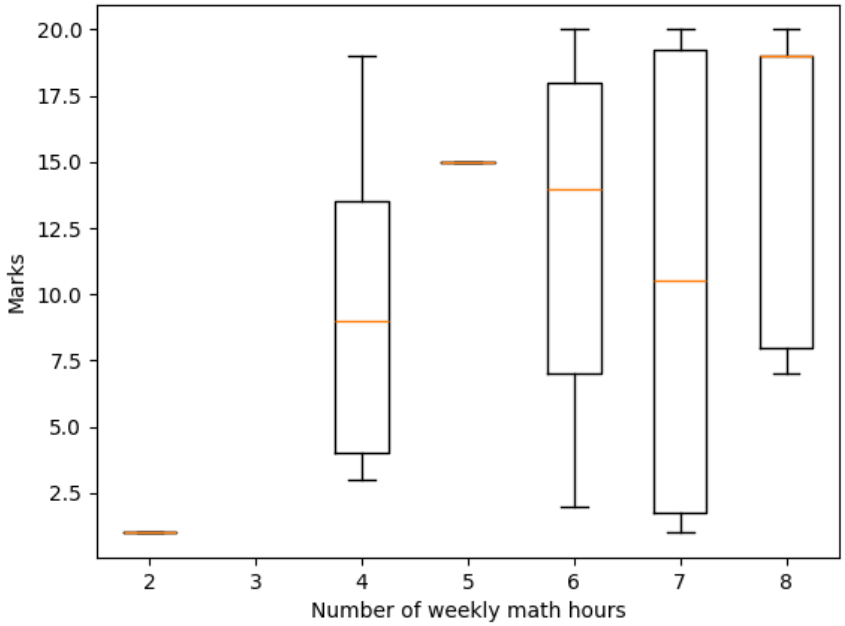

Figure 5.5: Boxplots showing the distribution of marks for the different amounts of mathematics hours.

Education lvl.	Avg.	Median	Median conf. intvl.	Std. dev.	Nr. of students
ASO	11.7273	12.5	[7.0, 18.5]	7.0453	22
TSO	11.0	13.0	[4.0, 19.0]	6.9642	9
KSO	10.0	10.0	/	0.0000	1

Figure 5.6: A table showing the average, standard deviation, median, median confidence interval, and number of students for the different levels of education.

Q_3, and Q_4 all have the same value, causing the boxplot to collapse in on itself. The top whisker for the mark for the group with four weekly mathematics hours is significantly longer than the bottom whisker, but this doesn't necessarily indicate a right skew. The median is centered in the box, and there are only 8 data points, which makes it difficult for the boxplot to represent the distribution accurately. The boxplot for the group with six weekly mathematics hours has whiskers of more equal size compared to the group with four hours. The median is also centered in the box, indicating a symmetric distribution. The group with seven weekly mathematics hours has a very large box and very small whiskers, indicating that a big part of the data is located around the edges. We know that there are only four students in this group, so from the boxplot, we can deduce that 2 students scored very high marks and the other two students had very low marks. This aligns with the high standard deviation observed for this group in Figure 5.5.

Table 5.6 shows the average, standard deviation, median, median confidence interval, and number of students for the different levels of education. There is only one student who had "KSO" as education, so there isn't enough information to interpret this data in a meaningful way. The average and standard deviation are very similar, but there are fewer students from the "TSO" level of education.

Figure 5.7 shows several boxplots visualizing the distribution of data for each of the different levels of education. The boxplot for "KSO" has collapsed because there is only one student for this level of education. The boxplot for "ASO" has a short whisker at the top of the box, indicating that around 25% of students had very high marks above 18. Compared to this, the boxplot for "TSO" has a long top whisker. The confidence interval for the median is once again very large because of the limited

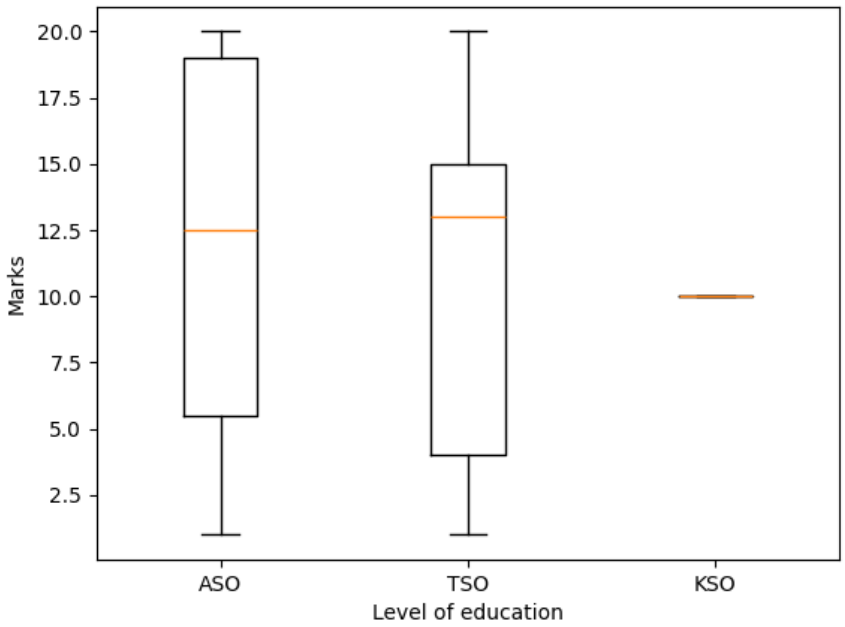

Figure 5.7: Several boxplots showing the distribution of marks for the different levels of education.

number of students per category.

5.2.4 Interpretation

A histogram visualizing the answers to the question "I know how to begin an exercise if I don't immediately see the path to a solution" is visible in Figure 5.8. The final marks were grouped into 5 categories, 0-4, 5-8, 9-12, 13-16, and 17-20, to make the visualization easier to understand and interpret. The colors were chosen intuitively, with red being the worst marks and dark green being the best marks. The x axis displays the agreement of the students to the question.

The majority of the higher marks are on the right-hand side of the graph, corresponding to the higher scores students gave themselves for the statement. This is consistent with the literature, suggesting problem-solving is an essential skill for programming and computer science. This pattern is also present in Figure 5.9, which is a visualization of the answers to the question, "I can devise an algorithm that solves a given problem.". The majority of the high marks correspond again to a higher self-assessment of students on their algorithm-constructing abilities.

An inverse pattern can be observed in Figure 5.10, which visualizes the question, "If I struggle with an exercise, I give up quickly.". Higher marks appear more on the left-hand side of the histogram, indicating that an inverse relation may exist between the answers to the question and the mark. This is unsurprising, as the question is about a negative trait. A high agreement with the statement means a student has a tendency to give up quickly. The other two questions that were visualized are about problem-solving, a positive skill to have, and in that context, high self-assessment indicates skill in problem-solving, which is a positive influence.

While this kind of visualization can provide a quick and intuitive overview of the answers and their relation to the marks, there are limitations to this approach. The marks are binned into 5 categories, which means, for example, that a mark of 15 and a mark of 16 will have different colors in the visualization even though they are very similar. With specific distributions of values, this can noticeably shift the colors present in the graph, which makes interpreting the results less consistent. The number of bins influences how strong this effect is. The more bins there are, the less this effect will be present, but the more difficult it will be to interpret the graph.

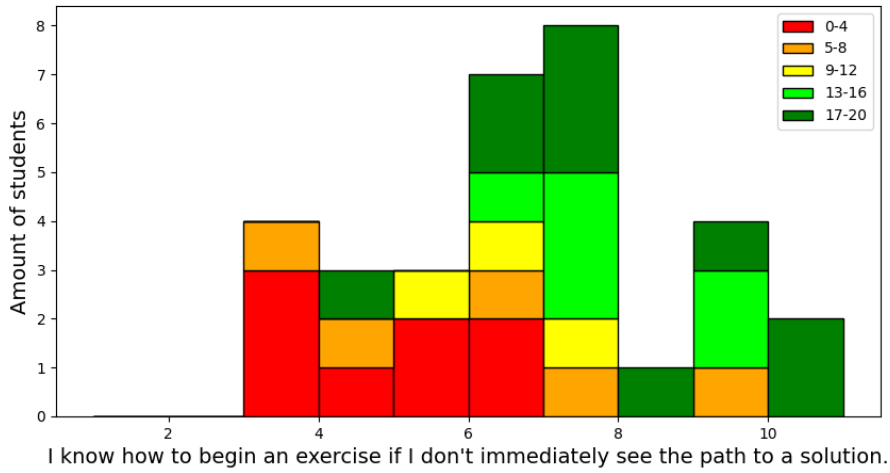

Figure 5.8: Answers to the question "I know how to begin an exercise if I don't immediately see the path to a solution." visualized by a histogram.

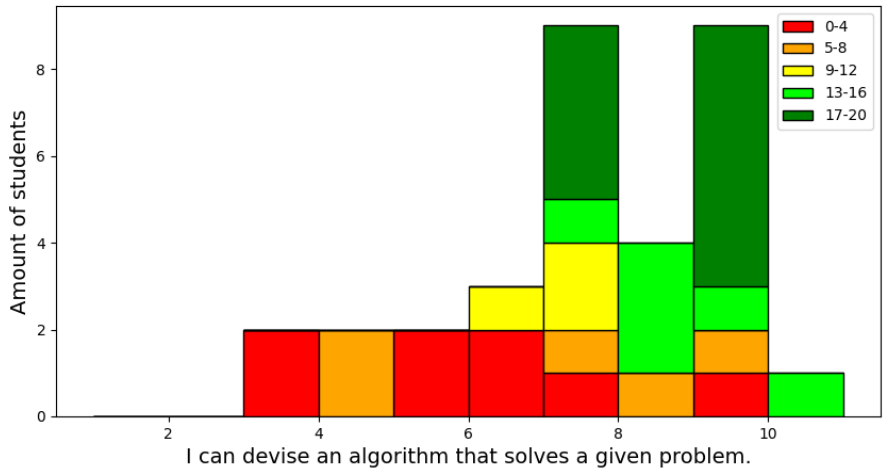

Figure 5.9: Answers to the question "I can devise an algorithm that solves a given problem." visualized by a histogram.

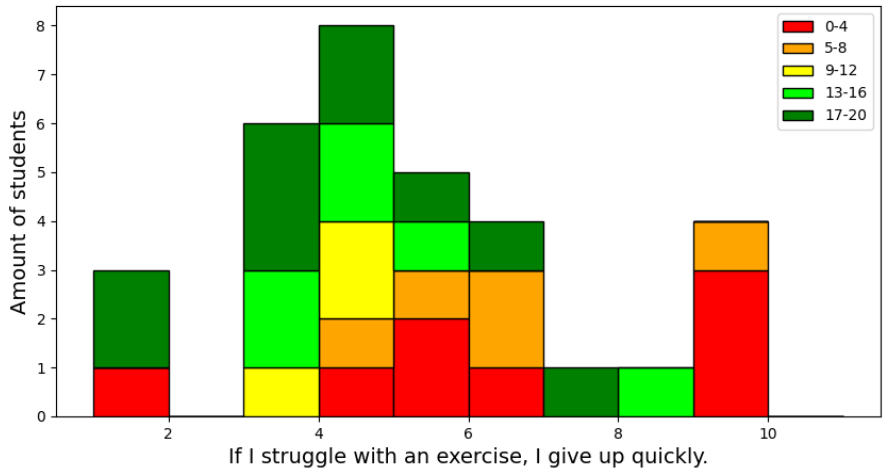

Figure 5.10: Answers to the question "If I struggle with an exercise, I give up quickly." visualized by a histogram.

Another difficulty is comparing the magnitude of the relationships between different answers to the survey and the marks. The problem previously mentioned also has an influence here, as changes in color can shift the interpretation of the strength of the relation. In general, a visual interpretation isn't suited to precisely pinpoint the strength of a relation as it leaves room for interpretation.

To more accurately determine the degree to which proficiency in a computer science subtopic influences the final mark, the correlation between the answers to the questions and the marks was used. The formula used to calculate the correlation uses covariance. The following formula defines the covariance between two random variables X and Y, where $E[X]$ is the average for X, $E[Y]$ is the average for Y.

$$cov(X,Y) = E[(X - E[X])(Y - E[Y])] = E[XY] - E[X]E[Y]$$

The covariance between two random variables is an estimator of a possible linear relationship between the two random variables. A covariance close to 0 indicates no linear relationship between the variables. If the covariance is negative, it suggests that an inverse linear relationship exists between the variables. This means that the larger one variable grows, the smaller the other variable will become. A positive covariance suggests a linear relationship. If one variable grows larger, so will the other variable. The following formula defines the correlation between two random variables X and Y, where σ_X and σ_Y are their respective standard deviations:

$$corr(X,Y) = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X - E[X])(Y - E[Y])]}{\sigma_X \sigma_Y}$$

This formula produces a value in the range [-1, 1]. The correlation is a standardized version of the covariance. The covariance depends significantly on the mean of the random variables. If the mean is large, the covariance will have a high value. This makes it difficult to compare the strength of a possible linear relationship between variables with different orders of magnitude. Correlation balances this out by dividing it by the product of the standard deviations, providing a normalized comparison method.

It is important to note that the covariance only works for linear relations. If the relation is of a different order, the covariance could potentially be close to zero. The covariance will also be zero if no relation exists between the variables. The only solid conclusion that can be made when the covariance is zero is that no linear relation exists between the variables. This is true for the correlation by extension because it is a rescaling of the correlation to the interval [-1, 1].

The correlation between the answers to the survey and the marks is presented in Figure 5.11. The correlation is between 0.5 and 0.65 for most of the statements. This implies that a significant linear relationship exists between students' self-assessments and their final marks. This confirms the relevancy of these problem areas for beginning students as outlined in the literature. However, not all of the

Statement	Correlation with marks
I know how to begin an exercise if I don't immediately see the path to a solution.	0.6398
I can devise an algorithm that solves a given problem.	0.6737
I can choose the appropriate data structures (=lists, strings,...) and variables to implement an algorithm.	0.5542
If I get an error after executing a program, I can debug it and locate the bug.	0.6039
If I get the wrong output after executing a program I can debug this and locate the bug.	0.5340
If I find the source of a bug, I can remedy the bug.	0.5661
I am motivated to solve exercises.	0.5401
If I struggle with an exercise, I give up quickly.	-0.4257
I have enough time to do my exercises.	0.3890
I immediately begin programming when I start an exercise.	-0.0387
I see the benefit of solving exercises to understand the learning material and to pass the exam.	0.0893

Figure 5.11: The correlation between each item and the marks of the students.

correlations are positive or significant. The correlation for the statement "If I struggle with an exercise, I give up quickly" is -0.4257. As mentioned previously, a negative correlation implies an inverse linear relationship. The question measures the students' tendency to give up when they face difficulty solving an exercise. Giving up is a negative trait, as solving exercises is an important part of the curriculum that helps teach programming skills.

The strongest correlation is for the statement "I know how to begin an exercise if I don't immediately see the path to a solution," and "I can devise an algorithm that solves a given problem.". The correlations for these two items are 0.6398 and 0.6737. These are the two statements related to problem-solving and algorithmic thinking. The fact that these two have the strongest correlation can be interpreted as a possible indication that this subarea is the most important part that beginning programmers struggle with.

The correlation for the last two statements is close to 0, which means there is no linear relation between the agreement with the statement and the marks. The distribution of the agreement to the statement "I see the benefit of solving exercises to understand the learning material and to pass the exam" in Figure 5.12 shows that the vast majority of them are 9 or 10. Because the data is so concentrated, the differences between the mean of the agreements and the agreements are small. The covariance and correlation are calculated by taking the expected value of these differences, meaning that the covariance and correlation will also be small. The intuitive meaning of this is that it's unlikely for a strong relationship to exist between variables when one of them shows little variability.

The correlation for the statement "I immediately begin programming when I start an exercise" is also close to zero. As shown in Figure 5.13, the distribution of the answers to this statement is a lot more spread out. The fact that the correlation is close to zero means no linear relation exists. This can mean one of two things: either a relation of a different magnitude or no significant relationship exists.

There is no clear pattern visible between the agreement and the marks. Based on this visualization and the correlation, it is more likely that no relation exists. However, as mentioned before, these histograms give an intuitive overview but aren't suited for an exact interpretation. It is impossible to say that no relationship exists based on the correlation and the visualization alone. The soft conclusion drawn is that no relation seems to exist, meaning the tendency of students to start programming immediately when beginning an exercise has no influence on their marks.

It is difficult to compare the self-assessment and the background characteristics. Their influence on the marks is particularly hard to gauge because the influence of the self-assessment is measured with the correlation, but the influence of the background characteristics is only shown with the different distributions per category of a specific characteristic. These two different methods make a different comparison

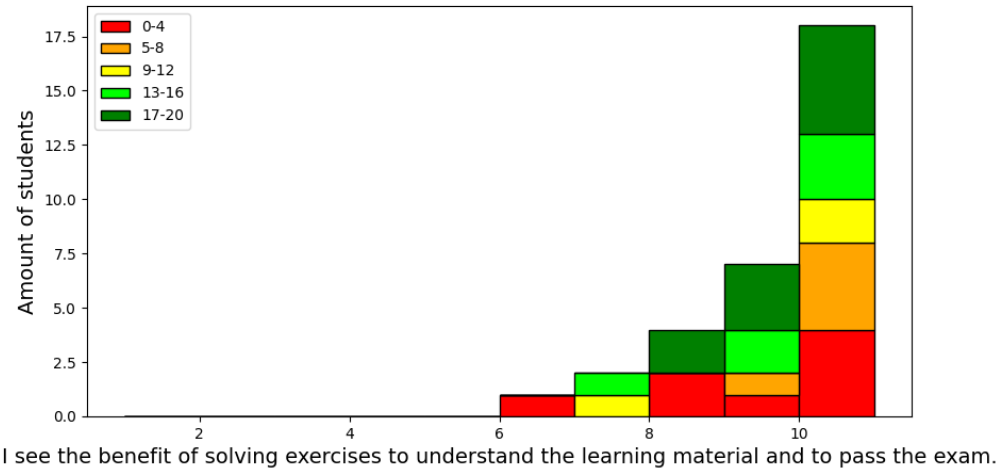

Figure 5.12: Answers to the question "I see the benefit of solving exercises to understand the learning material and to pass the exam", visualized by a histogram.

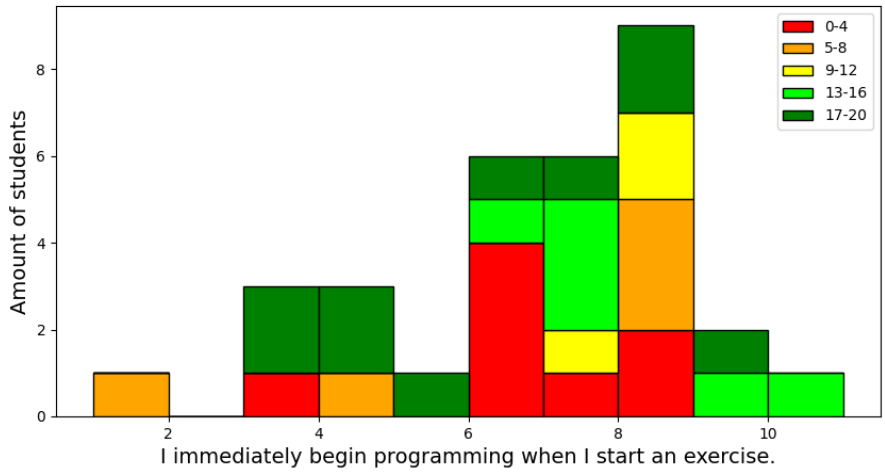

Figure 5.13: Answers to the question "I immediately begin programming when I start an exercise", visualized by a histogram.

impossible. This comparison becomes a lot easier should the method of measuring the influence be standardized across the background characteristics and the self-assessment. There are no categories present for the self-assessment, but the correlation can be calculated based on the characteristics. A critical note to make is that because some of the categories are nominal and aren't numbered, the correlation cannot be calculated in a straightforward way. The amount of experience is only described with the categories "no experience", "a little bit", "medium ", and "a lot".

These categories need to be converted to ordinal data first before the correlation can be computed. The categories can be ordered from least to most experienced and should be numbered in the same way. Intuition dictates that more experience corresponds with greater proficiency in the subject, so the category "a lot" should get the highest ranking, and the category "no experience" should get the lowest ranking. Deciding the specific numbers that accompany this order is difficult. How big is the difference between categories? Is the gap in experience between "no experience" and "a little bit" as big as the gap in experience between "a lot" and "medium"? This difference cannot be measured objectively, and any numbers chosen to represent this difference will be arbitrary. Nevertheless, it provides a method to compare the influence on the marks between the self-assessment and the programming experience.

The numbers picked to represent the different categories of experience are 0-3, which match the magnitudes of experience. The correlation computed with this choice of numbers is 0.2659, which is almost half the correlation for most of the self-assessments. This seems to suggest that the amount of prior programming experience has a lesser influence on the marks than the overcoming of the most common difficulties when introduced to programming. The correlation for the amount of mathematics hours is easier to compute as the number of hours is ordinal data. As mentioned before, the number of mathematics hours isn't an exact specification of the experience with mathematics, as the same number of hours can entail different amounts of content being taught. The correlation for the number of mathematics hours is 0.3037, which seems to imply that this characteristic influences the marks, but the influence isn't as strong as that of most of the self-assessments. The correlation for the level of education wasn't computed as the boxplot and distribution suggest that its influence is minimal. There is also no straightforward way to rank the education levels "ASO" and "TSO", which is similar to ranking the amount of experience.

So far, the correlation between the background characteristics and the differences in statistical distribution has been used to gauge the impact the background characteristics have on students' marks. These can serve as indicators, but ultimately aren't conclusive on the significance of the differences. As an example, the boxplots shown in Figure 5.3 seem to indicate that students who have no prior experience in programming have significantly lower marks on average. It is difficult to interpret how significant this difference is based on the boxplot alone. A boxplot is useful to visualize the distribution of the data, but it doesn't give information on how significant the differences are between boxplots. This is true in particular if the size of the data visualized by the boxplot is different, as this isn't immediately visible in the boxplot itself.

A better method to draw conclusions is to compare the categories using statistical methods. To do this, we use a null hypothesis H_0 and an alternative hypothesis H_A.[Bar19] These are mutually exclusive statements. Either H_0 is true or H_A is true, but not both simultaneously. A statistical test is then performed on sample data, and based on the test result, H_0 is rejected in favor of H_A or accepted. There are different types of H_A. For example, if the null hypothesis states that the mean μ equals a value μ_0, then possible alternative hypotheses are $\mu \neq \mu_0$, $\mu < \mu_0$, and $\mu > \mu_0$. These are called a two-sided alternative, a one-sided left-tail alternative, and a one-sided right-tail alternative.

Because there is only a random sample of the data available, the decision to accept or reject H_0 can still be wrong. Two types of errors can be made in relation to the acceptance or rejection of H_0. If H_0 is true but H_0 is rejected in favor of H_A, that is called a type I error. If H_0 is false but H_0 is accepted, that is called a type II error. Tests are done at a significance level α, which is the probability for a type I error $\alpha = P\{reject\, H_0 \mid H_0\, is\, true\}$. If $\alpha = 0.05$, that means there is a probability of 5% that we falsely reject H_0 in favor of H_A if H_0 is true.

The standard method to perform a level α test consists of 3 steps[Bar19]:

- A test statistic T is chosen that measures the distance to the situation where H_0 is true. This is based on the assumption that H_0 is true and there is a tabulated, known distribution F_0. This distribution is also called the null distribution.

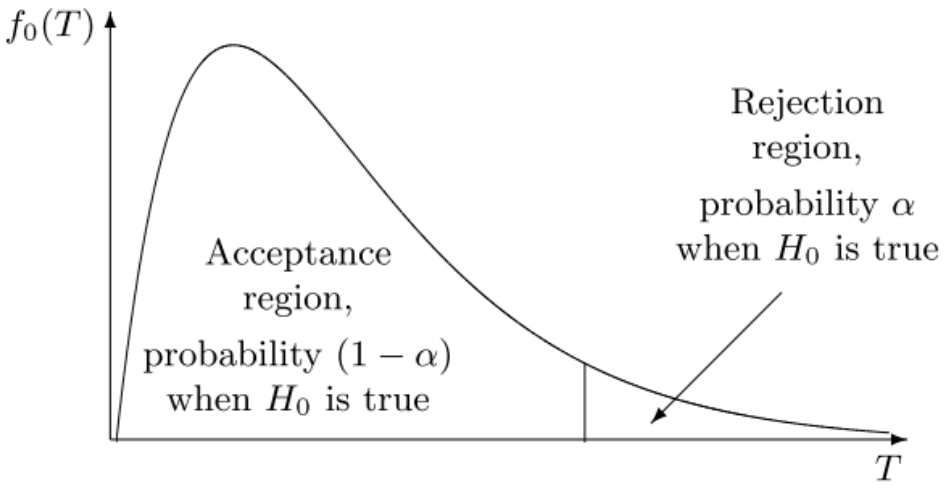

Figure 5.14: The acceptance and rejection regions visualized in the context of the density function f_0 and the test statistic T.
Source: [Bar19]

- The density of f_0 of the null distribution will always have an area of 1. A part of this area is considered where the subarea is equal to α. This area is called the rejection region. The probability of T being part of the rejection region is α under H_0. The remaining part of the density is called the acceptance region. Because it is the complement of the rejection region, its area will be $(1-\alpha)$. The area doesn't have to consist of one part. It can also be split into two subparts if H_A is a two-sided alternative.
- If T belongs to the rejection region, H_0 is rejected in favor of H_A because the data shows sufficient evidence in favor of H_A. If T belongs to the acceptance region, H_0 is accepted. The probability that we accept H_A even if H_0 is true is equal to α because of the way we defined the rejection region.

It is common to show the p-value for the test statistic T. The p-value is the probability of seeing an equal or more extreme observation of T under H_0. If $p < \alpha$, T is in the rejection region because of the definition of the rejection region. The p-value provides the necessary context to the test statistic in cases where p is very close to α, and the decision between rejection or acceptance is a small distance apart. It also serves as an indication of the certainty of the decision. A rejection or acceptance based on a p-value that is significantly smaller or larger than the significance level is made with strong confidence.

For the null hypothesis, the amount of programming experience is taken as an example. The null hypothesis for the number of weekly mathematics hours and the level of education is defined analogously. The null hypothesis for the programming experience states that the amount of programming experience does not have an influence on the marks. Under this null hypothesis, each category should have the same distribution of marks. H_0 = the distribution for each category is identical. The alternative hypothesis H_a states that programming experience does have an influence on the marks. The distributions of data should not be equal under the alternative hypothesis. It is important to note that the alternative hypothesis does not specify any relation between the different categories. It only states that the distributions are unequal.

The Kruskal-Wallis test was chosen to analyze the background characteristics. The Kruskal-Wallis test is a nonparametric test where no assumptions are made about the distribution of the data. Because of this, they are applicable in more contexts than parametric tests, but they are less powerful for the same reason. In this context, there isn't enough information to make any assumptions about the data, and the sample size is too small to apply the central limit theorem.

The central limit theorem states that significant quantities of data approximate a normal distribution, but 30 or more samples are needed to apply this theorem. There are 32 samples in total, but they

Characteristic	statistic value	p value
Amount of prior experience	5.5691	0.1346
Amount of weekly mathematics hours	4.9228	0.4254
Level of education	0.1257	0.9391

Table 5.1: The different background characteristics and their respective statistic and p values.

are spread across three to six categories. The number of samples per category is significantly lower than 30, so the central limit theorem cannot be applied. The significance value chosen for this test is $\alpha = 0.05$ because this is the standard choice.

The scipy[1] implementation of the Kruskal-Wallis test was used. The results are shown in Table 3.5.

The p-value for each characteristic is larger than the significance value of 5%, so the null hypotheses are accepted. There isn't enough evidence present to reject any of them in favor of the alternative hypothesis. The p-values indicate that the level of education is almost certainly of no influence. The odds of seeing a more extreme observation are 93%. This aligns with the boxplots shown in Figure 5.7 for each of the different levels of education. Since there is only one student with an education level of "KSO," it's impossible to say anything meaningful about this category, but the distribution between "ASO" and "TSO" is very similar. The number of weekly mathematics hours has a p-value of 42%, which is also a very strong indication that this characteristic does not have an influence on the marks. Figure 5.5 shows that the boxplots for each number of weekly mathematics hours don't differ significantly from each other. The lower bound for the students with eight weekly hours and the general distribution is higher than the other categories. The conclusion based on the Kruskal-Wallis test is that this difference isn't significant.

5.3 Limitations

All questions, minus the last one, ask the student to self-assess their programming, problem-solving, and critical thinking skills as well as their behavior. The survey heavily relies on the students' self-assessment, which is not an objective, reliable method of measurement.

The self-esteem of a student can significantly influence self-assessment, resulting in a blurred representation of their true abilities. This shift in the data has the potential to obfuscate real patterns that are present and make discovering these patterns more difficult. However, the analysis revealed expected patterns in the data. The correlation calculated shows a significant relation between the marks and the self-assessment performed by the students. This relation is the expected result, as the questions asked were based on the most common difficulties encountered by students as described in the literature. The higher the self-assessment of the students, the fewer these issues were present, and the higher the expected marks of the students. Based on this evidence, the bias of the students' self-esteem seems not to have any significant influence on their answers.

Because the students were asked in class to fill in the survey, and all of them complied, there is no selectivity bias present. The sample is representative of the full class that started the second part of the subject. However, 53 students started the subject originally, and only 33 students answered the survey, out of which one student didn't enter their marks and was thus removed from the analysis. Twenty students took part in the first part of the subject but did not participate in the second part. It is possible that a considerable number of these students actively took part in the first part of the subject but decided to drop out after disappointing results. In this scenario, the distribution of marks would be shifted towards the higher end of those who participated in the survey, influencing the analysis.

The sample consists of 32 students, which is sufficient to analyze the responses to the statements. However, these 32 students were further split across multiple groups to analyze the background characteristics, leading to each category having few data points, making it difficult to compare the different categories. This affects the analysis in two ways: the limited sample size gives less data to work with and forces the use of the Kruskal-Wallis test. The Kruskal-Wallis test is a nonparametric test, which

[1] https://scipy.org/

is less powerful than a parametric test. With more data points per category, the central limit theorem would be applicable, giving the opportunity to use a stronger test.

This difference could, in particular, influence the results of the test performed on the amount of programming experience. The p-value of this test is 13%, which is relatively close to the significance level α of 5%.

5.4 Conclusions

A survey was presented that researches how and if background characteristics, as well as the most common difficulties, influence the marks of students. This was done in the context of a CS1 subject at the university level, where students were asked to self-assess their proficiency with these cruxes. Based on this survey, education level, amount of weekly mathematics hours, and the amount of prior programming experience have no significant influence on the students' marks. This doesn't have to mean that these characteristics don't have an inherent influence on programming proficiency. The subject only touches upon the very basics of programming and requires no previous programming experience. It is possible that students with more experience with programming or mathematics have an increased proficiency later in the course or only in relation to computer science topics that lean towards mathematics. It is also possible that prior experience could matter more in subjects that are structured differently at a different university or focus on different aspects of programming. More research is needed to definitely say that there is no inherent influence on the marks from prior programming experience, the amount of mathematics hours, and the level of education.

Overcoming the most common difficulties outlined in the literature when introduced to programming significantly influences the marks of a student. These cruxes relate to different areas, such as problem-solving, debugging, student behavior and habits, and algorithmic thinking. The correlation between the self-assessment and the final marks of the students is between 0.5 and 0.65 for most of these areas. The highest correlation was observed for problem-solving, with a correlation of 0.6737 and 0.6398. The conclusion is that problem-solving is the most important subarea of programming to learn and overcome difficulties.

Chapter 6

Educational data analysis

6.1 Introduction

Educational data analysis is about predicting the likelihood that a student will pass a course ahead of time. This can be done by analyzing different types of data. One such type of data is the submission data of programming exercises. By analyzing this data, a machine learning model can be trained that predicts the performance of future students. The predictions of this model can then be used to identify struggling students early on in the course and allow a teacher to intervene if necessary.

One particular approach calculates different features from the submission data and uses these features as input for the model [Van24]. Many different types of features are used, and the data in this context is gathered from the submission platform Dodona. The features computed from this data can then be used to train a model on previous editions of the course, and this model can be used to predict the future performance of students. The features are computed for certain snapshots, which are fixed moments during the course that can be aligned over multiple years. For each of these snapshots, a model can be trained on the submissions up until that point.

Apart from the predictions themselves, the model itself also contains valuable information. By analyzing the coefficients of the model, the most important features can be identified, which gives an instructor more information on possible catalysts or inhibitors that aid or hinder students when learning to program.

6.2 Analysing data of Dodona

6.2.1 Dodona

Before the approach to analyzing the data is explained, a look at the submission platform used is in order. The submission platform used in this context is Dodona, which is a submission platform for code. Dodona allows the instructor to define multiple series, each of which has a set of exercises that are presented to the students. An example of a couple series is shown in screenshot 6.1.

When code is submitted for an exercise, several test cases are run, and only if all test cases succeed is the submission marked as correct. other possible statuses are runtime error, compilation error, wrong,... The time of submission is also recorded.

The submission metadata can be exported into a csv file, which includes the exercise, series, id of student, status, and time. Using this data, several features can be computed for each student and each series that can be used as input to train a classification model.

6.2.2 Classification

Classification problems are a subset of machine learning problems that aim to predict the data class. This is accomplished by using a machine learning model, such as linear regression or support vector machines, that uses data to train the model to improve the accuracy of the predictions. Each machine

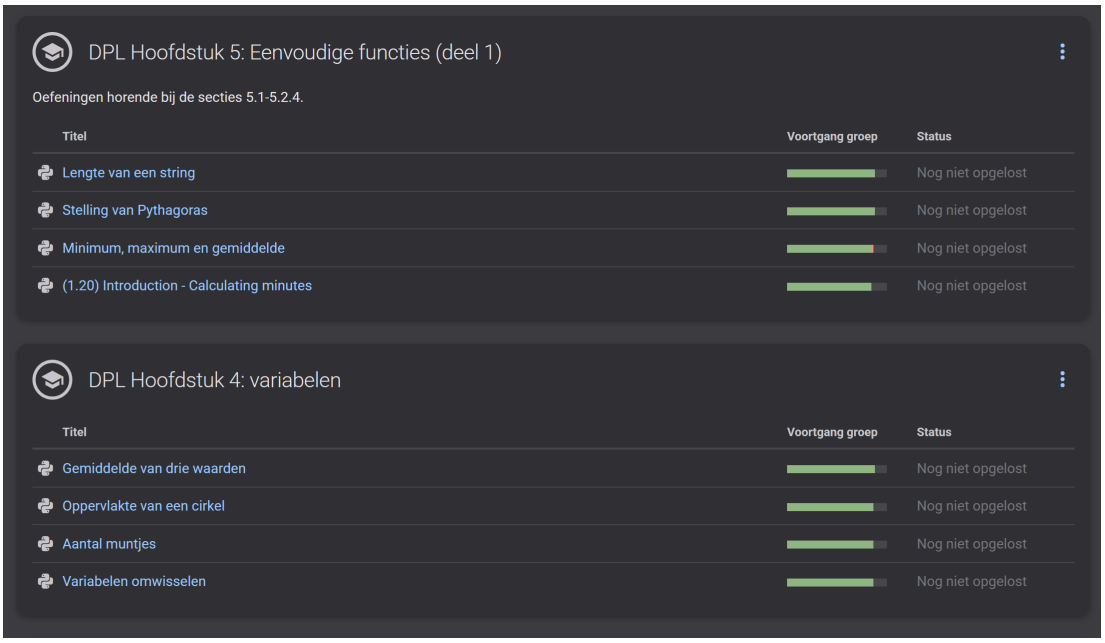

Figure 6.1: An overview of 2 different series on Dodona: "variabelen" and "Eenvoudige functies (deel1)", both of which have multiple exercises.

learning model has internal parameters that influence the classification process, and training refers to the process of optimizing these parameters such that the prediction is as accurate as possible for the training data presented. The model adapts to the training data presented and learns from it. To quantify the accuracy of the classifier, a loss function defines the distance between the actual class and the class predicted by the classifier for each sample.

The accuracy of the model is evaluated by keeping some of the data separate to evaluate the model. This set of data is referred to as the test set. It's important that the model isn't yet familiar with this data during training, because otherwise it would have trained itself to recognize these samples in particular.

6.2.3 Accuracy metrics

The most common metric used to quantify the performance of a classifier is the accuracy. The accuracy is defined as the number of correct predictions divided by the total number of predictions. It gives a general indication of the classifier's performance, but won't show everything. As an example, let's say that we have 100 samples that need to be classified, and 50 of them have class A and the other 50 have class B. If the classifier correctly classifies 25 of A and 25 of B, the accuracy will be $\frac{25+25}{50+50} = 0.5 = 50\%$. On the other hand, if the classifier is very biased towards class A, it might predict all samples to have class A. That would mean it correctly identifies the 50 samples of class A, but it mispredicts the 50 samples of class B. The accuracy in this case would be $\frac{50+0}{50+50} = 0.5 = 50\%$. Even though the accuracy is identical, the situations vastly differ from each other.

Different metrics can be used to gain more insight into the classification. Some definitions need to be established first to define these metrics. These definitions are in the context of binary classification, where there are only two possible classes a sample can have. One of these classes is given the label "positive" and the other is given the label "negative". It's irrelevant which class is given which label, as the labels themselves don't mean anything. A class prediction for the positive class can be either correct or wrong. A correct prediction is called a true positive (TP), while a false prediction is called a false negative (FN). The FN refers to the fact that the sample was predicted to be negative, but the sample was positive, making the negative prediction false. For a negative sample, true negatives (TN) and false positives (FP) are defined in a completely equivalent way. Figure 6.1 visualizes the confusion matrix, which shows the relation between the class of the sample, the predicted class, and the true or false positive or negative.

Actual / Predicted	Predicted Positive	Predicted Negative
Positive (Actual)	True Positive (TP)	False Negative (FN)
Negative (Actual)	False Positive (FP)	True Negative (TN)

Table 6.1: A table showing the confusion matrix, defining the relation between the class of the sample, the prediction made, and the true or false negative or positive.

These definitions allow the introduction of some new performance metrics. A more robust version of accuracy is called balanced accuracy, and it is defined as follows: $\frac{1}{2} * (\frac{TP}{(TP+FN)} + \frac{TN}{TN+FP})$. It gives equal importance to the ratio of true positives and total positive predictions and the ratio of true negatives and total negative predictions. If the data is unbalanced and there are a lot more positive or negative samples, it is possible that a model will predict the dominating class for all samples because this gives the best score. In this scenario, balanced accuracy will still be around 50%, because the dominated class will have very few true predictions. As an example, let's say 100 samples exist, 90 of them are positive samples, and the remaining 10 are negative samples. Let's also assume the model is biased towards the positives, and will predict positive for all samples. The accuracy will equal $\frac{90}{100} = 90\%$, even though the classifier is extremely basic. The balanced accuracy will equal $\frac{1}{2} * (\frac{90}{(90+0)} + \frac{0}{10+0}) = 50\%$. The balanced accuracy gives a more accurate indication of the model's performance, because it isn't making any useful predictions.

Two other important metrics are recall and precision. Recall is defined as the number of TP divided by the sum of TP and FN: recall $= \frac{TP}{TP+FN}$. If the amount of FN grows, the value of recall will decrease. This means that recall is an indication of the number of positives that are correctly identified. If there are 10 positives, and 8 of them are correctly identified, the recall will be $\frac{8}{8+2} = 80\%$.

Precision is defined as the number of true positives divided by the sum of true positives and false positives: precision $= \frac{TP}{TP+FP}$. IF the amount of false positives grows, the value of precision will decrease. This means that precision is an indicator of the number of positive predictions that are accurate. If there are 15 positive predictions, and 10 of them are TP, the other 5 are false negatives, then the precision will be $\frac{10}{10+5} = 66.6\%$.

Recall and precision are both important metrics. Recall measures the ratio of positives that are correctly identified. Precision measures the ratio of positive predictions for which the actual label is also positive. The F1 score is defined as the harmonic mean of precision and recall: F1 $= \frac{Precision * Recall}{Precision + Recall}$ $= \frac{2*TP}{2*TP+FP+FN}$. It is an important metric that indicates how well positives are detected, counting both positives that are undetected (FN) and negatives that are incorrectly counted as positives (FP).

For the tests that are to be performed, balanced accuracy and F1 score will both be used as performance metrics.

6.2.4 Approach to analyse the data

In this approach, the data from Dodona is used to predict whether a student will pass or fail. Several features are computed for the submission data, and those features are used as input for the machine learning prediction.

These are the features used to make the prediction:

- subm: numbers of submissions by student in series
- nosubm: number of exercises student did not submit any solutions for in series
- first_dl: time difference in seconds between student's first submission in series and deadline of series
- last_dl: time difference in seconds between student's last submission in series before deadline and deadline of series
- nr_dl: number of correct submissions in series by student before series' deadline
- correct: number of correct submissions in series by student
- after_correct: number of submissions by student after their first correct submission in the series
- before_correct: number of submissions by student before their first correct submission in the series

- time_series: time difference in seconds between the student's first and last submission in the series
- time_correct: time difference in seconds between the student's first submission in the series and their first correct submission in the series
- wrong: number of submissions by student in series with logical errors
- comp_error: number of submissions by student in series with compilation errors
- runtime_error: number of submissions by student in series with runtime errors
- correct_after_5m: number of exercises where first correct submission by student was made within five minutes after first submission
- correct_after_15m: number of exercises where first correct submission by student was made within fifteen minutes after firstsubmission
- correct_after_2h: number of exercises where first correct submission by student was made within two hours after first submission
- correct_after_24h: number of exercises where first correct submission by student was made within twenty-four hours after first submission

These features are computed separately for each student. For a specific student, each of these features is also present for every single series of exercises that exists on Dodona. For example, if a course has 18 series, then each individual feature will be present 18 times in the computed data, once for each series. There are 17 features in total, so the computed data would have 17 * 18 = 306 features in total for each student. It's important to note that only the submission results and submission behavior are taken into account. The actual code itself is not considered, only whether the code passes all tests, has a runtime error, has a compilation error,... and so on.

This approach generates snapshots at specific times during the course. All submissions before the time the snapshot is taken are counted, but any submissions afterwards will not contribute to that snapshot. Course A has specific deadlines for assignments at set moments during the course, so these deadlines were chosen as moments to take a snapshot at. There are four deadlines in total, including the exam gives 5 snapshots in total. The first 4 snapshots are made at the deadline of the 4 assignments, which are respectively at the end of week 1, 3, 4, and 7. The final snapshot is made on the day of the examination, which happens at week 9. Course B also has assignments, but the assignments aren't consistent over the years, making it impossible to compare the snapshots from multiple years. To mitigate this, a snapshot is taken after every week for course B, and the deadline-related features are removed from the features. Course B takes a full semester to complete, so 19 snapshots were taken in total.

The features can be computed for each student, and the resulting data can be used as the input for a machine learning model. There are many different kinds of machine learning that can be applied to the data. The method selected here is logistic regression, because logistic regression has a very straightforward interpretation of feature importance. Logistic regression has a coefficient that directly correlates with the importance of a feature. This coefficient also depends on the dimension of the feature, but if the data is normalized, the coefficient can be interpreted directly.

These features were used as input to train a logistic regression model. When training the model on data from a specific year, 80% of the data is used as training data and the remaining 20% is used as test data. A logistic regression model has several hyperparameters, which can influence the training process and thus the results of the classifier. Another choice that can influence the model is the split into training and test data. If this split is unbalanced, for example, if the test data has a lot more students who passed the course compared to the test data, then the results are likely to be poor. To mitigate this problem and to find the best hyperparameters for the model, grid search cross-validation was utilized. This is a combination of two different techniques to optimize model training, grid search and k-fold cross-validation.

Grid search needs possible values defined by the user for each hyperparameter. For each possible combination of the values for these hyperparameters, a model is trained with the given data. Afterwards, the accuracy is evaluated using the test data. The model with the best performance is selected and returned. By using this method, the optimal hyperparameters are selected by the model.

Multiple types of cross-validation exist, but only k-fold cross-validation is discussed here. K-fold cross-validation works by dividing the data into k pieces, where each piece has equal size. Each of these k

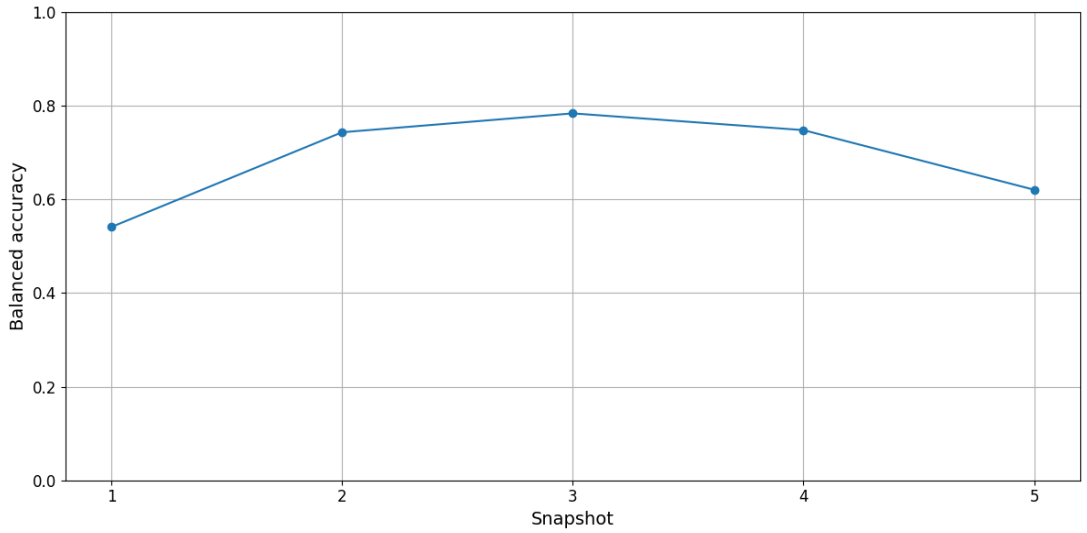

Figure 6.2: A line chart showing the evolution of the balanced accuracy over snapshots for course A in 2023.

pieces is used once as test data, while the other k-1 pieces are used as training data. This effectively means that k models are trained, and the final model is averaged across these k models. This provides a less biased model, as more variations of training and test data are used.

Cross-validation grid search combines these two approaches. First, the grid search will try each combination of hyperparameters, like a regular grid search. But instead of training a single model, the data is split into k pieces, and k-fold cross-validation is performed. The average of the k models is taken as the performance of the model, and grid search will take the best average for all possible combinations of the hyperparameters.

This whole process is run 20 times in total in order to minimize the randomness. The metrics used to evaluate the model, like balanced accuracy, for example, are then defined as the average of this metric across all 20 runs.

6.3 Context

The approach described in the previous section was applied to two different courses, course A and course B. Course A is the same course described in section 5.2.1.

Course B is similar to course A, both of them introduce the basics of programming in Python. However, course B is focused on a different public as it is part of the Bachelor's degree in commercial engineering and the Bachelor's degree in mathematics. Course A was primarily taken by students following the Bachelor's degree in computer science. Course B has an average of 70 students each course year, and the evaluation consists of an exam at the end of the semester that counts for 75% of the total mark, and 2 smaller exams during the semester that count for 25% of the total mark.

6.4 Results

6.4.1 Course A

As mentioned earlier, course A has 5 snapshots in total. The evolution of the balanced accuracy for the data for course A in 2023 is shown in Figure 6.2. The evolution of the F1 scores is shown in Figure 6.3.

The balanced accuracy starts out around 60%, increases to slightly less than 80%, and drops to a little bit above 60% for the final snapshot. This is counterintuitive, because students will have done more exercises and thus more data is present for the model. As the model has more data at its disposal in

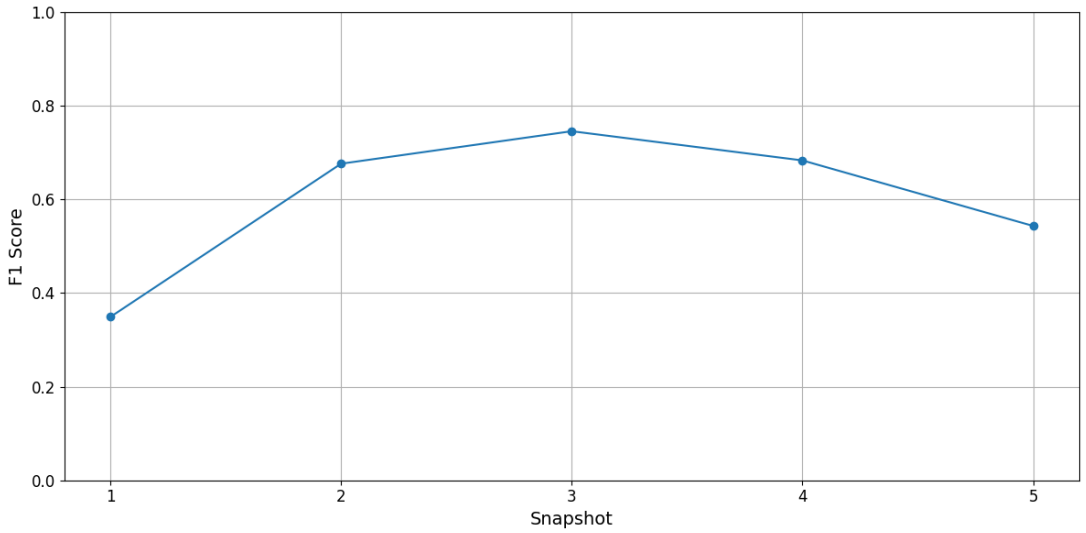

Figure 6.3: A line chart showing the evolution of the F1 score over snapshots for course A in 2023.

the latest snapshot. The expectation is that the performance should only improve as time increases. The F1 score follows a similar pattern to the balanced accuracy.

The evolution of the balanced accuracy for the data for course A in 2024 is shown in Figure 6.4 and the evolution of the F1 score is shown in Figure 6.5

The evolution of the balanced accuracy is significantly different in 2024 compared to 2023. The results are poorer, and the balanced accuracy stays around 60% for all snapshots. It is unclear what the underlying cause of this difference in performance is. There are no changes in the underlying course structure, and the number of students has stayed roughly the same. The students themselves are different, so a possible explanation would be that this class of students exhibits different behavior when solving exercises, which is more difficult to interpret for the logistic regression model. But this is only speculation, and the actual reason remains unclear. The F1 score follows a similar pattern to the balanced accuracy again.

6.4.2 Course B

For course B, no consistent deadlines over the years exist, so one snapshot was taken at the end of every week. In total, 19 snapshots were taken. The evolution of the balanced accuracy for the data for course A in 2022 is shown in Figure 6.6. The evolution of the F1 scores is shown in Figure 6.7.

An interesting observation can be made with these two charts. The balanced accuracy is on the rather low end, between 50% and 60%. However, the F1 score is very high, consistently above 80%. This seems to indicate very good performance with positives, but the balanced accuracy is rather low, so it seems that the negative predictions are poor. A manual check confirms this observation, as the model gives all samples a positive prediction. This is very undesirable behavior. This behavior was not present for course A, but it is present for course B, although the exact same method has been used. The logical conclusion here is that there is some difference between the data for course A and course B that makes this behavior appear.

A closer inspection reveals that the pass rate for course B is significantly higher than the pass rate for course A. Course B had 72 students in 2022, and 59 of them passed the course. The pass rate for course B in 2022 is $\frac{59}{72} = 0.819$. The pass rates for course A were 41.1% and 46.9% respectively. A manual review of the predictions revealed that the model trained for course B predicted almost only positives. The balanced accuracy for a model that only predicts positives is equal to $\frac{1}{2} * (\frac{59}{59+0} + \frac{0}{13+0}) = 50\%$. The F1 score for this model is equal to $\frac{2*52}{2*52+19+0} = 90.1\%$. This matches the observations made with the line chart. This is very undesirable behavior because a model that almost only predicts positives for given samples is not useful.

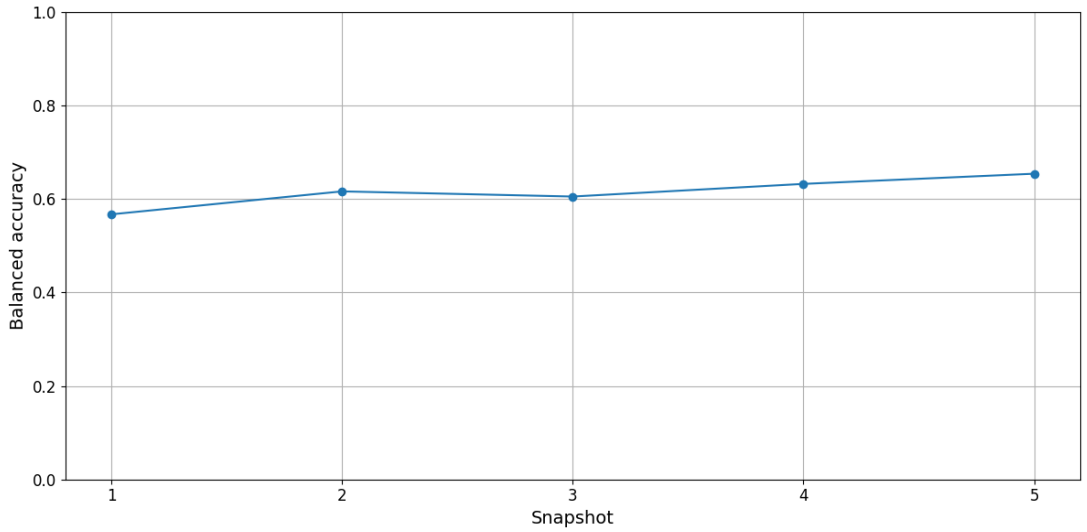

Figure 6.4: A line chart showing the evolution of the balanced accuracy over snapshots for course A in 2024.

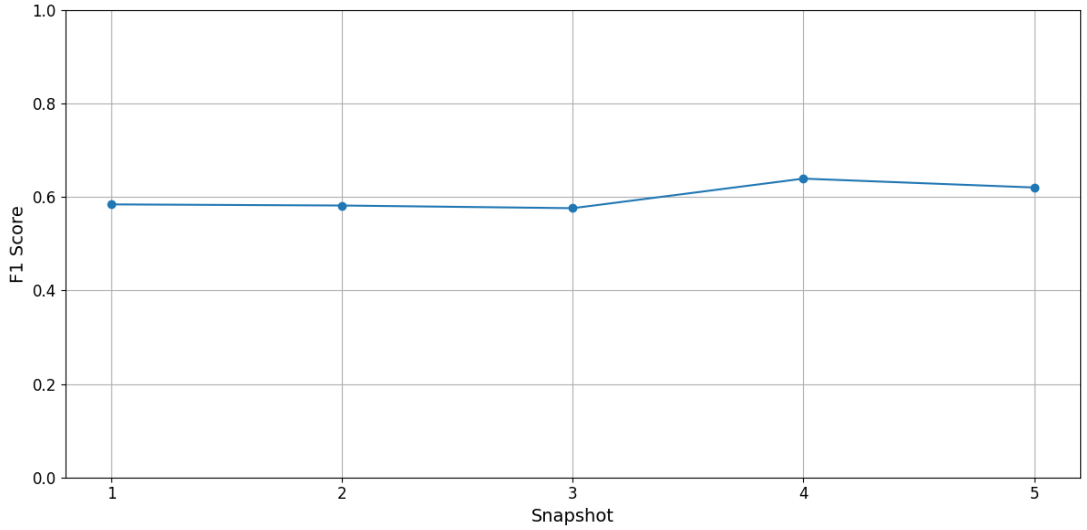

Figure 6.5: A line chart showing the evolution of the F1 score over snapshots for course A in 2024.

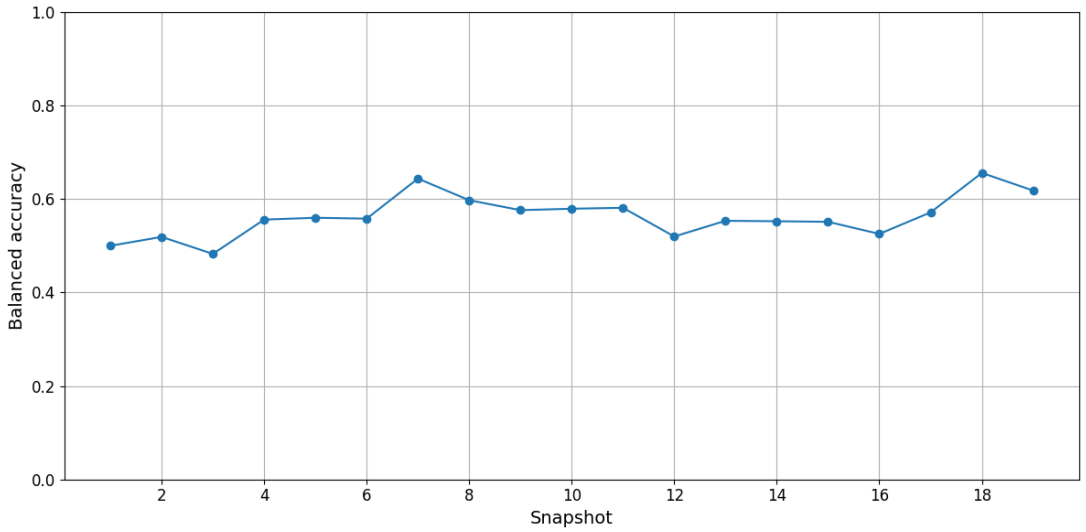

Figure 6.6: A line chart showing the evolution of the balanced accuracy over snapshots for course B in 2022.

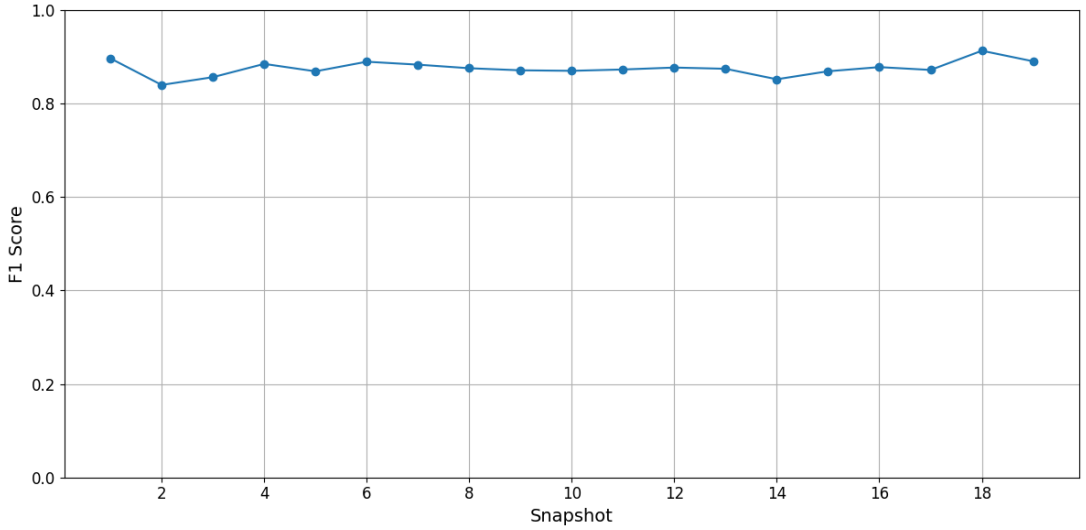

Figure 6.7: A line chart showing the evolution of the F1 score over snapshots for course B in 2022.

	No method	Oversampling	Undersampling	SVM	XGBoost (No GS)
2022	0.5896	0.7383	0.6369	0.5491	0.6663
2023	0.6326	0.6782	0.6336	0.5333	0.7741
2024	0.5832	0.7113	0.5398	0.5686	0.5641

Figure 6.8: A table showing the average of the balanced accuracy for each different method across all snapshots for the years 2022, 2023, and 2024

In order to fix this problem, there are several options that can be applied. The core of the problem is the unbalanced distribution of data classes. Getting more data isn't an option, so we have to apply a method that deals with the available data. Three possible methods that can be applied are the following:

- Oversampling: To artificially get more representatives of the minority class, we can duplicate some samples of this class. Because there are more samples of the minority class, the distribution of the data becomes more even. But because we duplicate existing data, we risk overtraining the model.
- Undersampling: Instead of getting more samples of the dominated class, another possibility is to eliminate some samples of the majority class. This will also even the spread, but there will be less data available to work with.
- Model parameters: Some machine learning models allow for associating a weight with each class. By increasing the weight for the minority class, we can force the model to give more importance to the dominated class, thereby hopefully getting a more balanced model.

It is impossible to predict which method will be the most effective for this particular context, so all three of these methods were applied. Because logistic regression does not allow for specifying a weight for each class, two different machine learning models were used that do allow this. Support vector machines (SVM) and XGBoost were chosen because both of these models allow for the specification of class weights. However, grid search was not applied for XGBoost due to performance constraints. The training phase of XGBoost takes an extremely long time compared to the other methods. Cross-validation was still applied to this method. Logistic regression is used for the methods of undersampling and oversampling because interpreting the coefficients for this model is the most straightforward. These methods were applied in the years 2022, 2023, and 2024 for course B. Due to space constraints and because this thesis would otherwise get very convoluted, only the average balanced accuracy and F1 score are offered instead of the full evolution, as for course A. To give an overview of the performance of different methods, the average balanced accuracy across all snapshots for each method across the years 2022, 2023, and 2024 is presented in Figure 6.8.

From this table, it seems that oversampling is the best method to use in this specific context. The evolution of the balanced accuracy over the snapshots for 2022, 2023, and 2024 is presented in Figures 6.9, 6.10, and 6.11.

The three line charts show a similar pattern. At first, the balanced accuracy starts out around 50%, but improves in later snapshots. The maximal balanced accuracy is always achieved around the middle of the course, and the balanced accuracy afterwards stagnates or even decreases a little. The danger of the oversampling method is that the model becomes overfitted due to the repetition of data. It is possible this model is overfitted as well, but the balanced accuracy is lower at the beginning, which seems to indicate that, at the very least, no overfitting has occurred for the early snapshots.

The F1 score for these years is shown in Figures 6.12, 6.13, and 6.14.

The F1 score remains almost constant at around 80% for all of the years. This is a slight decrease compared to when no oversampling was performed, but the balanced accuracy has increased significantly.

6.4.3 Interpretation of the coefficients

Training a model on the data of a single year doesn't help predict the future performance of students, but it can give more insight into the factors that influence the passing or failing of students. By

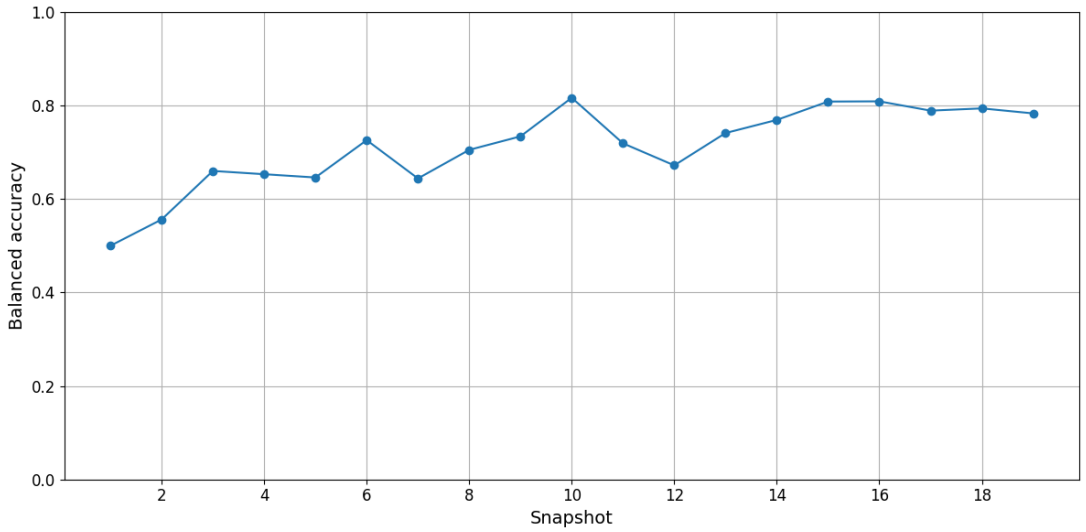

Figure 6.9: A line chart showing the evolution of the Balanced accuracy over snapshots for course B in 2022 when oversampling is used.

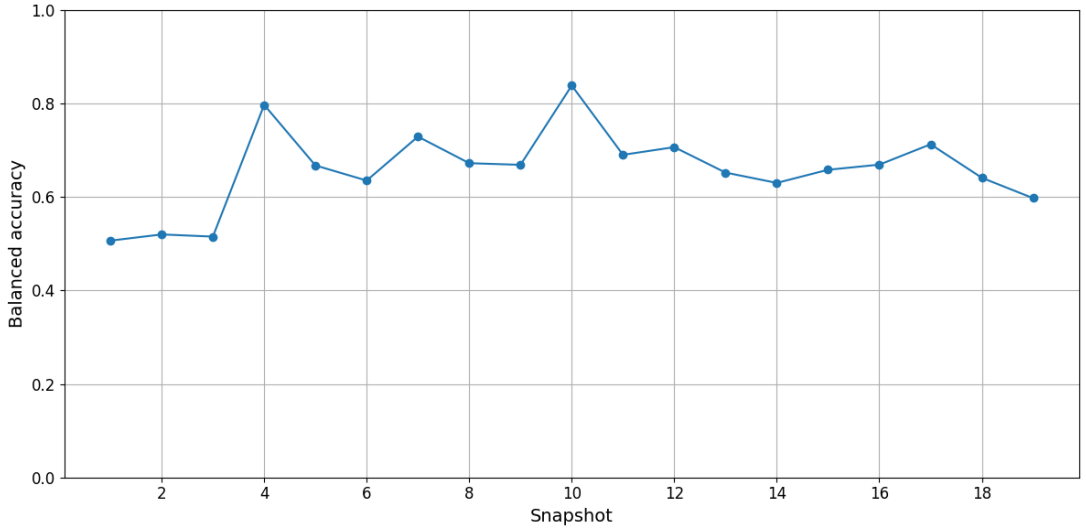

Figure 6.10: A line chart showing the evolution of the Balanced accuracy over snapshots for course B in 2023 when oversampling is used.

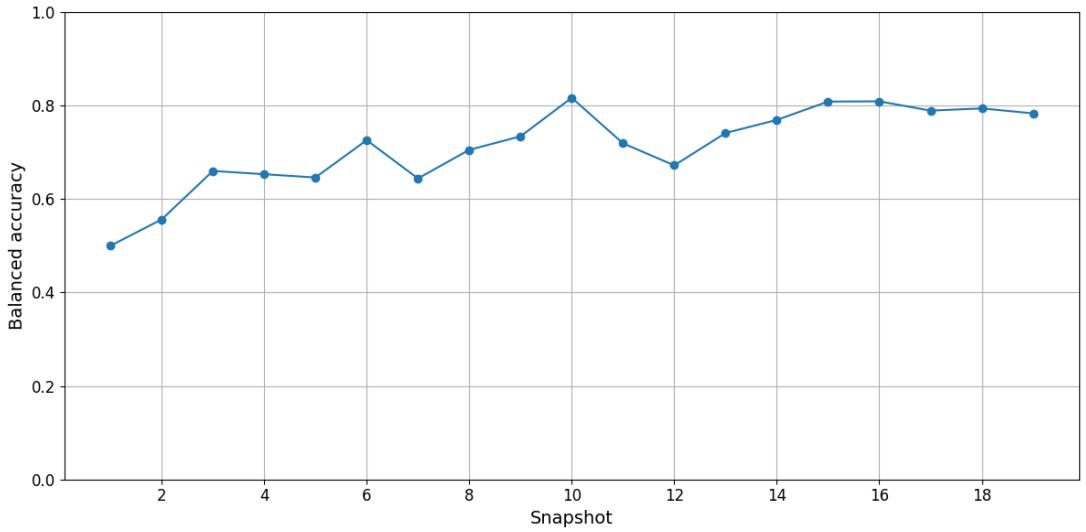

Figure 6.11: A line chart showing the evolution of the Balanced accuracy over snapshots for course B in 2024 when oversampling is used.

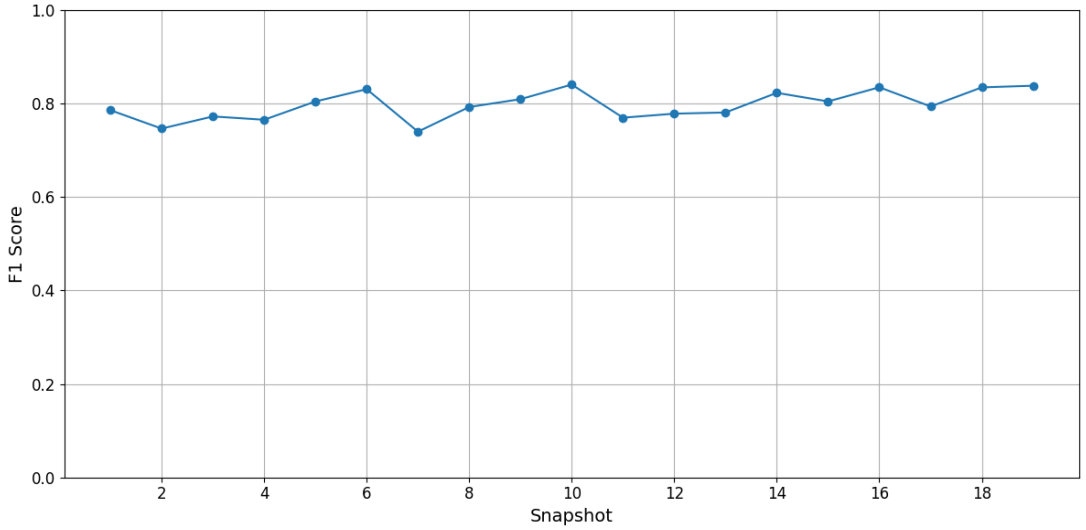

Figure 6.12: A line chart showing the evolution of the F1 score over snapshots for course B in 2022 when oversampling is used.

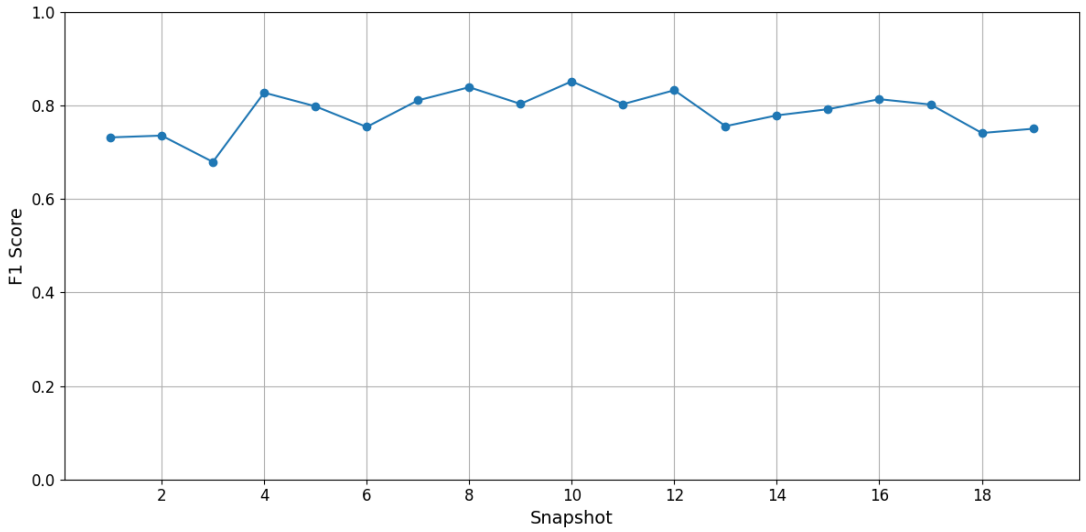

Figure 6.13: A line chart showing the evolution of the F1 score over snapshots for course B in 2023 when oversampling is used.

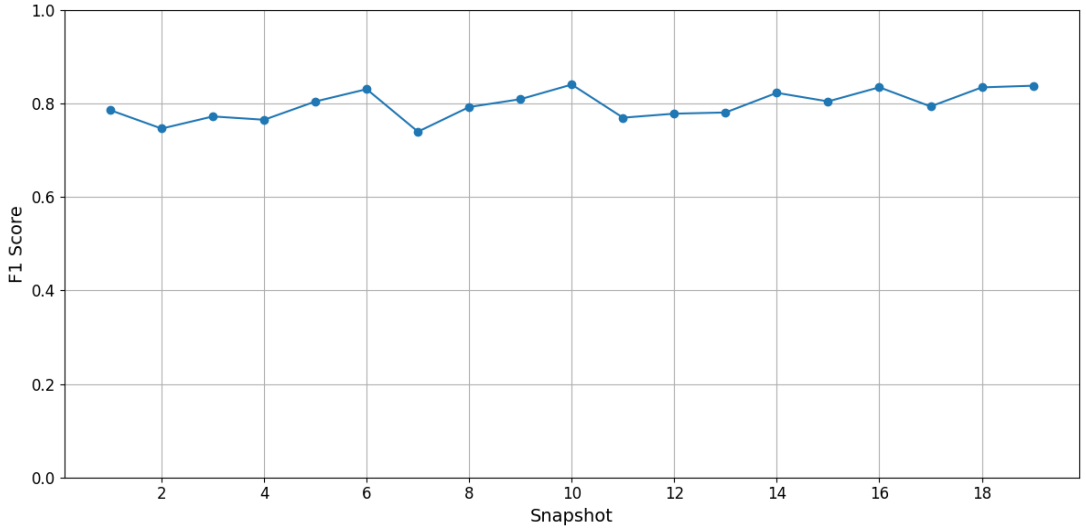

Figure 6.14: A line chart showing the evolution of the F1 score over snapshots for course B in 2024 when oversampling is used.

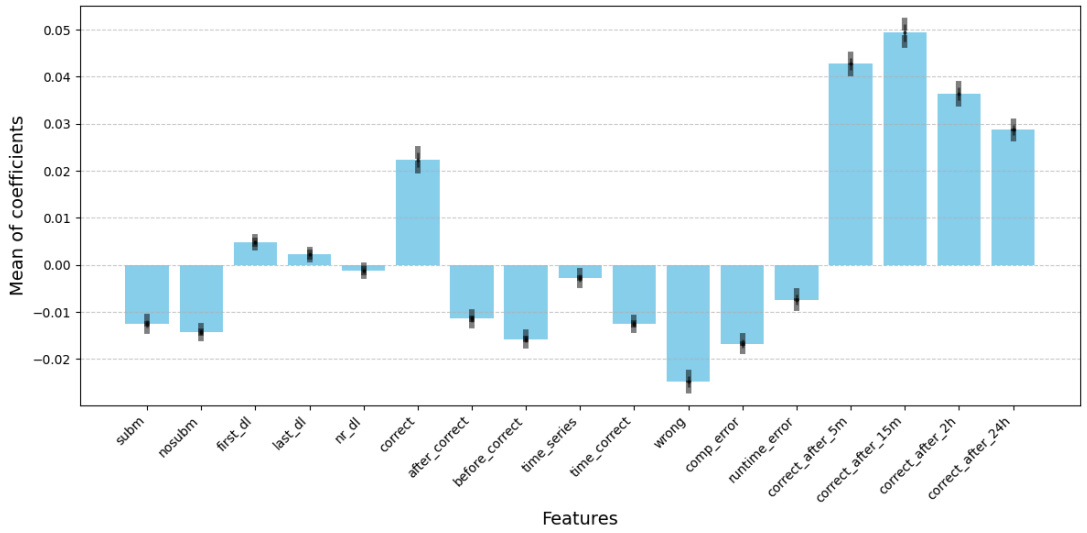

Figure 6.15: A bar chart showing the average of the coefficients of the logistic regression model trained on the data of course A from 2023.

analyzing the importance of the features that are given to the machine learning model, the most important features can be determined and interpreted to increase understanding of potential catalysts or inhibitors for students.

Logistic regression is the model that is chosen to interpret, because it provides the clearest interpretation. Logistic regression provides a coefficient for each feature that is input into the model, and the value of these coefficients can be compared to determine the most important features. But there is one issue: the dimension of the feature also plays a role. If a feature is distributed between 1 and 10, its coefficient will be on a different magnitude compared to a feature that is distributed between 10000 and 100000, even if the features themselves have equal importance. To mitigate this issue, standardization is performed first.

The individual feature values are normalized before they are input into the model. The values of each feature are converted into Z scores. A z-score is defined as follows: for a given value x, the mean of all values $\bar{x}$, and standard deviation σ_x is used to normalize the value of x. The new value of x will be $\frac{x-\bar{x}}{\sigma_x}$. This produces a normalized z-score, regardless of the dimension of the original data. This normalization is performed for each singular value of a feature. Afterwards, the logistic regression model is trained as usual. This model was trained for 2023 and 2024 for course A, and 2022, 2023, and 2024 for course B.

Each feature is present many times in the data, once for each series. This makes the interpretation of the importance of specific features difficult, as there exist many variations of this feature with possibly different values. To mitigate this, the average of each feature across all series is computed. Bar charts showing the average coefficients for these models are shown in Figures 6.15, 6.16, 6.17, 6.18, and 6.19.

These bar charts show the importance of each feature. Note that the deadline-related features are omitted for course B because there are no consistent deadlines for this course. For many of the features, the importance is inconsistent and varies wildly. As an example of this, consider “correct“ in Figure 6.16 and Figure 6.17. For course B in the year 2022, “correct“ has a significant negative influence on the prediction. But for the same course in the year 2023, “correct“ has a large positive influence. This makes the interpretation of the importance of these features difficult. The fact that such a large variation exists for all the features could be an indicator that the results themselves are inconsistent.

Because the data was normalized, it’s possible that the performance of the model has changed with regard to the unedited data. A look at the balanced accuracy results reveals that the balanced accuracy is significantly higher, starting at 73% for course A in 2023, for example. The first snapshot is taken

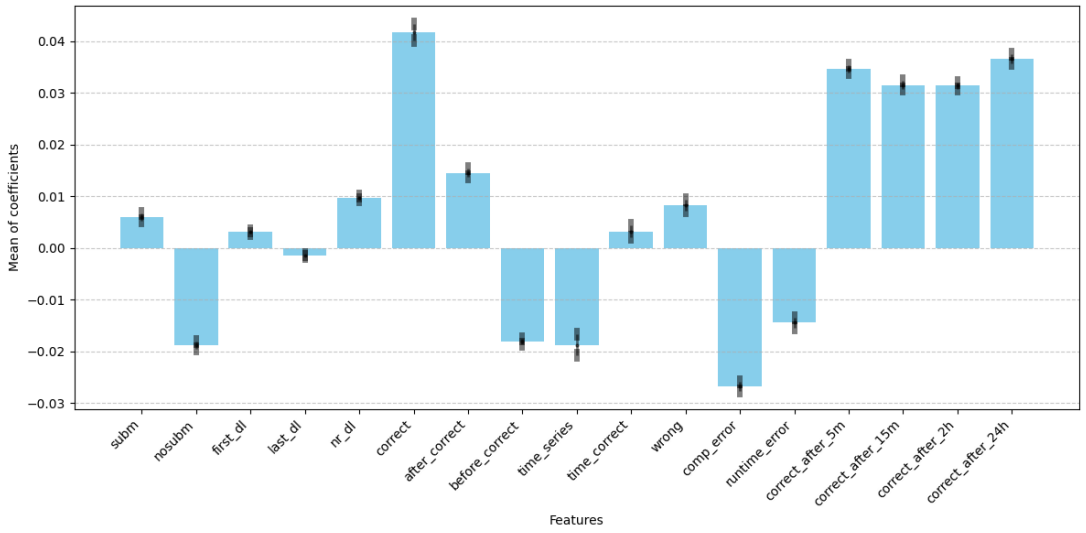

Figure 6.16: A bar chart showing the average of the coefficients of the logistic regression model trained on the data of course A from 2024.

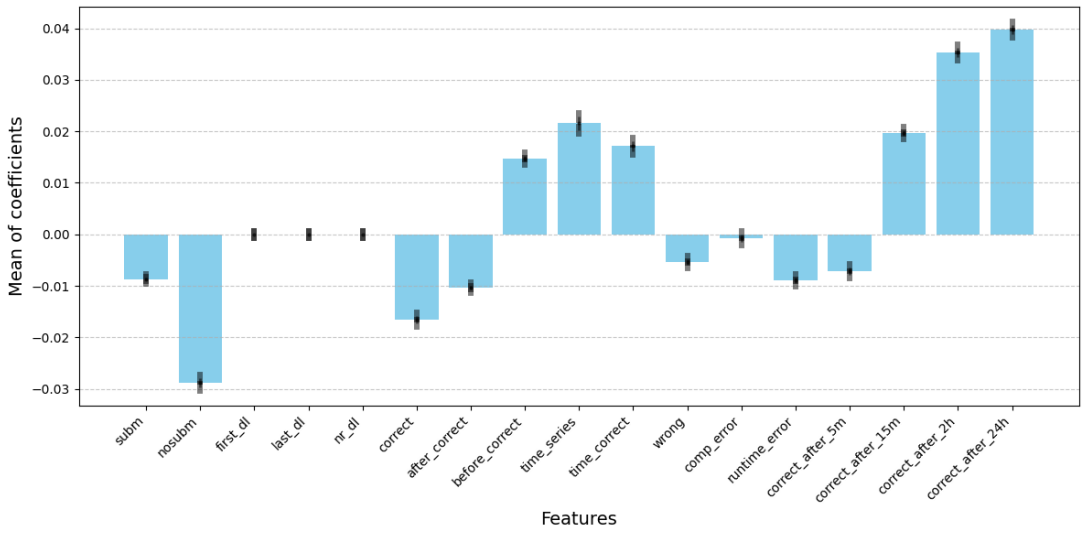

Figure 6.17: A bar chart showing the average of the coefficients of the logistic regression model trained on the data of course B from 2022.

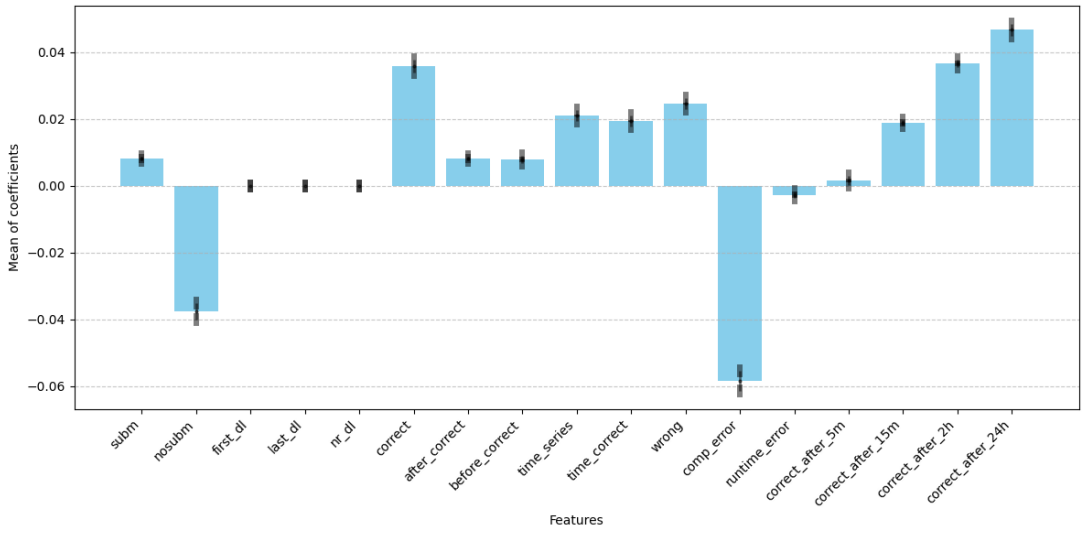

Figure 6.18: A bar chart showing the average of the coefficients of the logistic regression model trained on the data of course B from 2023.

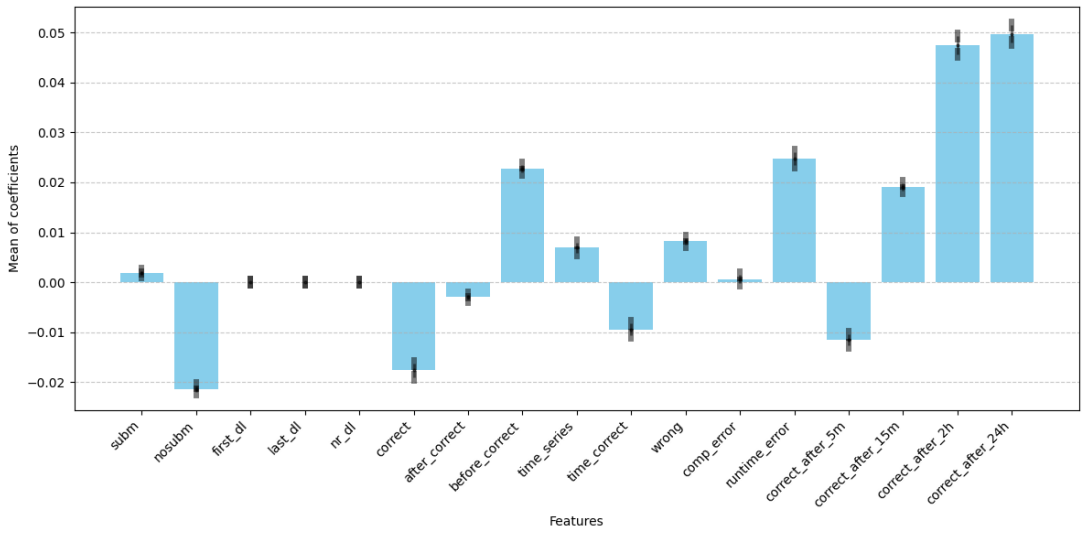

Figure 6.19: A bar chart showing the average of the coefficients of the logistic regression model trained on the data of course B from 2024.

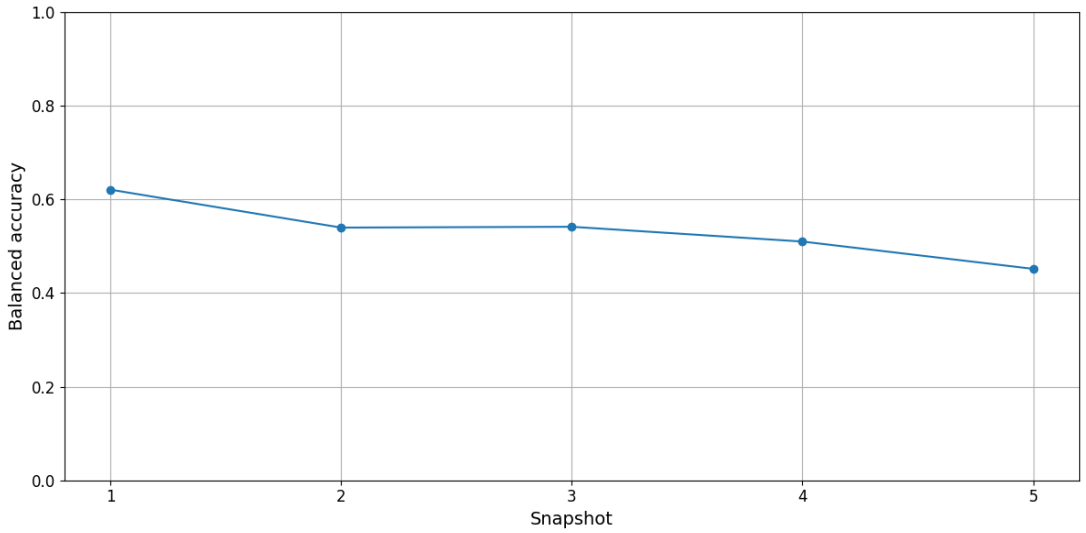

Figure 6.20: A line chart showing the evolution of the balanced accuracy over the snapshots when the training data is from 2023 and the test data is from 2024.

only a couple of days after the start of the semester, and it seems highly unlikely that such an accurate prediction can be made this early into the semester. It is most likely that the model is severely overfitted due to the normalization of the data, which also makes the value of the coefficients doubtful. This can also explain why the coefficients themselves vary wildly across different years and across different courses.

However, not all of the coefficients vary wildly. The "nosubm" feature has a consistent negative influence on the prediction. This feature is the number of exercises for which no attempt was submitted. This seems to indicate that, at the very least, attempting exercises instead of skipping them has a very negative influence on the odds of passing a course. Other features that have a consistent value are the "correct_after_x" features. These features are equal to the number of exercises for which a correct submission exists that is submitted within the time mentioned in the feature after the first submission for that exercise. The sole exception to the rule that these features are important, is"correct_after_5m". This feature has a large coefficient for course A, but a small or negative coefficient for course B. The fact that these features have a large coefficient means they have a significant positive influence on the prediction. This suggests that being able to solve an exercise within a reasonable time period after first starting the exercise is a good predictor for passing the course.

However, this interpretation still has to be taken with a grain of salt. The other coefficients still vary wildly, and it could very well be a coincidence that these features appear to have a consistent coefficient. The fact that the model is very likely to overfit makes the results even less reliable, because overfit models focus in great detail on the data presented to them, being unable to identify global patterns that exist in the data.

6.4.4 Predicting future years

The previous predictions only used training and test data for the same year. But the main utilization of this approach is to predict the performance for future students, which requires building a model that is based on data from one or more years, and predicting pass or fail for students from a different year. To simulate this, the data from 2023 was used as training data, and the data for 2024 was used as test data. The results of this approach are shown in Figure 6.20 and 6.21.

The balanced accuracy starts out around 60% like the predictions for 2023 and 2024, but decreases with later snapshots. This seems to indicate that the predictions later in the course are less accurate than the very first prediction, which seems extremely unlikely. The first snapshot is made at the end of the first week, and being able to accurately predict whether a student will pass or fail at this time seems highly unlikely. This seems to indicate that the model trained with the data from 2023 is an

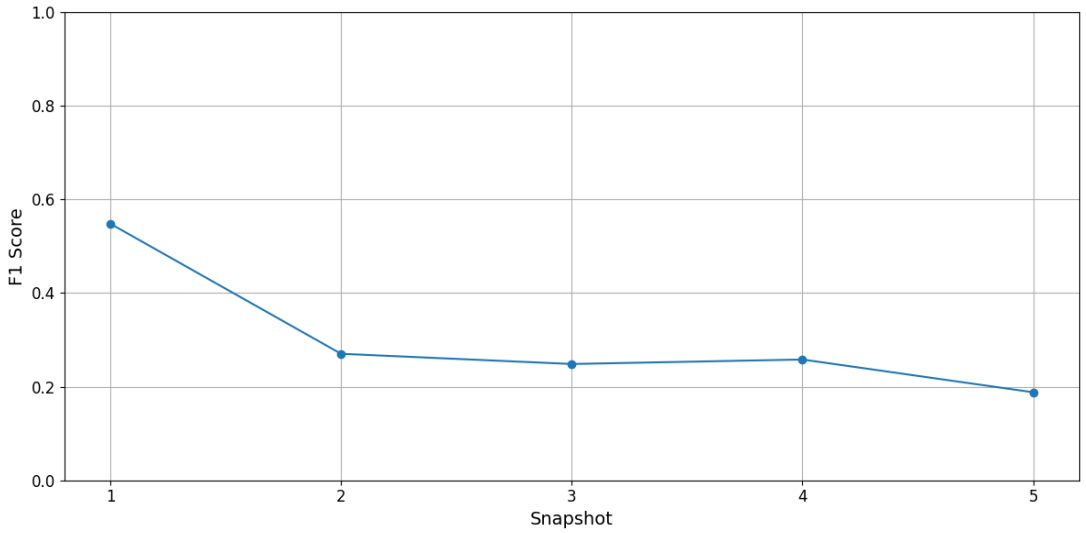

Figure 6.21: A line chart showing the evolution of the F1 score over the snapshots when the training data is from 2023 and the test data is from 2024.

No method	Oversampling	Undersampling	SVM	XGBoost (No GS)
0.0.5100	0.0.5373	0.5298	0.5253	0.6135

Figure 6.22: A table showing the average of the balanced accuracy for each different method when the data for the years 2022 and 2023 is used as training data for the model to predict 2024.

unreliable estimator for the performance of students in 2024. The F1 score is even lower than the balanced accuracy in this case.

Like with course A, the main goal is to predict the performance of future students. To simulate this, the snapshots from 2022 and 2023 are combined and used as training data for a model to predict the snapshot of 2024. As with the approach for a single year, it is unknown which method aimed at mitigating the class imbalance will have the best performance here. Once again, all methods are tried and the results are shown in Figure 6.22.

All of these predictions perform very poorly. Only XGBoost has a decent balanced accuracy. There is no method that achieves good performance when predicting the passing of students in 2024 when using the data of 2022 and 2023 as input.

6.5 Discussion

Overall, the results of this approach to analyze the submission data of students show worse results than the original thesis in which this approach was used[Van24]. Most noticeably, the prediction of whether students will pass in future years based on the data of previous years performs significantly better. The balanced accuracy in the original work starts at around 60% for both courses the method was tested on, and climbs to 80% towards the end of the course. The balanced accuracy in the context of courses A and B stays around 50%, which is no better than a random classifier.

The approach in the original thesis was tested across two different courses, each with a different structure. Additionally, a replication study was performed at a different university in Finland, where similar results were achieved [Zhi+24]. This seems to indicate that this approach should be generalizable in different contexts, but the results in this particular context seem to be subpar when compared to previous studies.

One of the major differences in this context compared to the previous studies is the length of the course. Course B is a full semester course, but course A only takes half a semester, and finishes within

10 weeks due to a special system employed at Hasselt University. This only gives half the time to make predictions compared to full-semester courses, which is a big difference in context. Another major difference is the number of students enrolled in the course. Course A has around 55 students each year, and course B has around 70 students each year. In the original thesis, the average number of students enrolled in the courses was 280 and 390, respectively. This is a massive difference in size, and also a massive difference in data. A course with a larger number of students has significantly more training data to work with to fine-tune the model.

The conclusion that can be drawn here is that the approach to analyze the submission data of students is not effective in courses that have a smaller number of students because there isn't enough training data to work with.

Chapter 7

Course additions

7.1 Introduction

This chapter describes the additions that were made to a CS1-level computer science course. This is the same course that was described in section 5.2.1. It is the course for which the survey was administered and one of the two courses on which the educational data analysis was performed. This course was evaluated based on the recommendations on the literature and several additions were made to improve this course. The course already has a good variety of exercises, but parsons puzzles and worked examples were missing from it. Parsons puzzles are good exercises for absolute novices because they take away the burden of having to write code from scratch, and worked examples are important to show the problem-solving process that should be followed when solving an exercise.

Additionally, a document describing the different roles of variables, with code example showing a variable used in the different roles was also appended to the course. This document is intended to help students reason about how variables can be used in an algorithm they want to use or implement.

7.2 Course setup

The context of the course has already been described in section 5.2.1, but more clarification is needed on the type of exercises present in this course. As mentioned in the chapter about educational data analysis, the course uses the learning platform Dodona. The type of exercises on this learning platform are all coding exercises, where the student is tasked with writing code that solves a given problem specification. The input and output of this problem are defined, and the students have to write the code that transforms the given input into the output.

Apart from these exercises, there is a study guide that contains many more types of exercises. These exercises include predicting the value of expressions, rewriting pieces of code, debugging, summarization of code, and predicting the output of a program. There are many different types of exercises, providing a balanced curriculum for the students. However, two important parts described in the literature were missing. One of them is Parsons puzzles, as described in section 3.4.1. Parsons puzzles are a good type of exercise for the very start of a subject, when the students don't have the necessary qualities to write longer pieces of code themselves. Rearranging lines of code into the correct order trains algorithmic thinking without placing the burden of writing the code from scratch on the students.

Another important part that is missing is worked examples. Worked examples show the solution process for an exercise in detail, including bugs and solution attempts that didn't lead anywhere. It shows a realistic solution process, without giving the illusion to students that they should be able to solve an exercise on the first attempt. The detailed problem-solving process described gives students the opportunity to learn a problem-solving model for themselves and apply it the next time they solve an exercise.

```
print("Hello", name, "nice to meet you!")
name = input("What is your name?")
```

Figure 7.1: The first Parsons problem presented to students, for a simple hello program.

```
price = float(input("How much needs to be paid?"))
print("The return is", return)
paid2 = float(input("How much did person 2 pay?"))
paid1 = float(input("How much did person 1 pay?"))
return = paid1 + paid2 - price
```

Figure 7.2: A parsons puzzle for a problem that calculates the amount of return after 2 people pay a bill.

7.3 Parsons puzzles

The Parsons puzzles start out very easy, as they are aimed at beginning students. This is also why the first three lessons contain the most Parsons puzzles. They are intended to show students how code works and train them in algorithmic thinking by placing the code in the correct order. Later on in the subject, the number of puzzles per lesson decreases, and they are more intended to show the new concepts learned that lesson and to make the student familiar with them without having to write a piece of code that uses these new concepts themselves. The first Parsons puzzle asks students to place only 2 lines in the correct order to write a simple hello program. The exercise is shown in Figure 7.1.

Three more Parsons puzzles follow after this. The last Parsons puzzle for the first lesson is one that calculates the amount of change to be returned. Because conditions haven't been taught yet, it's not possible to use them yet. This exercise is shown in Figure 7.2.

The second lesson starts off with a Parsons puzzle that introduces conditions. The exercise is shown in Figure 7.3.

There are three more Parsons puzzles for this lesson. The last Parsons puzzle for lesson two is for a more advanced version of the last problem of lesson one. Instead of calculating the change, conditions need to be used to calculate if the total amount paid is too much, exact, or too little.

This Parsons puzzle is significantly more advanced than the previous Parsons puzzle. The difficulty of the puzzles gradually increases. Puzzles later in the subject are more difficult to give the student the chance to progress.

Starting from lesson four, the number of Parsons puzzles per lesson gradually decreases as students move more towards writing code themselves. The goal of the Parsons puzzles also shifts a little. The focus of the puzzles is now on serving as examples for new concepts learned because it is assumed that students have been adequately trained in algorithmic thinking.

An example of a Parsons puzzle that serves as an introduction is shown in Figure 7.5. This program calculates the factorial of a number. It is not a complicated program, but it serves as an example of a function that takes an input and produces an output.

Another Parsons puzzle exists for this lesson is shown in Figure 7.6. The goal of this puzzle is to show how functions can be nested and how a larger program can be split into multiple functions to better

```
number = int(input("What's the number?"))
print("The number is even")
if number % 2 == 0:
else:
print("The number is odd")
```

Figure 7.3: A Parsons problem that introduces conditionals, the program calculates I a given number is even or odd.

elif total_paid < price:
print("The return is", over_pay)
paid2 = float(input("How much did person 2 pay?"))
price = float(input("How much needs to be paid?"))
print("The pay is exact!")
print("The pay is", under_pay, "too little")
under_pay = price - total_paid
total_paid = paid1 + paid2
if total_paid > price:
over_pay = total_pay - price
paid1 = float(input("How much did person 1 pay?"))
else:

Figure 7.4: A Parsons problem that calculates if two people paid enough, too much, or too little for a bill.

return fac
for i in range(2,x+1):
fac = 1
fac = fac * i
def calculate_factorial(x):

Figure 7.5: A Parsons problem calculates the factorial of a number with a function.

separate responsibilities.

7.4 Worked examples

Worked examples were added to the subject as well to showcase the full problem-solving process when solving an exercise. There are four exercises for which a full worked example was developed. The first exercise is about trilateration, or calculating the relative position of circles. The second exercise is about the sum of squares, where two numbers m and n are given as input and all numbers a have to be listed as output for which two numbers x and y exist such that $x^2 + y^2 = a$. The third exercise is about simulating Conway's Game of Life. Three functions are specified and have to be implemented so that a simulation of the Game of Life can be run. The last exercise is about calculating the shared surface of two intersecting rectangles.

Each of those examples are solved in a common way. First, the exercise itself is presented and explained thoroughly, to indicate that the first step of any solution is to understand what exactly the problem is. Second, before attempting to find any solutions or algorithms for the problem, some examples are manually solved to get a feel how this problem can be solved. Afterwards, a formal algorithm is constructed, and code is written that implements this algorithm. Finally, the code is fully written and extensively tested with multiple test cases that try to be diverse and have a full coverage of the program. After the code has been deemed functional, it is reflected upon to ascertain the quality and make potential improvements or identify alternative solutions.

This is a good way of solving exercises, and each of these steps is explicitly labeled in each worked

factorial = calculate_factorial(i)
def sum_of_factorials(n):
print("The sum is", sum)
for i in range(n+1):
sum += factorial
sum = 0

Figure 7.6: A Parsons problem that calculates the sum of factorials.

Name	Condition
concentric	$d = 0$
inner touching	$d = \|r_1 - r_2\|$
outer touching	$d = r_1 + r_2$
enclosing	$d < \|r_1 - r_2\|$
intersecting	$\|r_1 - r_2\| < d < \|r_1 + r_2\|$
seperated	$d > r_1 + r_2$

Table 7.1: The name and condition for all possible relative positions.

```
treshold = pow(10, -6)
d = sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2))
if d < treshold:
    print("concentric")
elif -treshold < d - abs(r1 - r2) and d - abs(r1 - r2) < treshold:
    print("inner touching")
elif -treshold < d - (r1 + r2) and d - (r1 + r2) < treshold:
    print("outer touching")
elif d < abs(r1 - r2):
    print("enclosing")
elif abs(r1 - r2) < d and d $<$ abs(r1 + r2):
    print("intersecting")
elif d > r1 + r2:
    print("seperated")
else:
    print("something else")
```

Figure 7.7: Code that solves the trilateration problem, using the conditions described in Figure 7.1.

example so students can learn from this approach.

7.4.1 Trilateration

For the trilateration exercise, the goal is to determine the relative position of circles. The parameters given as input are the radius and center coordinates of both circles. For this particular exercise, the conditions that specify the relative position of the circles are given. These are shown in Figure 7.1. This exercise was chosen as a worked example because it can be shown early on in the subject, only needing the understanding of conditions. It is also a good example to show the general steps taken to convert a specification into code because the conditions themselves are already given.

These conditions themselves have to be converted into code, which is described in the worked example. The problem statement also specifies that because we are working with floating point numbers, rounding errors should fall within a certain threshold for two numbers to be equal. The full process of converting these specifications is shown in the worked example, but we only show the final code minus the input here in Figure 7.7.

During the development of this code, several bugs were found. The initial code that contained these bugs and the discovery of the bugs is also in the worked example because it is also part of the development process.

7.4.2 Sum of squares

Sum of squares is an exercise where two numbers m and n are given as input, and all numbers a have to be found such that two numbers x and y exist such that $x^2 + y^2 = a$. A condition for the solution is that $x <= y$ must hold. Unlike the previous worked example, no condition or algorithm is given to solve this exercise. The solution for this exercise is not trivial but also not too difficult, which makes

```
x = 0
while x**2 + x**2 <= n:
    print("x=",x)
    y = x
    while x**2 + y**2 > n:
        print("y=", y)
        if m <= x**2 + y**2 and x**2 + y**2 <= n:
            print("x=", x, "y=", y, "is a solution!")
        y = y + 1
    x = x + 1
```

Figure 7.8: Code that solves the sum of two squares problem by listing all possible pairs of numbers.

it a good exercise to create a worked example for.

First, some manual examples are tried with the numbers $m = 4$ and $n = 6$. On sight, the solutions $0^2 + 2^2 = 4$ and $1^2 + 2^2 = 5$ can be found, but this is not an algorithm. A larger example is then attempted with $m = 85$ and $n = 87$. It is no longer possible to tell on sight which numbers might be a solution for this choice of m and n, so to manually solve this I start by listing every possible combination of numbers and check if the sum of their squares is between 85 and 87. This approach is then later converted into an algorithm that uses a double for loop, and the approach to convert this is described into more detail in the worked example. This eventually results in the code shown in Figure 7.8.

This code works by setting x and y to 0 and incrementing y until $x^2 + y^2 < n$. The full development of code is shown in the worked example, including the explanation why $x^2 + x^2$ and $x^2 + y^2$ are the conditions for the while loop. The first condition needs to be false when no more pairs can be found, which occurs when $x^2 + x^2$ is larger than the largest boundary n.

This code is then thoroughly tested with multiple examples, and manually verified that all identified solutions are correct and that no solutions exist that were not found. During the testing phase, two bugs were found once again and the identifying of the source and the fixing of these bugs is also described in the worked example.

During the reflection, I reflect on the code and notice that $x^2 + y^2$ is an expression that is often used in the code, so the variable sum is introduced and initialized with this value to replace all occurrences of this expression.

7.4.3 Game of life

Conway's Game of Life is not a typical game, it is more of a survival simulation that takes place on a 2D grid where each cell can be dead or alive at a certain point. The next generation is then computed from the current generation of cells, and cells can die or come to life depending on the amount of neighbors. The problem statement defines three functions that have to be implemented. The first function simply prints out the current game state, the second function calculates the number of neighbors for each cell and the third function calculates the next generation given a game state. This worked example is aimed to show how to solve larger exercises where you have to write not just a single function that can solve the problem.

The first function is just printing of a given game state and developing this function goes smoothly. The second function is more difficult to implement. Each cell has maximum 9 neighbors, but cells on the edge or at the corner of the board have less neighbors, so it is important to pay attention to these edge cases and make sure the index used to access the nested list remains in bounds.

This takes a few attempts, first the index is too large which throws an error. This bug is easy to identify because of the error which gives an explicit notification that is wrong. After fixing this error, a bug is also present where the index drops below 0, but in python this is interpreted as len(list) - 1, so no error is thrown but the behavior of the function is incorrect. After identifying that something is

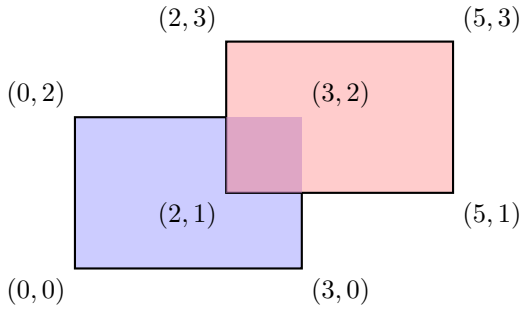

Figure 7.11: An example of two intersecting rectangles, where the bottom left corner of the overlap is (2,1) and the top right corner of the overlap is (3, 2).

wrong by testing the function, this bug has to be discovered and eventually fixed. The full process is documented in the worked example and eventually leads to the code shown in Figure 7.9.

These are two helper functions that help the function specified by the problem statement that needs to compute the number of neighbors for each cell. The importance of splitting code into multiple functions is also explained in the worked example. Even though these functions are not explicitly requested by the exercise, it is a good idea to have them.

Finally the last function is developed that calculates the next generation based on a given game state. The code for this function is shown in Figure 7.10.

Once again, only function is required by the exercise but two other helper functions were added to separate responsibilities and avoid large, unreadable functions. After this code was written, extensive testing was once again performed, and no bugs were found this time.

Reflection on the code written was performed as well, which revealed that the function that computes the amount of neighbors for a single cell has many repetitive lines of code. It could be written more elegantly by looping over a set of indexes instead of repeating the same lines of code with different indexes.

7.4.4 Intersecting rectangles

The last exercise is part of a larger exercise that asks for a class to be written that represents a rectangle. The rectangle class has the bottom left corner, the width and the height of the rectangle as required parameters for the constructors as these parameters uniquely identify a rectangle. The exercise is to find the overlap between two rectangles that intersect and return this overlap as a new rectangle object. This is an exercise that is known as an exercise a lot of students struggle with, so it is a good exercise to create a worked example for.

Once again, a couple examples were manually attempted to begin with. One of these examples is shown in Figure 7.11.

In this example it is clear that the bottom left corner is (2, 1), the width is 3 - 2 = 1 and the height = 2 - 1 = 1. But there exist many different cases of overlap, and the aim is to identify each case so that an algorithm can be constructed that calculates the overlap.

After trying a few different examples, three different categories of overlap have been found based on the amount of corners they overlap.

- Single corner overlap: one of the rectangles overlaps one corner of the other, but no other corners than that.
- Two corner overlap: this means that one edge of a rectangle is completely overlapped by the other rectangle.
- Four corner overlap: one of the rectangles completely overlaps the other.

For the first two cases, there exist 8 variations (one for each side/corner, for both rectangles). For the last case, two variations exist. So a total of 18 conditions need to be implemented. But after

```
def calculateNeighbors(game_state):
    neighbors = []
    for i in range(len(game_state)):
        neighbor_row = []
        for j in range(len(game_state[I])):
            neighbor_row.append(aantalneighborsVoorCel(game_state, i, j))
        neighbors.append(neighbor_row)
    return neighbors


def amountOfNeighboursForCell(game_state, i, j):
    if isValidIndex(game_state, i-1, j-1):
        if game_state[i-1][j-1]:
            nr_neighbors += 1
    if isValidIndex(game_state, i-1, j):
        if game_state[i-1][j]:
            nr_neighbors += 1
    if isValidIndex(game_state, i-1, j+1):
        if game_state[i-1][j+1]:
            nr_neighbors += 1
    if isValidIndex(game_state, i, j-1):
        if game_state[i][j-1]:
            nr_neighbors += 1
    if isValidIndex(game_state, i, j+1):
        if game_state[i][j+1]:
            nr_neighbors += 1
    if isValidIndex(game_state, i+1, j-1):
        if game_state[i+1][j-1]:
            nr_neighbors += 1
    if isValidIndex(game_state, i+1, j):
        if game_state[i+1][j]:
            nr_neighbors += 1
    if isValidIndex(game_state, i+1, j+1):
        if game_state[i+1][j+1]:
            nr_neighbors += 1
    return nr_neighbors


def isValidIndex(game_state, i, j):
    i_is_valid = i >= 0 and i < len(game_state)
    if i_is_valid:
        return False
    else:
        j_is_valid = j >= 0 and j < len(game_state[i])
        return j_is_valid
```

Figure 7.9: Code that calculates the amount of neighbors for a specific cell.

```
def calculateLifeOrDeath(alive, nr_neighbors):
    if alive:
        if nr_neighbors == 2 or nr_neighbors == 3:
            return True
        else:
            return False
    else:
        if nr_neighbors == 3:
            return True
        else:
            return False

def computeNewGameState(game_state, neighbors):
    new_game_state = []
    for i in range(len(neighbors)):
        new_game_state_row = []
        for j in range(len(neighbors[i])):
            nr_neighbors = neighbors[i][j]
            alive = game_state[i][j]
            new_game_state_row.append(calculateLifeOrDeath(alive, nr_neighbors))
            new_game_state.append(new_game_state_row)
    return new_game_state

def nextIteration(game_state):
    neighbors = calculateNeighbors(game_state)
    new_game_state = computeNewGameState(game_state, neighbors)
    return new_game_state
```

Figure 7.10: Code that calculates the next iteration based on the previous iteration.

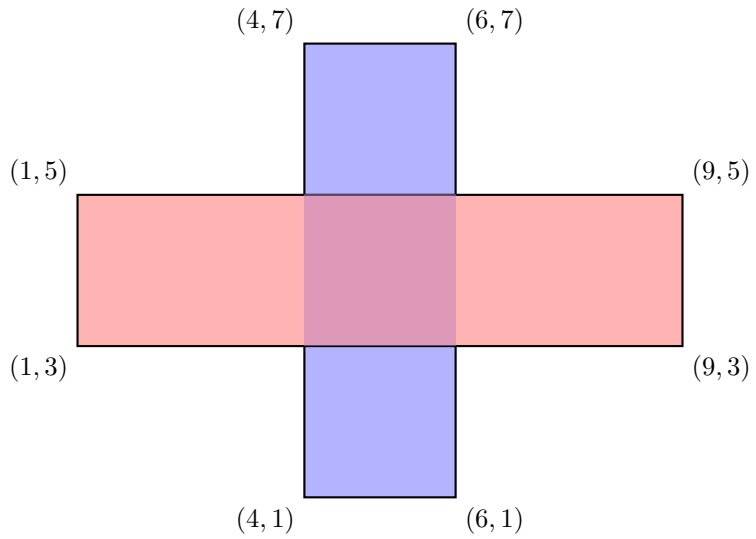

Figure 7.12: An example of an overlap where zero corners of either rectangle are overlapped

```
def computeOverlap(self, rectangle):
    x = max(self.x, rectangle.x)
    y = max(self.y, rectangle.y)
    width = min(self.x + self.width, rectangle.x + rectangle.width) - x
    height = min(self.y + self.height, rectangle.y + rectangle.height) - y
    return Rechthoek(x, y, width, height)
```

Figure 7.13: Code that calculates the next iteration based on the previous iteration.

implementing the first 3 variations, another variation is discovered. The zero corner overlap, where one rectangle covers only part of the side of the other rectangle, on both sides of the rectangle. An example of this is visible in Figure 7.12.

This adds another two variations, giving a total of 20 variations. At this point, it starts to feel like this is not the most efficient approach to this problem. This approach will work, but a large number of if tests is usually a sign that a better approach exists.

Eventually, after going back to the examples and attempting to find patterns for a while, a discovery is made. A much more elegant solution by observing that the overlapped area is always found between the second and third edges (counting the four edges from both rectangles), regardless of orientation or type of overlap. This observation results in the code shown in Figure 7.13.

This code was then tested, upon which no bugs were discovered. The reflection also didn't reveal any potential areas of improvement.

7.5 Roles of variables

The final addition to the course is a document that describes different roles variables can make. These are the same roles that are described in section 4.3.2. The role of a variable is explained, and a section of code is shown where one variable takes the role being described. An example for the role of most-wanted is shown in Figure 7.14.

This is done for all 10 roles described, and serves as a guide for students on the different ways variables can be used. By becoming more familiar with the different ways variables can be used, it should be easier to write code and devise algorithms to be used.

```
ages = [24, 63, 12, 25, 54, 1, 5]
    max_age = ages[0]
    for age in ages:
        if age > max_age:
            max_age = age
    print("The largest age is " + str(max_age))
```

Figure 7.14: An example of code where the variable max_age is a most-wanted holder.

Chapter 8

Conclusions

This thesis consists of two big parts. The first part of this thesis explores the literature surrounding computer science education and gives an overview of the different topics discussed in the literature. The literature surrounding this topic is extremely broad and includes a wide variety of topics, so a selection of the most prominent topics had to be made. This selection was based on papers that provided an overview of the landscape by analyzing many different papers, both through manual review of the papers and through automated methods. The topics discussed are introduced at a high level because explaining every concept in full detail would produce a much longer and more convoluted text, which is undesirable.

Nevertheless, the core of computer science education and the most important topics have been touched upon and introduced in this paper. An overview has been presented of the many different struggles students face when learning programming. Learning to program is a very challenging task, and there are many skills that need to be acquired and many concepts to be learned before someone is able to successfully develop larger projects.

To deal with these issues, assessment methods and teaching strategies have been presented. These methods and exercise types exist as a way to challenge the students and their understanding of computer science concepts, to help them grow and overcome possible misconceptions they may have. Some teaching strategies have been presented as well, like the active learning paradigm. Active learning is particularly suited for programming because programming is an active skill that cannot be passively learned by just learning the syntax and semantics of a programming language.

For the final part of the literature study, the domain of problem solving is investigated, and different problem-solving methods are presented. Problem solving is one of the biggest challenges that beginning students face, and is one of the biggest differences between novices and experts. Experts have years of experience with programming and have solved countless problems. They are familiar with many patterns that can be applied across different problems that share a common solution. Novices, however, lack this knowledge and have to “invent“ new solutions regularly. This process of finding a new solution is very difficult, but there exist methods that give more structure to this process.

The second part of this thesis involves research and a personal contribution to a CS1-level university subject. A survey was presented to students after the subject was finished, and they were asked to self-assess their abilities in the context of the most prominent difficulties students face when learning to program. Their background in mathematics, previous programming experience, and education level were also inquired about. The influence of these background characteristics was then evaluated by analyzing the relation between the different categories for each characteristic and the marks received for the subject.

This analysis revealed that almost certainly no influence exists of the education level on the marks. A slight influence exists from the previous experience with mathematics, but no statistically significant influence was found. The influence of previous programming experience is the largest of the three background factors considered, but is once again not statistically significant. However, the sample size is quite small. It is possible that a larger sample size might very well conclude that a statistically significant influence does exist.

The self-assessment of students was also analyzed in relation to their marks. The correlation between the two was computed, and a significant correlation exists between the self-assessment and the marks for the subject. This correlation was the strongest for the self-assessment related to problem solving. This indicates that being able to successfully perform problem-solving when dealing with new problems gives the biggest increase in odds of passing the course compared to overcoming other types of problems. This also aligns with the literature surrounding computer science education, as it states that problem-solving is one of the toughest yet most important skills to acquire.

Apart from this survey, educational data analysis was performed on the exercises students submitted on the learning platform of Dodona. An approach that was previously described in the literature and successfully applied to other courses was used. Several features were computed for each series and each student, and machine learning was applied to this data. For one of the courses, the passing rate was high, which led to imbalanced data. Several methods were tried to mitigate this issue, and oversampling the data was the approach that gave the best results.

The results in this context were disappointing. Compared to previous research, the balanced accuracy was significantly lower, and the prediction of future years with the data of previous years yields performance that is often no better than a random classifier. This approach has been tested for multiple subjects in different contexts, so one would expect it to also be applicable in this particular context. The biggest difference is the number of students for which data is present, which is several times higher in previous studies. The most important conclusion here is that this approach doesn't produce qualitative results with the amount of data available.

Lastly, a course was evaluated and several things were improved. The course already supplied a variety of exercises, but lacked 2 types of exercises that were found in the literature. Parsons puzzles are good types of exercise at the start because the code only needs to be rearranged so students don't have to write code from scratch, which is a significantly more difficult task. Worked examples were also provided to explicitly show the problem-solving process and the full path taken to arrive at the solution, not just the code that solves the exercise. Additionally, a document was added showing different roles variables can take and examples of a variable being used in such a role.

Bibliography

[May81] Richard E. Mayer. "The Psychology of How Novices Learn Computer Programming". In: *ACM Comput. Surv.* 13.1 (Mar. 1981), pp. 121–141. ISSN: 0360-0300. DOI: 10.1145/356835.356841. URL: https://doi.org/10.1145/356835.356841.

[Sol86] E. Soloway. "Learning to program = learning to construct mechanisms and explanations". In: *Commun. ACM* 29.9 (Sept. 1986), pp. 850–858. ISSN: 0001-0782. DOI: 10.1145/6592.6594. URL: https://doi.org/10.1145/6592.6594.

[Ben98a] Mordechai Ben-Ari. "Constructivism in computer science education". In: *SIGCSE Bull.* 30.1 (Mar. 1998), pp. 257–261. ISSN: 0097-8418. DOI: 10.1145/274790.274308. URL: https://doi.org/10.1145/274790.274308.

[Ben98b] Mordechai Ben-Ari. "Constructivism in computer science education". In: *Proceedings of the Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '98. Atlanta, Georgia, USA: Association for Computing Machinery, 1998, pp. 257–261. ISBN: 0897919947. DOI: 10.1145/273133.274308. URL: https://doi.org/10.1145/273133.274308.

[DTM99] F P Deek, M Turoff, and J A McHugh. "A common model for problem solving and program development". In: *IEEE Trans. Educ.* 42.4 [+CDROM] (1999), pp. 331–336.

[Bru+03] Christine Bruce et al. "Ways of experiencing the act of learning to program: A phenomenographic study of introductory programming students at university". In: (Jan. 2003).

[RRa03] Anthony Robins, Janet Rountree, and Nathan Rountree and. "Learning and Teaching Programming: A Review and Discussion". In: *Computer Science Education* 13.2 (2003), pp. 137–172. DOI: 10.1076/csed.13.2.137.14200. eprint: https://doi.org/10.1076/csed.13.2.137.14200. URL: https://doi.org/10.1076/csed.13.2.137.14200.

[KMA04] Amy J Ko, Brad A Myers, and Htet Htet Aung. "Six learning barriers in end-user programming systems". In: *2004 IEEE Symposium on Visual Languages - Human Centric Computing*. Rome, Italy: IEEE, 2004.

[BR06] Matt Bower and Deborah Richards. "Collaborative learning: Some possibilities and limitations for students and teachers". In: 1 (Jan. 2006).

[BS06] Pauli Byckling and Jorma Sajaniemi. "Roles of variables and programming skills improvement". In: *SIGCSE Bull.* 38.1 (Mar. 2006), pp. 413–417. ISSN: 0097-8418. DOI: 10.1145/1124706.1121470. URL: https://doi.org/10.1145/1124706.1121470.

[Sch08] Carsten Schulte. "Block Model: an educational model of program comprehension as a tool for a scholarly approach to teaching". In: *Proceedings of the Fourth International Workshop on Computing Education Research*. ICER '08. Sydney, Australia: Association for Computing Machinery, 2008, pp. 149–160. ISBN: 9781605582160. DOI: 10.1145/1404520.1404535. URL: https://doi.org/10.1145/1404520.1404535.

[Lee+11] Irene Lee et al. "Computational thinking for youth in practice". In: *ACM Inroads* 2.1 (Feb. 2011), pp. 32–37. ISSN: 2153-2184. DOI: 10.1145/1929887.1929902. URL: https://doi.org/10.1145/1929887.1929902.

[NGK11] Uolevi Nikula, Orlena Gotel, and Jussi Kasurinen. "A Motivation Guided Holistic Rehabilitation of the First Programming Course". In: *ACM Trans. Comput. Educ.* 11.4 (Nov. 2011). DOI: 10.1145/2048931.2048935. URL: https://doi.org/10.1145/2048931.2048935.

[Pie+12] Chris Piech et al. "Modeling how students learn to program". In: *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. SIGCSE '12. Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 153–160. ISBN: 9781450310987. DOI: 10.1145/2157136.2157182. URL: https://doi.org/10.1145/2157136.2157182.

[GM14] Anabela Gomes and Antonio Mendes. "A teacher's view about introductory programming teaching and learning: Difficulties, strategies and motivations". In: *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*. 2014, pp. 1–8. DOI: 10.1109/FIE.2014.7044086.

[KLM15] Theodora Koulouri, Stanislao Lauria, and Robert D. Macredie. "Teaching Introductory Programming: A Quantitative Evaluation of Different Approaches". In: *ACM Trans. Comput. Educ.* 14.4 (Dec. 2015). DOI: 10.1145/2662412. URL: https://doi.org/10.1145/2662412.

[ESH16] Kessia Eugene, Catherine Stringfellow, and Ranette Halverson. "THE USEFULNESS OF RUBRICS IN COMPUTER SCIENCE". In: *INFORMS Journal on Computing* 31 (Apr. 2016), pp. 5–20.

[Lok+16] Dastyni Loksa et al. "Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance". In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. CHI '16. San Jose, California, USA: Association for Computing Machinery, 2016, pp. 1449–1461. ISBN: 9781450333627. DOI: 10.1145/2858036.2858252. URL: https://doi.org/10.1145/2858036.2858252.

[She+16] Swapneel Sheth et al. "A Course on Programming and Problem Solving". In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. SIGCSE '16. Memphis, Tennessee, USA: Association for Computing Machinery, 2016, pp. 323–328. ISBN: 9781450336857. DOI: 10.1145/2839509.2844594. URL: https://doi.org/10.1145/2839509.2844594.

[CGA18] Ricardo Caceffo, Guilherme Gama, and Rodolfo Azevedo. "Exploring Active Learning Approaches to Computer Science Classes". In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. SIGCSE '18. Baltimore, Maryland, USA: Association for Computing Machinery, 2018, pp. 922–927. ISBN: 9781450351034. DOI: 10.1145/3159450.3159585. URL: https://doi.org/10.1145/3159450.3159585.

[QL18] Yizhou Qian and James Lehman. "Students' misconceptions and other difficulties in introductory programming". en. In: *ACM Trans. Comput. Educ.* 18.1 (Mar. 2018), pp. 1–24.

[Wil18] Greg Wilson. *Teaching tech together*. Greg Wilson, July 2018.

[Bar19] Michael Baron. *Probability and statistics for computer scientists*. en. 3rd ed. London, England: CRC Press, June 2019.

[Gre+19] Tyler Greer et al. "On the Effects of Active Learning Environments in Computing Education". In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. SIGCSE '19. Minneapolis, MN, USA: Association for Computing Machinery, 2019, pp. 267–272. ISBN: 9781450358903. DOI: 10.1145/3287324.3287345. URL: https://doi.org/10.1145/3287324.3287345.

[Izu+19] Cruz Izu et al. "Fostering Program Comprehension in Novice Programmers - Learning Activities and Learning Trajectories". In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. ITiCSE-WGR '19. Aberdeen, Scotland Uk: Association for Computing Machinery, 2019, pp. 27–52. ISBN: 9781450375672. DOI: 10.1145/3344429.3372501. URL: https://doi.org/10.1145/3344429.3372501.

[MRF19a] Rodrigo Pessoa Medeiros, Geber Lisboa Ramalho, and Taciana Pontual Falcao. "A systematic literature review on teaching and learning introductory programming in higher education". In: *IEEE Trans. Educ.* 62.2 (May 2019), pp. 77–90.

[MRF19b] Rodrigo Pessoa Medeiros, Geber Lisboa Ramalho, and Taciana Pontual Falcão. "A Systematic Literature Review on Teaching and Learning Introductory Programming in Higher Education". In: *IEEE Transactions on Education* 62.2 (2019), pp. 77–90. DOI: 10.1109/TE.2018.2864133.

[SWK19] Sue Sentance, Jane Waite, and Maria Kallia. "Teaching computer programming with PRIMM: a sociocultural perspective". In: *Computer Science Education* 29.2-3 (2019), pp. 136–176.

[HRL20] Orit Hazzan, Noa Ragonis, and Tami Lapidot. *Guide to teaching computer science*. en. 3rd ed. Cham, Switzerland: Springer Nature, Aug. 2020.

[Pap+20] Zacharoula Papamitsiou et al. "Computing Education Research Landscape through an Analysis of Keywords". In: Aug. 2020. DOI: 10.1145/3372782.3406276.

[BF21] J.H. Berssanette and Antonio Francisco. "Active Learning in the Context of the Teaching/Learning of Computer Programming: A Systematic Review". In: *Journal of Information Technology Education: Research* 20 (Jan. 2021), pp. 201–220. DOI: 10.28945/4767.

[Mai+23] Shahid Maida et al. "Project-based Iterative Teaching Model for Introductory Programming Course". In: *Nile Journal of Communication and Computer Science* 5.1 (2023), pp. 10–41. ISSN: 2805-2366. DOI: 10.21608/njccs.2023.321167. eprint: https://njccs.journals.ekb.eg/article_321167_652485880eebdfd326d599aae2efdb92.pdf. URL: https://njccs.journals.ekb.eg/article_321167.html.

[Sen+23] Sue Sentance et al., eds. *Computer science education*. en. London, England: Bloomsbury Academic, Mar. 2023.

[Agg+24] Ashish Aggarwal et al. "Do Behavioral Factors Influence the Extent to which Students Engage with Formative Practice Opportunities?" In: *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*. SIGCSE 2024. Portland, OR, USA: Association for Computing Machinery, 2024, pp. 18–24. ISBN: 9798400704239. DOI: 10.1145/3626252.3630833. URL: https://doi.org/10.1145/3626252.3630833.

[Bog+24] Christopher Bogart et al. "What Factors Influence Persistence in Project-based Programming Courses at Community Colleges?" In: *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*. SIGCSE 2024. Portland, OR, USA: Association for Computing Machinery, 2024, pp. 116–122. ISBN: 9798400704239. DOI: 10.1145/3626252.3630965. URL: https://doi.org/10.1145/3626252.3630965.

[HTC24] Colton Harper, Keith Tran, and Stephen Cooper. "Conceptual Metaphor Theory in Action: Insights into Student Understanding of Computing Concepts". In: *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*. SIGCSE 2024. Portland, OR, USA: Association for Computing Machinery, 2024, pp. 463–469. ISBN: 9798400704239. DOI: 10.1145/3626252.3630812. URL: https://doi.org/10.1145/3626252.3630812.

[MP24] Syeda Fatema Mazumder and Manuel A. Pérez Quiñones. "The Correctness of the Mental Model of Arrays After Instruction for CS1 Students". In: *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*. SIGCSE 2024. Portland, OR, USA: Association for Computing Machinery, 2024, pp. 806–811. ISBN: 9798400704239. DOI: 10.1145/3626252.3630943. URL: https://doi.org/10.1145/3626252.3630943.

[Van24] Van Petegem, Charlotte. "Dodona : improving programming education through automated assessment, learning analytics, and educational data mining". eng. PhD thesis. Ghent University, 2024, XVI, 172.

[Zhi+24] Denis Zhidkikh et al. "Reproducing Predictive Learning Analytics in CS1: Toward Generalizable and Explainable Models for Enhancing Student Retention". In: *Journal of Learning Analytics* 11.1 (Jan. 2024), pp. 132–150. DOI: 10.18608/

jla.2024.7979. URL: https://learning-analytics.info/index.php/JLA/article/view/7979.

[25] *Software developers, quality assurance analysts, and testers*. Apr. 2025. URL: https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm#tab-6.

[Win25] George Winograd. *College Dropout Rates Statistics 2025 (By Majors)*. May 2025. URL: https://missiongraduatenm.org/college-dropout-statistics/.