ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Development of an Interactive Web-Based Learning Environment for a Secondary School Computer Science Textbook

Master Thesis

Tobias Aeschbacher

September 30, 2021

Advisors: Prof. Dr. J. Hromkovič, Regula Lacher, Dr. Elizabeta Cavar
Department of Computer Science, ETH Zürich

Acknowledgements

Before you lies the thesis “Development of an Interactive Web-Based Learning Environment for a Secondary School Computer Science Textbook”. It was written as a part of the Master’s degree at the Swiss Federal Institute of Technology Zürich. Between April and September 2021, I designed, investigated, and tested a framework that allows solving exercises of the topic “compression”.

The development was challenging and time-consuming but also incredibly exciting. It showed me once again that studying computer science was the right decision. I have many exciting extensions in mind, which I would have loved to implement, but unfortunately, time is limited, and six months is a short time.

I would like to thank my supervisor, Prof. Dr. Juraj Hromkovič, for providing valuable feedback and pointing me in the right direction. I also wish to thank his group, especially Regula Lacher, Prof. Dr. Jens Gallenbacher, Giovanni Serafini, and Jacqueline Staub for carefully testing the application and giving constructive and detailed feedback. Furthermore, I would like to thank Dr. Elizabeta Cavar for giving me an overview of the tools from the Center for Computer Science Education of ETH Zurich (ABZ) and Kevin Tang for patiently explaining the technical details. I also benefited from discussing problems with my friends and family.

I hope you enjoy your reading.

Tobias Aeschbacher, Zürich, September 30, 2021

Abstract

In schools, computer science is becoming increasingly important. Computer science education in secondary schools will be mandatory starting in 2022. It is a new subject for teachers, so the quality of learning materials is crucial to ease the burden on teachers and improve computer science education.

This thesis aims to obtain additional interactive exercises for a computer science textbook's chapter "compression". The exercises' goal is primarily to visualize the mathematics behind the theory so that its understanding is strengthened. An effective graphical representation gives students a clear picture of how the algorithms work and lets them apply through drag & drop. We develop a modern and robust system that can provide individual and tailored feedback to students. In this thesis, we investigate different frameworks, evaluate different architectures, and implement the exercise environment. The core paradigms that are being sought are extensibility and maintainability. Testing the application thoroughly is part of the thesis. A detailed evaluation concludes the thesis.

Contents

Contents **v**

1 Introduction **1**

2 Related Work **3**

3 Requirements **5**
3.1 Functional Requirements 5
3.2 Educational Requirements 6
3.3 Architectural Requirements 6

4 Exercises **9**
4.1 Draw Shannon-Fano Tree 9
4.1.1 Goals 11
4.1.2 Correctness checking 11
4.1.3 Random creation 11
4.2 Draw Max-Balance Tree 12
4.2.1 Goals 12
4.2.2 Correctness checking 12
4.2.3 Random creation 13
4.3 Draw Huffman Tree 13
4.3.1 Goals 15
4.3.2 Solution checking 15
4.3.3 Random creation 16
4.4 Reverse Max-Balance Tree 16
4.4.1 Goals 18
4.4.2 Solution checking 18
4.4.3 Random creation 18
4.5 Reverse Huffman Tree 18
4.5.1 Goals 18

4.5.2 Solution checking . . . 19
4.5.3 Random creation . . . 19
4.6 Binary to Decimal . . . 19
4.6.1 Goals . . . 19
4.6.2 Solution checking . . . 20
4.6.3 Random creation . . . 20
4.7 Decimal to Binary . . . 20
4.7.1 Goals . . . 20
4.7.2 Solution checking . . . 20
4.7.3 Random creation . . . 20
4.8 Shortest Number in Interval . . . 21
4.8.1 Goals . . . 21
4.8.2 Solution checking . . . 21
4.8.3 Random creation . . . 21
4.9 Arithmetic Coding . . . 21
4.9.1 Goals . . . 22
4.9.2 Solution checking . . . 22
4.9.3 Random creation . . . 23

5 Architecture **25**
5.1 Framework Comparison . . . 25
5.1.1 React . . . 25
5.1.2 Vue . . . 26
5.1.3 Angular . . . 26
5.2 General . . . 27
5.3 Chapters . . . 27
5.4 Layout Components . . . 27
5.4.1 Book Overview . . . 27
5.4.2 Chapter Overview . . . 28
5.4.3 Exercise Container . . . 29
5.4.4 Header . . . 30
5.4.5 Page Not Found . . . 30
5.4.6 Side-Nav-Item . . . 31
5.5 Configuration . . . 31
5.5.1 Providing different exercise sets . . . 31
5.5.2 Creating a configuration file . . . 32
5.6 State . . . 37
5.7 Context . . . 40
5.8 Router . . . 40
5.9 Adding more educational content . . . 40
5.9.1 Adding more exercise types . . . 40
5.9.2 Adding more chapters . . . 41

6 Testing **43**

6.1 Coverage 43
6.2 Exercise Components 43
6.3 Solution Checking 44
6.4 Layout Components 45
6.5 Other tests 46
6.5.1 Random Generator 46
6.5.2 Rational Numbers 46
6.5.3 Exercise Configuration 47

7 Evaluation **49**

8 Conclusion **55**
8.1 Conclusion 55
8.2 Future Work 56

A README file of Code Repository **59**

B Short Chapter Documentation **63**

C Configuration File Example **65**

D Exported State Example **71**

E Tests for Huffman Solution Checking **79**

F First User Study Survey **85**

G Second User Study Survey **89**

Bibliography **93**

Chapter 1

Introduction

Digitization has been underway for decades and is advancing into almost all areas of society. The majority comes into contact with IT as users. Due to the rapid change and increasing spread of IT, acquiring at least a basic understanding of the technology is becoming increasingly important. More and more scientific disciplines require fundamental knowledge of computer science. The way of thinking in computer science positively affects school education and should therefore be a mandatory part of high school education and possibly also of elementary school [14].

At the core of information theory is efficient coding. Data compression is an integral part of it, where the goal is to reduce the amount of data to store or transmit. A simple illustration is to transform characters into a short binary representation. Shannon [26] and Fano [11] proposed such codings. As a student of Rober Fano, David Huffman wondered why an algorithm that leads to the optimal coding, assuming that every character gets mapped to a binary encoding, does not exist [27]. Therefore he came up with his idea [15]. This way of thinking is typical in information theory and computer science in general. Apart from being an elementary field of computer science, it is an excellent subject to teach because it is theoretic but has many fundamental and illustrative practical applications.

Understanding theoretical concepts can be challenging without good illustrations. If we learn, for example, that Shannon-Fano coding is not always as good as Huffman coding, then this is a fact that is hard to remember. This knowledge becomes valuable only if the backgrounds are understood. When comparing Shannon-Fano and Huffman, it makes sense to look at a concrete example that does not produce the same trees. Therefore having a set of interactive exercises that work with Shannon-Fano and Huffman trees and provide information about its optimality is outstanding support for understanding these concepts and their threads of thought.

As interactive information processing is an aid for learning theoretical concepts [9], this thesis aims to provide such a framework. In this work, we design a set of interactive exercises and bundle them into one platform. We present eight exercise types with different variations. Some exercises prepare the students for more advanced exercises. The preparation exercises ask for simple tasks, such as converting a decimal number into binary. More advanced exercises require the students to draw trees interactively or find a path to the uncompressed message. We provide a modern platform-independent framework that bundles those different exercises in a central place.

We can split this thesis into three primary goals. The first goal is to develop a platform-independent framework that is easily maintainable and extendable for different topics. In Chapter 2, we analyze and discuss such tools, and in Chapter 3, we specify the requirements. Second, this thesis seeks to identify suitable additional exercises for the chapter *compression* of a secondary school computer science textbook. How this has been achieved is explained in Chapter 4 in detail. The implementation and realization of those exercise types is the third goal. Chapter 5 presents framework-specific choices and gives an overview of the project's structure and maintenance. Detailed information about testing is available in Chapter 6. In chapters 7 and 8, we evaluate and summarize the findings of this work.

Chapter 2

Related Work

In the field of computer programming education, there are many online tools. Even just the Center for Computer Science Education of ETH Zurich (ABZ) provides numerous programming environments, e.g., XLogoOnline [3] and WebTigerJython [2]. Very common, especially for beginners, is so-called block-based programming [30]. Many different block-based programming languages exist, such as Scratch [20] or Tynker [29]. However, block-based programming also advances in more professional environments, i.e., robots programming [31] or mobile device programming [28], also because creating block-based programming languages has become much easier with Blockly [13] or OpenBlocks [24].

The ABZ has several learning environments. A very extensive learning environment with various interactive exercises is the *einfachinformatik* platform[1]. It contains exercises for different levels. There are various topics, like number representation, self correcting codes, minimum spanning trees, maximum flow, and many more. Data compression is also available, but not covered in such a depth as in this work. The platform is similar to this work in the sense, that it is an interactive learning environment for computer science that is not programming. Most of the discovery learning platforms focus on programming. *einfachinformatik* is a tool that provides immediate feedback for the students and supports highly interactive exercise types. The exercise types are configurable by teachers, and also students are able to construct exercises with the playground functionality. Exporting and importing the exercise state, so that solving the exercises can be interrupted without having to restart from scratch, is also possible.

Practice-It [21] is a collection of online exercises from the university of Washington. It contains exercises for the book "Building Java Programs" [22]. Some exercises expect the students to write actual java code, some others ask for theoretical concepts of computer science. The exercise types are all very similar, as they always expect plain text input and do not show pic-

tures or graphs. The tool is also used in different courses of the university of Washington and therefore also supports hand-in and teacher administration functionality. Most of the exercise types simply ask for knowledge of the book. However, there are some exercise types, that require the students to apply the knowledge and construct the solution with it.

Brilliant [8] is an interactive website that teaches theoretical computer science topics. The website contains more than just programming exercises. Exercise types include decision making, Eulerian paths, and related topics. This tool differs from our thesis in terms of its concept. The website contains the theory and is not based on a book.

Chapter 3

Requirements

This chapter describes the different requirements of the application. The requirements are split into different categories. First, we define the basic functional requirements, then the essential educational requirements, and finish with the needed architectural requirements.

3.1 Functional Requirements

Some functional requirements are fundamental for the application. They must be implemented carefully and of high quality. The following requirements describe the basic functionalities that characterize the application.

- The environment must display different exercises.
- The exercises can be chosen and solved in any order.
- It must be possible to switch between the exercises without losing the current progress.
- It must be visible whether the current exercise state is correct or not.
- If the exercise state is incorrect, a helpful notification should assist the user in finding the mistake.
- Teachers must be able to create different exercises for different students.
- The exercise state should be saveable or exportable so that it is possible to submit the solution.
- Exercise configurations must be sharable.
- Motivated students must be able to let the application set them more exercises to solve.

3.2 Educational Requirements

The educational requirements are essential for a good learning experience.

- The application should be self-explanatory so that the focus lies on the exercise content. How to navigate through the exercises or the steps necessary to solve the exercises should not be the main focus.
- The exercises should support individual learning. Teachers are not expected to guide students one by one throughout solving exercises in this framework. Students must be able to solve those exercises individually and at their speed.
- The exercise collection is expected to be an extension. The application should be an addition to a textbook and not explain the theory with its background. The purpose is to provide exercises that allow students to apply and train their knowledge. The application must do all the math in the background, enabling the students to look at the material from a high-level perspective.
- The exercise design should be motivating and should prevent subsequent errors. Students should be informed when they are on the wrong path.
- The exercise collection should be adaptable by the teacher, so that exercise difficulty meets different student's needs.
- Helpful feedback should be provided to lighten the teacher's load. Significantly, the application needs to inform the students whether their solution is correct or not. Correctness of this feedback is of elementary importance.

3.3 Architectural Requirements

The architecture requirements should make sure that the application can be deployed and used at schools. Besides being didactically meaningful, there are practical needs that make the usage of the application by the teacher more attractive. Schools have financial, technical, and administrative limitations that lead to the following requirements.

- *Platform independence:* It is crucial for schools that the application does not support just a single architecture. Otherwise, substantial financial efforts would be needed to use it.
- *Easy installation:* The technical efforts to make the application running should be as low as possible. Such an addition to a textbook must be straightforward to deploy and must not cost too much time by teachers or IT administrators.

- *Easy maintainability:* It is not just educational platforms that suffer from the ephemeral nature of technology. After a short period, such tools can get outdated. On the one hand, this has implications for the graphical user interface (GUI); on the other hand this often entails security issues. Therefore, an architecture that minimizes the effort of keeping the application up-to-date in terms of appearance, functionality, and security standards, is wanted.
- *Future-oriented technology:* Another aspect of keeping the application up to date is choosing a technology whose support is guaranteed in the long term.
- *Low resource utilization:* Schools may not always provide the latest hardware. Depending on the school, they may not even provide any hardware. Therefore it is vital to make the application support older hardware and not requiring too many resources.

Chapter 4

Exercises

In this chapter, we focus on the different exercise types. We explain what the exercise types expect, why they are helpful, and provide background information. We explain how the student's solution is checked for correctness and how the random creation works for each exercise type. More information about the creation of exercise instances with a configuration file can be found in Section 5.5.

4.1 Draw Shannon-Fano Tree

This exercise type shows a table with the letter frequencies and a tree-drawing frame. Students are expected to draw the Shannon-Fano tree for the given letter frequency distribution. The tree-drawing frame allows the student to build the tree visually with drag&drop. The letters that need to be distributed are already available and simply need to be dragged to the correct location. This setup prevents wrong letters from being inserted or some of the letters getting lost or forgotten.

The initial table is shown in Figure 4.4a, and the initial tree-drawing frame is shown in Figure 4.1. After performing some steps of the exercise, it might look like in Figure 4.2. Figure 4.3 shows the tree after finishing the exercise.

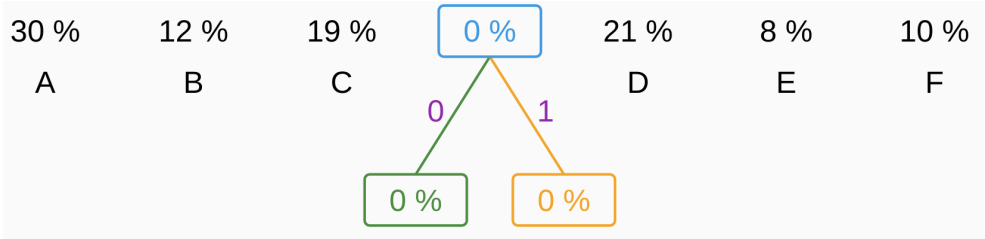


Figure 4.1: The Shannon-Fano exercise in its initial state.

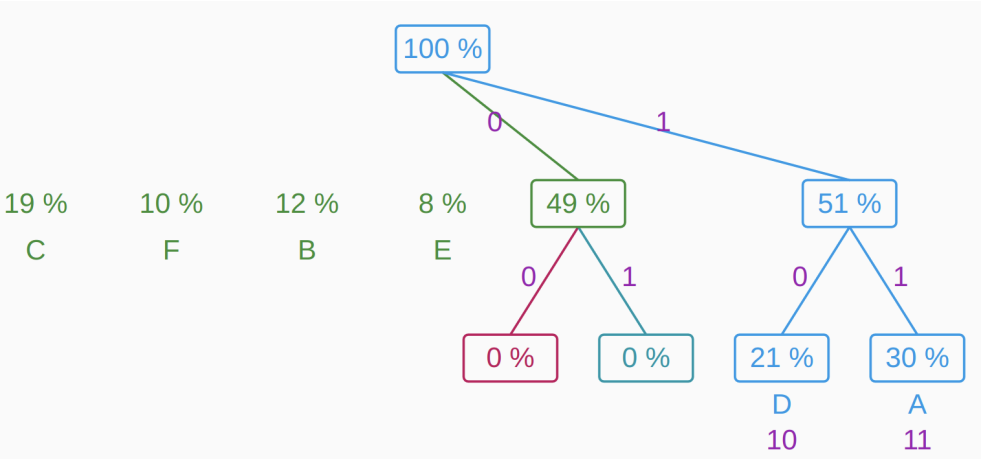


Figure 4.2: The Shannon-Fano exercise where A and D are in its final positions and B, C, E, and F have not been split yet.

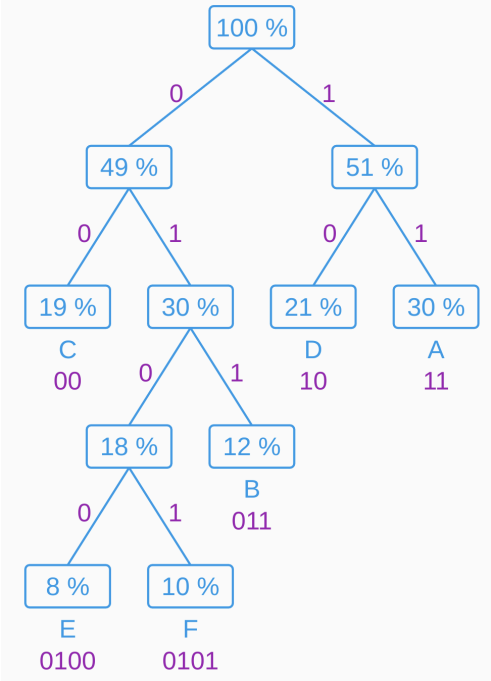


Figure 4.3: The Shannon-Fano exercise where every letter is in its final position.

Buchstabe	Häufigkeit	Codierung
A	30%	
B	12%	
C	19%	
D	21%	
E	8%	
F	10%	

(a) Initial table

Buchstabe	Häufigkeit	Codierung
A	30%	11
B	12%	
C	19%	
D	21%	10
E	8%	
F	10%	

(b) Table in progress

Buchstabe	Häufigkeit	Codierung
A	30%	11
B	12%	011
C	19%	00
D	21%	10
E	8%	0100
F	10%	0101
		2.48

(c) Final table

Figure 4.4: Different states of the shown table for the Max-Balance exercise.

Shannon-Fano coding refers to two different techniques to construct a prefix code from Claude E. Shannon [26] and Robert M. Fano [11]. *Fano's method* is the approach used in the computer science textbook that this work is based on.

4.1.1 Goals

The main focus of the exercise is to understand the concept of Shannon-Fano trees. The understanding of how prefix-free encodings are created is also strengthened along the way and shown visually. The table[1] also shows how many bits per letter are needed for the particular tree for interested students. This allows the results to compare to Max-Balance or Huffman trees.

4.1.2 Correctness checking

Solution checking follows the Shannon-Fano algorithm as described by Fano [11]. The numbers are first sorted, then the indices are computed where the split can happen. There are two split positions if they both lead to an equally good sum on both sides. Then it is checked that the student's solution corresponds to one of the possible two correct solutions.

4.1.3 Random creation

For each exercise, there is an implementation to create a random instance of it. The only thing that differs between different instances of drawing

[1] Shown in Figure 4.4c

Shannon-Fano tree exercises is the letter frequency distribution. The number of letters is reduced to a maximum of 7 to make the tree fit well on computer and tablet screens. Less than three letters are educationally worthless, so we take three as the minimum and determine the number of letters n randomly, such that $2 < n < 8$. To create a random letter frequency distribution of n letters that sums up to 100%, we generate the first $n-1$ letters l_i for $i = 0, 1, ..., n-2$ normally distributed in $[1, r_i - (n-i) + 1]$ where $r_i = 100 - \sum_{k=0}^{i-1} l_k$ is the still available percentage after having chosen the first i frequencies. l_{n-1} is then set to the remaining frequency r_{n-1}. The advantage of using Gauss distribution for choosing the letter frequencies is that it is less likely to get one or more letters with a probability of 1%.

4.2 Draw Max-Balance Tree

The exercise setup is the same as for the previous exercise. The only difference is that the criterion is different for splitting the letters. In this exercise, the letters must be split with the best possible balancing. Therefore, the difference of the left and the right sides' sums must be as close to zero as possible. We call a tree with this splitting criterion *Max-Balance tree* from now on. They do not necessarily lead to a better compression than Shannon-Fano.

The GUI is the same as for the previous exercise. Figure 4.5 shows the tree drawing frame, where the user has completed the first step (A and E on the left side with 49%, and the rest on the right side with 51%), but more actions need to be done on the right side. However, the two decisions that need to be made are trivial as only two letters are left per decision. Figure 4.6 illustrates the completed tree. The corresponding tables are shown in Figures 4.7b and 4.7c, whereas Figure 4.7a shows the initial table.

4.2.1 Goals

The goal of the exercise is to think about what different splitting criteria mean for the encoding. With this method, the students know another approach to construct a prefix-free encoding, and they have something to compare Shannon-Fano to.

4.2.2 Correctness checking

Checking whether the task has been solved correctly is NP-hard. It is equivalent to the subset sum problem as an optimization problem. This problem can be solved in pseudo-polynomial time with dynamic programming as the sums are small (up to 100). However, this does not work for floating point numbers. Therefore, we use the brute force algorithm to check whether the solution is correct or not. We calculate every possible subset sum, take the

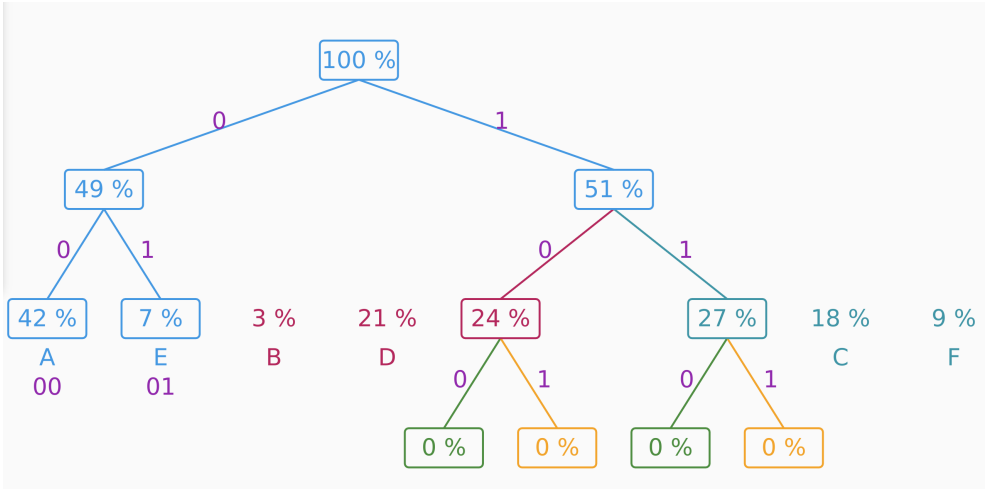


Figure 4.5: The Max-Balance exercise where A and E are in its final positions and B, D, C, and F have been split correctly.

best one[2], and check whether the student's solution is equally good. As computing every subset sum for n values costs exponential time, this is only acceptable for $n \approx 25$. For this exercise type, it is enough. However, we use split&list to improve the runtime further. Therefore, we divide the values into two equally sized subsets S_0 and S_1 and compute all the subsets sums for both sets. After sorting the subsets of S_1, we can simply find for each subset sum of S_0 the best other subset sum of S_1 with binary search. Therefore, the overall asymptotic complexity is $O(n \cdot 2^{n/2})$, which is acceptable for the expected input sizes of this exercise as it allows around $n \approx 50$ letters to be corrected in a feasible time.

4.2.3 Random creation

The same random creation procedure is used for this exercise as in the previous exercise. Details can be found in Section 4.1.3.

4.3 Draw Huffman Tree

This exercise is very similar to the top-down drawing exercises. The letter frequency table is the same, and the tree frame is similar. The tree frame allows to build the tree bottom-up with drag&drop mouse or touch gestures, as shown in Figure 4.8. In contrast to the previous exercise, students are expected to draw a Huffman tree [15].

[2]The best subset sum according to Max-Balance is the one that is closest to $0.5 \cdot \Sigma_{i=0}^{n-1} v_i$ for the n values $v_0, \ldots, v_{n-1}$.

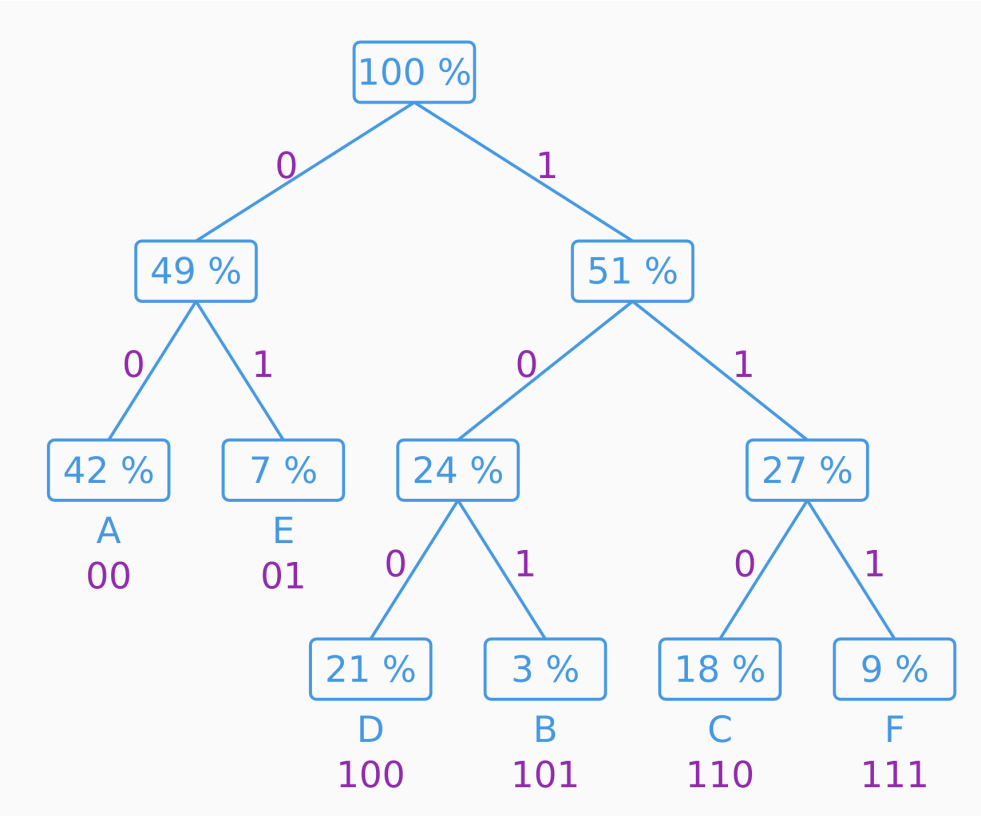


Figure 4.6: The Max-Balance exercise where every letter is in its final position.

Letter	Frequency	Encoding
A	42%	
B	3%	
C	18%	
D	21%	
E	7%	
F	9%	

(a) Initial table

Letter	Frequency	Encoding
A	42%	00
B	3%	
C	18%	
D	21%	
E	7%	01
F	9%	

(b) Table in progress

Letter	Frequency	Encoding
A	42%	00
B	3%	101
C	18%	110
D	21%	100
E	7%	01
F	9%	111
		2.51

(c) Final table

Figure 4.7: Different states of the shown table for the Max-Balance exercise.

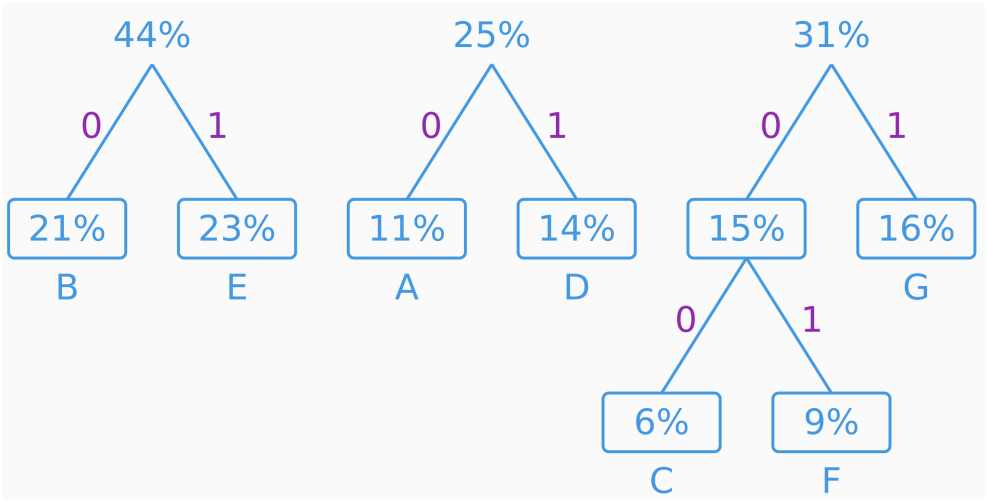


Figure 4.8: The Huffman exercise where three sub trees are left to be connected.

Two different modes can be configured in the framework. One option is to prevent wrong moves, which means that the exercise framework directly informs the student that an action is not possible if it is not the correct action. This option prevents errors directly when they are made so that frustration in the learning experience can be reduced as much as possible. A disadvantage of this approach is that the problem can be solved with simple trial and error. However, as this platform should provide an addition to the textbook, we chose this approach as default. The other option is allowing those moves and informing the student after the whole tree has been built, whether it is correct or not. This approach requires more concentration to prevent subsequent errors.

4.3.1 Goals

The goal of this exercise is to understand how Huffman trees are built. Like the previous exercise, the assignment of prefix-free encodings is done by the framework and displayed on the fly. The average encoding length is calculated to enable students to also think about the optimality of the coding.

4.3.2 Solution checking

We first perform some sanity checks to ensure the tree has a valid format to check the solution. These checks are unnecessary in this specific case, as the GUI guarantees a correct tree format but are implemented to make the code usable in the general case. If the tree is valid, it is traversed and analyzed

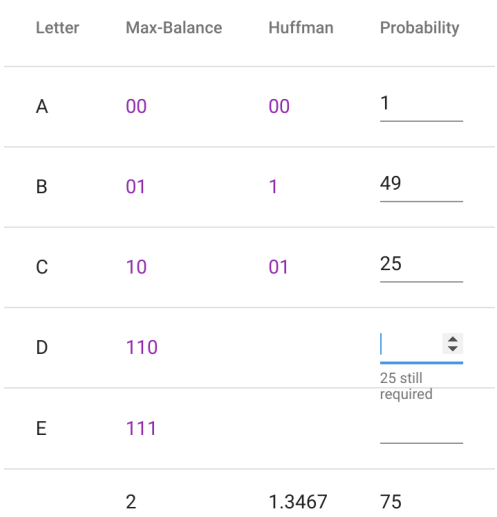


Figure 4.9: The Reverse Max-Balance exercise where letters A, B, and C already are assigned to frequencies.

from the bottom up to check that each decision[3] is correct. The asymptotic complexity for determining whether it is correct is in $O(n^2)$ because for each of the $\frac{n}{2} - 1$ inner nodes we need to find the two smallest nodes, which can be done in $O(n)$.

4.3.3 Random creation

The same random creation procedure is used for this exercise as in the Shannon-Fano exercise. Details can be found in Section 4.1.3.

4.4 Reverse Max-Balance Tree

In this exercise, we expect students to think of the reverse process of Max-Balance trees. The Max-Balance tree is shown in a tree frame, as illustrated in Figure 4.10, and the letter frequencies are sought, such that the shown tree is the correct Max-Balance tree. The table, depicted in Figure 4.9, is editable, and the letter frequencies must be entered there. An additional constraint is to choose the letter frequencies in a way that the resulting Huffman tree is a better encoding than the shown Max-Balance tree. The Huffman tree for the currently entered letter frequencies is also shown to assist the students in finding such a letter frequency distribution. The table also shows the average encoding length per letter, which can be used to see which tree

[3]We use the term *decision* for choosing the two smallest subtrees to connect. Therefore every inner node in the Huffman tree is a *decision*.

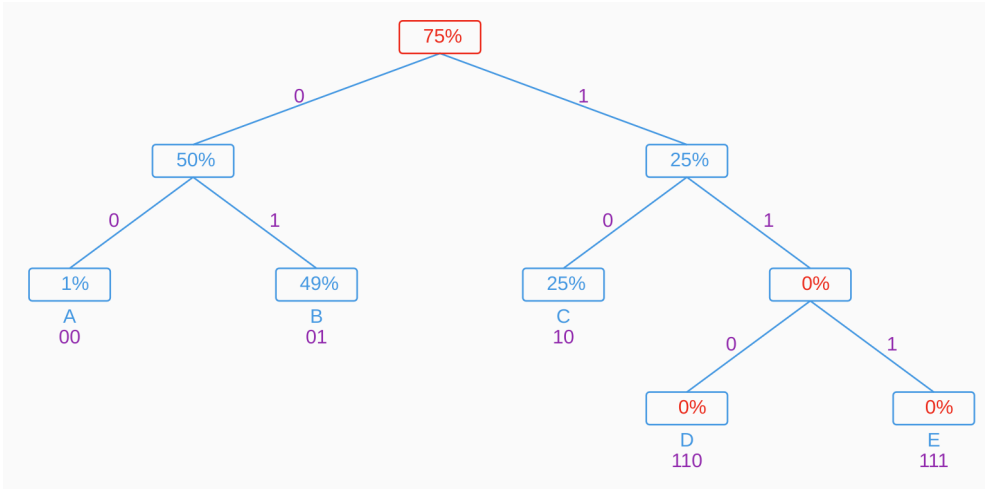


Figure 4.10: The Max-Balance tree that must result of the entered letter frequencies.

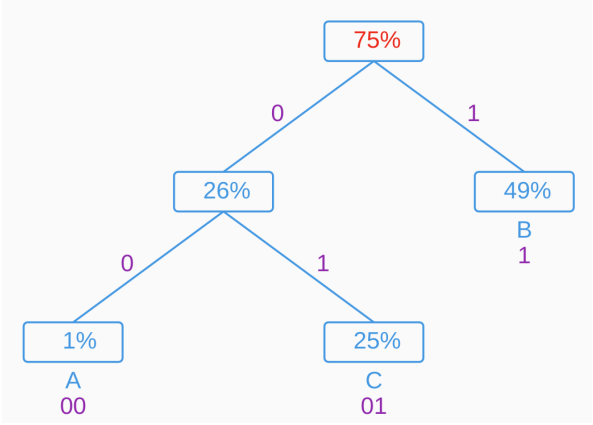


Figure 4.11: The Huffman tree that results of the currently typed letter frequencies.

is a better encoding. Figure 4.11 shows the Huffman tree for the entered frequencies of Figure 4.9. Students can always get feedback for their progress by clicking the feedback button. The possible feedbacks are notes that the letter frequencies do not lead to the given Max-Balance tree, that the Max-Balance tree is equally good as the Huffman tree or that there are invalid letter frequencies, e.g., the sum is not 100. Additionally, the application colors the numbers in red, if they are negative or zero, or if the root is not 100. These indicators give an immediate hint where an improvement is necessary. The border of a node in the Max-Balance tree frame is colored red to indicate that its children are not optimally divided.

4.4.1 Goals

The goal of this exercise is first to think about how Max-Balance trees are constructed and how they compare to Huffman trees. Also noteworthy is that students learn to think about comparing the quality of an encoding used for compression. This capability is strengthened by showing the average encoding lengths per letter. With this exercise type, students can prove that they understand the concepts of Max-Balance and Huffman trees and what characterizes good encodings.

4.4.2 Solution checking

To check whether the given letter frequency distribution is correct, we check that the given Max-Balance tree corresponds to this distribution. This check is done equivalently for the Max-Balance exercise described in Section 4.2.2. If this check is successful, we verify that the resulting Huffman tree is indeed also the better encoding than the given Max-Balance tree.

4.4.3 Random creation

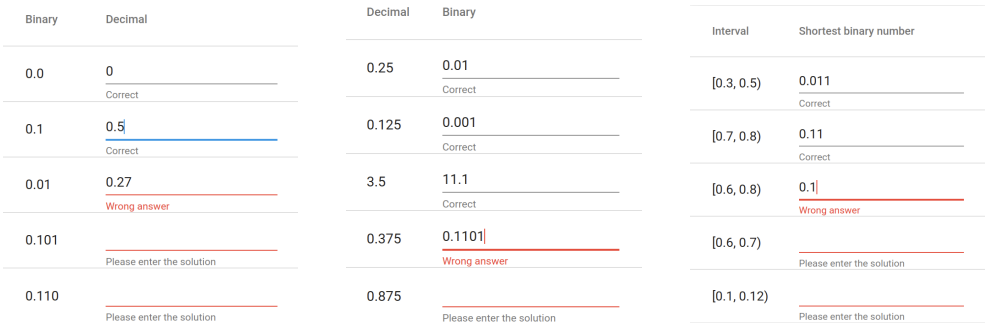
The random creation of this exercise is done by randomly creating a binary tree. The tree needs to have at least four leaves so that a better Huffman tree is even possible. Therefore the first two levels of the tree are set initially. For each node, two children are added with a probability of 0.5. At each lower level, the probability that another two children are added is halved.

4.5 Reverse Huffman Tree

This exercise is conceptually the same as the Reverse Max-Balance tree. The difference is that the given tree must be a Huffman tree, based on the entered letter frequency distribution. There are two different variants of the exercise. The first one is without any additional constraints. For the second variant, there is an additional constraint, where the letter frequency distribution must be chosen so that the resulting (given) Huffman tree is perfect among all compression schemes, e.g., arithmetic coding. Huffman trees are unbeatable for compression schemes where every letter is assigned an encoding, but better compression schemes may generally exist depending on the frequencies. So, we look for a letter frequency distribution in this variant, where it is *impossible* to create a better compression scheme.

4.5.1 Goals

The main focus of this exercise is again understanding Huffman trees. The ability to apply the reverse process of creating Huffman trees enforces the understanding of the concept. The second variant is an advanced exercise,



(a) Binary to Decimal **(b)** Decimal to Binary **(c)** Shortest number

Figure 4.12: The three exercise types that consist of an editable table.

where a good understanding of compression schemes is required in general. Many information-theoretic concepts can be elaborated on, e.g., Shannon entropy [26]. These concepts do not need to be understood in detail, as the textbook covers sufficient information.

4.5.2 Solution checking

The version without any additional constraints is checked analogously to the Huffman drawing exercise, as explained in more detail in Section 4.3.2. The variant with the optimal tree needs an additional check to ensure every letter l_i has the probability of 2^{-e_i}, where e_i is the length of the encoding of letter l_i.

4.5.3 Random creation

The random creation of this exercise type is the same as in Section 4.4.3 of the Reverse Max-Balance tree.

4.6 Binary to Decimal

In the binary to decimal exercise, binary rational numbers are shown, and the students need to write them in decimal format, as shown in Figure 4.12a.

4.6.1 Goals

The main goal is to familiarize the students with the binary notation of numbers. It is a preparation for the arithmetic coding exercise, where this knowledge is applied. However, the arithmetic coding exercise hides the math, and therefore this conversion exercise is not necessary to solve the arithmetic coding exercise.

4.6.2 Solution checking

Checking solutions is trivial here, as the entered number is converted from string to a typescript number, a floating-point number defined by IEEE-754. Due to this exercise's relatively short binary numbers, the conversion happens without rounding, and the numbers can be compared precisely. The reference value is computed by simply iterating over the binary string.

4.6.3 Random creation

When creating a random instance of this exercise type, a binary number in $[0, 1)$ is generated with a maximum of four digits. Zeros and ones are created with equal probability. The exercise instance is complete as soon as ten unique binary numbers have been generated.

4.7 Decimal to Binary

The second exercise that consists of a table to convert numbers is the inverse of the first one. Students are expected to find the binary string that matches the decimal number, as shown in Figure 4.12b. As usual for this kind of exercise, the table gives instant feedback for each entry.

4.7.1 Goals

With this exercise, the ability to understand how binary numbers can be constructed is encouraged. This knowledge is beneficial for seeing how arithmetic coding can be used and enables one to see arithmetic coding in a broader context.

4.7.2 Solution checking

The solution checking is, similar to the previous exercise, trivial. The entered binary string is traversed bit by bit, and the decimal number as IEEE-754 floating-point number can be constructed without rounding.

4.7.3 Random creation

The random creation of this exercise is done by randomly creating a binary number and converting it to a decimal number. The difference from the previous exercise is that up to six digits are generated. The probability that a one is being generated decreases for the least significant bits. That is a countermeasure to ensure that a table contains a good mix of easy and more challenging numbers.

4.8 Shortest Number in Interval

The third exercise type consisting of a single table is where the students must find the shortest binary number that fits into a given interval. Figure 4.12c shows this exercise.

4.8.1 Goals

This exercise is the core of arithmetic coding, which leads to good compression. By extracting this exercise from the arithmetic coding exercise, students will be familiar with how the shortest number can be found. As this exercise may sound quite complex and mathematical at first sight, it will get more straightforward than expected when trying to solve it.

4.8.2 Solution checking

The framework uses Algorithm 1 to check whether a number is the shortest possible number in the interval. This approach might also be the easiest for the students, when solving the exercise.

Algorithm 1 Find the shortest number inside an interval $[a, b)$

Require: $0 \leq a < b < 1$
  $x \leftarrow 0$
  $i \leftarrow 1$
  while $x < a$ **do**
    if $x + 2^{-i} < b$ **then**
      $x \leftarrow x + 2^{-i}$
    end if
    $i \leftarrow i + 1$
  end while
  return x

4.8.3 Random creation

The random creation process generates ten intervals where every interval has a different solution. The intervals bounds are chosen such that $b - a \geq \frac{1}{50}$. Therefore the solution is limited to 6 bits.

4.9 Arithmetic Coding

In this exercise, we display the encoding of a message as the number defining the interval and the message length. An interactive bar is shown, which is used to find the original message. The bar is labeled with the interval

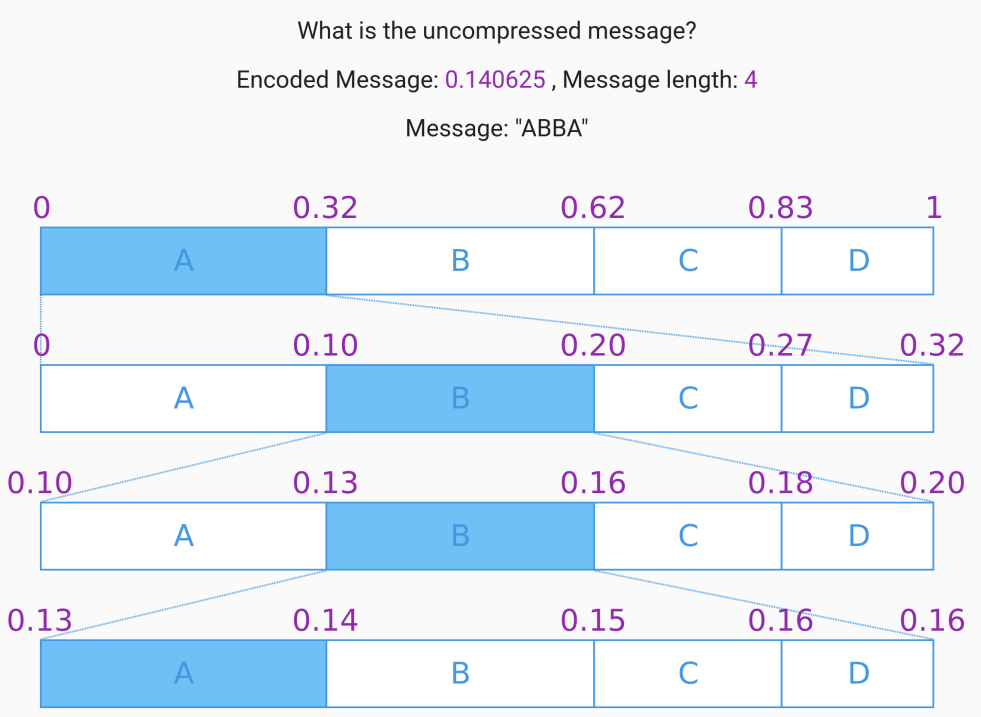


Figure 4.13: The Shannon-Fano exercise where A and E are in its final positions and B, D, C, and F have been split correctly.

bounds, and with clicking intervals, they get extended and show the subsequent intervals. Figure 4.13 shows the exercises where the user has already selected the correct message.

4.9.1 Goals

Students should not be annoyed with the math behind the arithmetic coding, which can get cumbersome quickly. Therefore the entire math is done by the framework, and the students only need to understand the concept of arithmetic coding to select the correct intervals.

4.9.2 Solution checking

Checking the solution is trivial for this exercise, as only message length and the selected interval must be compared. The implementation, however, needs arbitrary precision integers to support the whole set of rational numbers. To support rational numbers with all the basic operations, we wrote our own rational number implementation. With that, arbitrarily large messages could be compressed in theory. We need to limit the message length for convenience and visual representation, as the interval bound representations get too long with the increasingly necessary precision.

4.9.3 Random creation

The random creation algorithm generates an exercise with 2, 3, or 4 letters. The probability of a letter is at least 20% because smaller intervals cannot be neatly arranged in the GUI. The message length is then randomly derived as a number between 2 and 4, and the target interval is described by a random number r in $[0, 100)$, where $\frac{r}{100}$ defines the interval. Note that this interval description is not the shortest possible binary number as used in arithmetic coding. The number will be converted to the shortest binary representation when the framework loads the exercise.

Chapter 5

Architecture

In this chapter, we will explain how the project is structured and which directories contain the relevant parts. The project's directory structure is illustrated in Figure 5.1. First, we evaluate different frameworks in Section 5.1. Sections 5.2 to 5.8 explain the main concepts of the application, while Section 5.9 describes how to extend it. In general, this thesis' directory structure follows the *einfachinformatik* platform that Sonja Blum initiated [6].

5.1 Framework Comparison

It is essential to use a framework that can be used and kept up-to-date for a long time. Therefore, we require a framework that is both modern and well documented and is still in active development. It must be robust, easy to use, and provide support for different platforms. Our framework needs to withstand major releases of operating systems since technology changes rapidly. Web-based solutions are best suited to this purpose. Native applications often have advantages in terms of performance and usability, but they must be completely rewritten for every platform. With a web-based technology, only one code repository needs to be maintained, while native solutions require active maintenance of around four to six code bases.

5.1.1 React

React [17] is an open source front-end JavaScript library developed by Facebook. It was initiated in 2013 and has been in active development ever since. React is a component based library that also features a virtual DOM, and supports JSX [16]. The disadvantage of react is that it is written in JavaScript and uses JavaScript. To achieve high code quality a separate TypeScript compiler is needed.

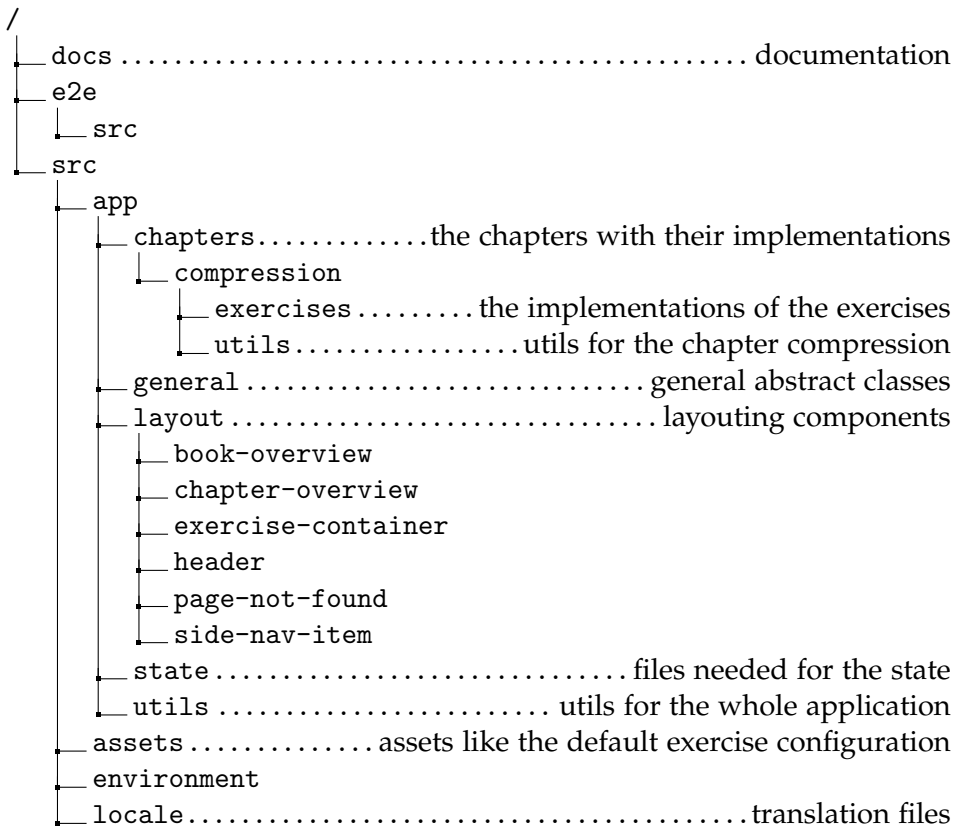


Figure 5.1: The directory structure of the application.

5.1.2 Vue

Vue [32] is an open source front-end JavaScript framework for single-page applications. Evan You has been developing and maintaining it since 2014. It features components and templates and has a good support for various transition effects. Its main disadvantage is that it only supports single-page applications. As a result, sharable URLs are no longer possible. For example, it is not possible to create an exercise and then share its URL with others. That is a limitation that does not meet this application's requirements.

5.1.3 Angular

Angular [12] is an open source web application framework developed by Google. It is based on TypeScript and features components, templates, and routing. It was first released in 2010 under the name AngularJS. Since 2016 it is actively maintained under the name Angular. Angular is the framework of choice for this application because it meets all the requirements. An advantage of using Angular is that it is a modern framework with long-term

support. It is also advantageous that the ABZ already maintains other Angular projects, i.e., einfachinformatik [1], enabling synergies in maintaining and further developing it.

5.2 General

The general directory contains two essential files. First, there is the exercise feedback class, which is shown in Listing 1. It just provides a method that returns a boolean value, saying whether an exercise has been solved correctly or not. There are two direct sub classes. On the one hand, there is a class that represents a correctly solved exercise, and on the other hand, there is a class for wrong solutions. The wrong solution class `ExerciseFeedbackWrong` must be overwritten and provide a method that returns the error message. Depending on the exercise, this is the message that is shown to the user.

The second file contains the abstract general exercise component. Every exercise type has one main component responsible for state management, solution checking, and showing feedback. Exercise components may, of course, have sub-components. The abstract exercise class, shown in Listing 2, guarantees its sub-classes implement the required functionality by defining abstract methods. It also contains the functionality to display messages to the users, handle *undo* and *redo* with the shortcuts `ctrl + z` and `ctrl + y`, and manage state changes. Section 5.6 discusses states in more detail.

5.3 Chapters

Each chapter has its own folder in the chapters directory. As described in Section 5.2, there is one main exercise component for each exercise type. Those components are located in the 'exercise' directory. A chapter folder's directory structure is flexible and can be tailored to a particular chapter.

5.4 Layout Components

The layout directory contains layout components, such as the navigation bar and the sidebar. In the following sections we will present and explain those in more detail.

5.4.1 Book Overview

The book overview component displays the available chapters of the book as a grid. The chapters can be selected by clicking. This component is shown as the root component, i.e., `www.example.com/` would route to this component.

Listing 1 The exercise feedback classes.

```
/**
 * This defines the minimum functionality of a feedback object.
 */
export abstract class ExerciseFeedback {
    abstract isCorrect(): boolean;
}

/**
 * This class represents a correctly solved exercise.
 */
export class ExerciseFeedbackCorrect extends ExerciseFeedback {
    isCorrect(): boolean {
        return true;
    }
}

/**
 * This class represents a wrong exercise. Subclasses must provide at
 * least an error message as feedback.
 */
export abstract class ExerciseFeedbackWrong extends ExerciseFeedback {
    abstract getFeedbackAsString(): string;

    isCorrect(): boolean {
        return false;
    }
}
```

It is also the component that parses the configuration encoded in the URL (if available). More information about sharing configurations is covered in Section 5.5.

5.4.2 Chapter Overview

The chapter overview component looks very similar to the book overview. The difference is that it shows a grid with the exercises from the selected chapter. There are two ways to open an exercise. This component takes an URL parameter `chapter`, so an URL pointing to this component could look like `www.example.com/chapter/2`, which would display the chapter with the id 2. This component loads the exercises of a chapter and redirects to the exercise that was clicked.

Listing 2 An excerpt of the general exercise component.

```
/**
 * The general exercise component that can be extended by components
 * that implement different exercise types. This class handles
 * chapter/exercise IDs and also providing functionality for setting
 * the state and checking whether the solution of an exercise is
 * correct.
 */
@Directive()
export abstract class GeneralExerciseComponent
       <T extends ExerciseConfig<S>, S extends ExerciseState> {

    /**
     * This method gets called when the state changes. It should
     * return ExerciseFeedbackCorrect if the exercise has been solved
     * correctly (based on the given state). Otherwise a specific
     * subclass of ExerciseFeedbackWrong should be returned.
     */
    abstract checkSolution(state: S): ExerciseFeedback;

    /**
     * This method gets called, when another state should be displayed
     * because an undo/redo has been triggered.
     * @param state the state that should be shown
     */
    protected abstract onUndoRedo(state: S): void;
}
```

5.4.3 Exercise Container

The exercise container component displays an exercise. One can specify the ids of the chapter and exercise in the URL, i.e.,

`www.example.com/chapter/0/exercise/2`,

which displays chapter 0 and exercise 2. The component checks that the passed arguments are valid, looks up the exercise type, and passes the exercise configuration to the corresponding exercise component. Furthermore, it allows to specify an exercise type and a seed argument, for instance:

`www.example.com/chapter/0/type/draw-huffman/seed/276427631`

In this example, a Huffman drawing exercise would be generated with 276427631 as seed. The exercise component is then added to the user's view and becomes visible.

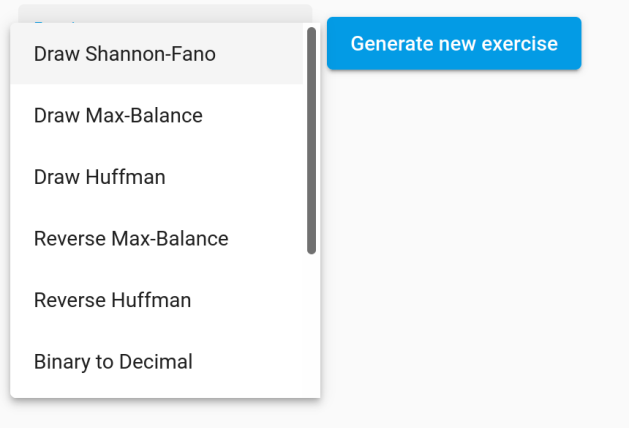


Figure 5.2: The list of exercise types to create a random instance of.

The configuration object, described in Section 5.5, provides specific information about the exercise type, such as the random creation. The only exercise type specific information in the exercise container component is a mapping of exercise types to their exercise component so that the correct component can be loaded. In the exercise container component, exercise types are mapped to their exercise components so that the correct component can be loaded. This mapping is shown later in Listing 16.

This component also handles the random exercise selection. The random section can be accessed in the navigation sidebar. The users can select an exercise type of a list as shown in Figure 5.2.

5.4.4 Header

The header component displays the header bar. The functionalities provided by this bar include displaying feedback for exercises, importing or exporting states and configurations, or resetting the current exercise progress. This component manages the import and export, and displays the feedback using the context service (see Section 5.7).

5.4.5 Page Not Found

The page not found component is the most straightforward component as it only shows a 404 page not found error. The angular router redirects to this component if the loaded route is not valid. More information about the router can be found in Section 5.8.

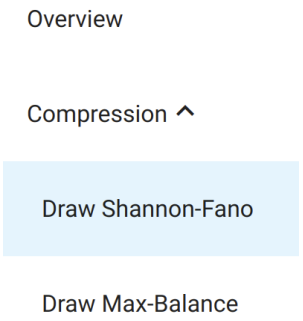


Figure 5.3: Some items of the side navigation bar.

5.4.6 Side-Nav-Item

The side-nav-item component represents one item of the side navigation bar. A selection of those items is shown in Figure 5.3. The component subscribes for URL changes and, therefore, can update its look[1].

5.5 Configuration

We call a set of specific exercises a configuration. The application can read a JSON [10] formatted configuration file. Such a file is included in the project. For every project deployment, a different default configuration `book_config.json` can be located in the `assets` directory.

The configuration service is the provider of the configuration. It gets injected into those components that need to access the configuration. If a new configuration is loaded, it is passed to the configuration service so that all the components can access the latest version of it. After the JSON formatted object is loaded in the system, it is checked for validity and converted to a TypeScript object. As a result, the application will benefit from a typed configuration.

5.5.1 Providing different exercise sets

Besides changing the default configuration, it is possible to generate an URL containing a different configuration. For this purpose, a configuration file needs to be imported into the application. The import functionality is located in the top right corner in the overflow menu. After importing the file,

[1]The currently shown exercise is highlighted in blue. In Figure 5.3, the Shannon-Fano exercise is shown and therefore highlighted.

an URL is opened with the following structure:

`www.example.com/c/<config>/h/<checksum>`

Instead of `<config>`, a long configuration string is inserted, and `<checksum>` is a number used to check the validity of the configuration string. This URL will trigger the book overview component to parse the given configuration and check that it is valid. A snack bar message will appear, showing whether loading the configuration has been successful. If the configuration has been loaded successfully, a copy button will appear that copies the URL into the clipboard. The copied URL is slightly different as it contains `/r/r` at the end of the URL. This argument triggers a redirect so that after loading this configuration, the root page is shown. This URL is intended to be shared with the students. It is structured as follows.

`www.example.com/c/<config>/h/<checksum>/r/r`

Instead of sharing a configuration URL, sharing the configuration file and letting the students import it themselves is also possible. However, sharing a clickable URL might be more convenient. An URL shortener service may enhance usability, as the configuration URLs can get very long for large configurations. The next section explains how to create a configuration.

5.5.2 Creating a configuration file

A configuration file follows a specific structure. In Appendix C, such a file is shown. The file contains only one attribute `chapters` at the highest level. The `chapters` attribute expects an array of chapters, as the name suggests. One chapter must be given a type and an array of exercises. A configuration containing the compression chapter with no exercises is shown in Listing 3. The type attribute must contain the type string of the enum `ChapterType`. In theory, one could allow different custom chapters, but this would have a significant disadvantage. We would end up in configuration files that need to be translated into different languages. By restricting the configuration file to contain a set of predefined chapters, we can create a configuration file and use it in all available languages. Also, it would not make sense to create custom chapters that are not contained in the textbook this project is based on.

To create a new configuration, it is advisable take an existing configuration and then to adapt it. With this approach, it is easier to keep a proper structure and missing opening and closing JSON characters as {, }, [, or] is less likely to happen.

An exercise configuration must contain a type argument specifying the type string of the exercises and a `typespecifics` section as shown in Listing 4.

Listing 3 A configuration containing one empty compression chapter.

```
{
  "chapters": [
    {
      "type": "compression",
      "exercises": []
    }
  ]
}
```

Listing 4 The general exercise configuration structure.

```
{
    "type": "<exercise-type>",
    "typespecifics": { }
}
```

Listing 5 The available exercise types.

```
export enum ExerciseType {
    DrawShannonFano = 'draw-shannon-fano',
    DrawMaxBalance = 'draw-max-balance',
    DrawHuffman = 'draw-huffman',
    SuboptimalReverseMaxBalance = 'suboptimal-reverse-max-balance',
    ReverseHuffman = 'reverse-huffman',
    PerfectReverseHuffman = 'perfect-reverse-huffman',
    Bin2Dec = 'binary-to-decimal',
    Dec2Bin = 'decimal-to-binary',
    Interval = 'interval',
    ArithmeticCoding = 'arithmetic-coding',
}
```

The different type strings of the exercise configurations are listed in Listing 5. In the following subsections, we show how configurations can be created for the different exercise types.

Draw Shannon-Fano, Max-Balance, and Huffman

The configurations for the tree-drawing exercises can only be distinguished by the type attribute. Their `typespecifics` section is identical and lists the probabilities of the letters. An example is shown in Listing 6. The specified letter frequency distribution must sum up to 100. Otherwise, the configuration gets rejected on import. Technically, it is also possible to specify floating-point numbers as letter frequencies. However, the GUI is not optimized for floating-point numbers, and therefore it may look strange.

Listing 6 An example letter frequency distribution.

```
"typespecifics": {
    "frequencies": {
      "A": 30,
      "B": 12,
      "C": 19,
      "D": 21,
      "E": 8,
      "F": 10
    }
}
```

Listing 7 An example of a tree declaration.

```
"typespecifics": {
    "tree": {
        "left": {
            "left": "A",
            "right": "B"
        },
        "right": {
            "left": "C",
            "right": {
                "left": "D",
                "right": "E"
            }
        }
    }
}
```

The letters specified in the configuration must consist of one single character. In principle, any character is allowed, but it is highly recommended to use letters. The letters are all capitalized on loading the configuration.

Reverse Tree exercises

The reverse tree exercises consist of three distinct types: `reverse-huffman`, `perfect-reverse-huffman`, and `suboptimal-reverse-max-balance`. The type-specific section consists of a tree declaration, as shown in Listing 7. It starts with a `tree` attribute that must contain a `left` and `right` attribute. Each of those attributes can either contain a letter (no special characters!) or an object containing a `left` and `right` attribute again. The letters must be single characters that will get capitalized on loading the configuration.

Listing 8 An example of a binary to decimal exercise configuration.

```
"typespecifics": {
    "binaryNumbers": [
        "0.1",
        "0.01",
        "101.111"
    ]
}
```

Listing 9 An example of a decimal to binary exercise configuration.

```
"typespecifics": {
    "decimalNumbers": [
        0.25,
        0.875
    ]
}
```

Table exercises

Examples for the configurations of the three table exercises, where numbers must be converted or the shortest representations must be found, are shown in the Listings 8, 9, and 10.

The binary to decimal exercise must contain an array `binaryNumbers` that contains binary numbers as strings. It is important not to include numbers in the number format because these numbers may not be transformed into precise machine numbers. However, the application prevents this by displaying an error.

The decimal to binary exercise accepts the decimal numbers as numbers and as strings in its `decimalNumbers` array. We do not have precision problems in this exercise, as the given numbers can be converted into reasonably sized binary numbers. If the binary number representation has more than 23 bits, we might also have a precision loss. However, a binary number with that many bits is didactically worthless.

The shortest binary number exercise expects an array `intervals` in its type-specific section. Each entry of the array is again an array consisting of two numbers. The first number denotes the minimum value of the interval (inclusive), and the second number is the upper bound (exclusive) of the interval. The application ensures that only intervals are accepted where $0 \leq l < u < 1$, l is the first, and u is the second number.

Listing 10 An example of a shortest binary number exercise configuration.

```
"typespecifics": {
    "intervals": [
        [0.3, 0.5],
        [0.7, 0.8]
    ]
}
```

Listing 11 An example of an arithmetic coding exercise configuration.

```
"typespecifics": {
    "encodingNumerator": 135,
    "encodingDenominator": 1000,
    "messageLength": 4,
    "frequencies": {
        "A": 32,
        "B": 27,
        "C": 21,
        "D": 20
    }
}
```

Arithmetic coding exercise

The arithmetic coding configuration must specify the `encodingNumerator`, the `encodingDenominator`, the `messageLength`, and the `frequencies`. The encoding numerator and denominator specify the interval together with the message length. The frequencies must follow the same format as in the tree-drawing exercises. It is recommended to choose only values greater or equal to 20. Otherwise, the bounds in the graphical representation may overlap. Listing 11 shows an example configuration for arithmetic coding.

Random creation

The application supports randomly created exercises. In Chapter 4, we explained how each exercise type is generated randomly. The random generation is based on a seed so that the exercises are reproducible and sharable. Section 5.4.3 discussed how the seed value is embedded in an URL. It is also possible to include the seed in a configuration file. A seed can be a number or a string and can be placed at the highest level of an exercise configuration as shown in Listing 12. If a seeded value is provided, then the type-specific information is ignored and therefore can be omitted.

As JavaScript does not offer a seeded random generator, we implemented a linear congruential generator as defined in [18]. For the frequency gener-

Listing 12 An example of a seeded arithmetic coding exercise configuration.

```
{
    "type": "arithmetic-coding",
    "seed": 152835
}
```

ation, we want Gauss distributed values, and therefore we added support for those with the Box-Transform [7]. The random generator is located in `random.ts` in the general utils directory.

5.6 State

We call the result of user interactions in an exercise the *state*. The state defines to which extent an exercise has been solved. The state history includes the redo/undo stack and can be exported to a file. This file can be imported again. This functionality allows students to interrupt solving the exercises and later on continue where they left off. It can also be used to share a solution with others or hand in a solution to the teacher.

An exported state is a JSON formatted object structure as defined by the interface `ExternalFileState` in Listing 13. The exported state contains a checksum that is added by the application. This is used to detect modified states to prevent students from trying to submit states that have been tampered with. A hash of the current configuration is added to the state so that the application can detect whether the state matches the current configuration on import. If the current configuration does not match, the user must first load the configuration and then import the state. The external file state contains a list of chapter and exercise IDs together with its state history. The state history consists of a list of the exercise's states and the state index. The state index is the index of the current state used for the undo/redo functionality. The exercise state interfaces in Listing 14 define an exercise state's structure. All the data structures in the state objects must be writable and readable to and from JSON formatted files. Appendix D shows a complete example of a state where the first exercise has three states in its state history, and the last exercise has one saved state.

When the user exports the current state, the following steps are performed by the application.

1. A new `ExternalFileState` object is created.
2. The `configHash` property is set to the hash of the current configuration and the `checksum` property is set to zero.
3. For each exercise, an `ExternalFileStateElement` is created and added to the `states` property.

Listing 13 The external state structure.

```
export interface StateHistory<S extends ExerciseState> {
    states: S[];
    stateIndex: number;
}

interface ExternalFileStateElement<S extends ExerciseState> {
    chapterID: number;
    exerciseID: number;
    stateHistory: StateHistory<S>;
}

export interface ExternalFileState {
    configHash: number;
    states: ExternalFileStateElement<ExerciseState>[];
    checksum: number;
}
```

4. A string is generated from the external file state object using the JSON `stringify` method. This string is then hashed, and the result is written into the checksum property.
5. The final external file state object is then written into a downloadable text file.

The import of a state consists of the following steps.

1. Check that the `configHash` property contains the hash of the current configuration. If this is not the case, the import is stopped, and the user is requested to load the corresponding configuration first.
2. Extract the `checksum` property of the external file state object and replace it with zero.
3. Compute the hash of the string of the object and compare it to the extracted checksum. If they are not equal, the import is aborted as the file must have been modified outside the application.
4. The state histories in the current configuration are replaced by the state histories given in the external file state object.

Listing 14 The state declarations of the different exercise types.

```
export interface ExerciseState {
    correct?: boolean;
}

export interface DrawTopDownState extends ExerciseState {
    topDownTreeState: TopDownTreeState;
}

export interface DrawHuffmanState extends ExerciseState {
    bottomUpTreeState: BottomUpTreeState;
}

export interface CreateFrequencyState extends ExerciseState {
    letterFrequencyDistribution: { [id: string]: number; };
}

export interface Bin2DecState extends ExerciseState {
    decimals: (number | undefined)[];
}

export interface Dec2BinState extends ExerciseState {
    binaries: string[];
}

export interface IntervalState extends ExerciseState {
    shortestNumbers: string[];
}

export interface ArithmeticCodingState extends ExerciseState {
    message: string;
}
```

5.7 Context

An essential module of this application is the context service. It provides some context to different components. For example, it holds the feedback for the currently displayed exercise. The header component updates its state based on the context service.

The context service also provides the snack bar functionality, which is used to notify the user. Exercise feedback, success or error messages are shown via snack bar. The context service handles the show duration of the messages and provides different styles. A success message is styled differently compared to an error message.

The undo/redo functionality is located in the main app component. However, those events need to be propagated to the currently shown exercise. This is again done with the context service, which provides an observable where the redo and undo events can be observed.

5.8 Router

Angular includes the router in the front-end. Therefore it must be configured to show the correct components for different URLs. Listing 15 shows the configuration for the different pages. As routes are handled in the front-end, the web server must be configured to handle this. The project's README file, which can be found in Appendix A, provides more information about the deployment.

5.9 Adding more educational content

This section discusses the technical steps to add more chapter or exercise types. A short explanation is given on how the exercise types must be registered and where chapter-specific files should be placed.

5.9.1 Adding more exercise types

This subsection contains all the information that is needed to create a new exercise type. Note that adding a new *instance* of an exercise is discussed in Section 5.5.

Every exercise type needs to be added to the `ExerciseType` enum shown earlier in Listing 5. An exercise type must have a main component, which must be a subclass of `GeneralNumberConversionComponent`. The new exercise type component must be registered in `exercise-component-mapping.ts` shown in Listing 16. The component should be placed in the exercise folder

Listing 15 The route configuration for the different URLs.

```
const routes: Routes = [
  { path: '', component: BookOverviewComponent },
  {
    path: 'chapter/:chapter/exercise/:exercise',
    component: ExerciseContainerComponent
  },
  {
    path: 'chapter/:chapter/type/:type/seed/:seed',
    component: ExerciseContainerComponent
  },
  {
    path: 'chapter/:chapter/type/:type',
    component: ExerciseContainerComponent
  },
  {
    path: 'chapter/:chapter',
    component: ChapterOverviewComponent
  },
  {
    path: 'c/:conf/h/:checksum/r/:redirect',
    component: BookOverviewComponent
  },
  {
    path: 'c/:conf/h/:checksum',
    component: BookOverviewComponent
  },
  { path: '**', component: PageNotFoundComponent },
];
```

of the chapter's directory, e.g., the component files of the arithmetic coding exercise type are located in `chapters/compression/arithmetic-coding/`.

The exercise type needs a type declaration (an interface) for its state. The state interface can be added to `state.ts`.

If the exercise should appear in the random exercise type list shown in Figure 5.2, then it must be added in `getRandomExerciseTypes()` of the class `Chapter`.

5.9.2 Adding more chapters

Adding new chapters does not require many steps. Creating a directory with the chapter name in the `chapters` directory is recommended. The

Listing 16 The mapping from exercise type to the angular components.

```
export function getComponent(type: ExerciseType):
        Type<GeneralExerciseComponent<ExerciseConfig<ExerciseState>,
            ExerciseState>> {
    switch (type) {
        case ExerciseType.DrawShannonFano:
            return TopDownComponent;
        case ExerciseType.DrawMaxBalance:
            return TopDownComponent;
        case ExerciseType.DrawHuffman:
            return HuffmanComponent;
        case ExerciseType.SuboptimalReverseMaxBalance:
            return SuboptimalReverseMaxBalanceComponent;
        case ExerciseType.ReverseHuffman:
            return ReverseHuffmanComponent;
        case ExerciseType.PerfectReverseHuffman:
            return PerfectReverseHuffmanComponent;
        case ExerciseType.Bin2Dec:
            return BinaryToDecimalComponent;
        case ExerciseType.Dec2Bin:
            return DecimalToBinaryComponent;
        case ExerciseType.Interval:
            return ShortestBinaryNumberComponent;
        case ExerciseType.ArithmeticCoding:
            return ArithmeticCodingComponent;
    }
}
```

exercise components and utils of this chapter should be placed inside this directory. It is recommended to write a short description of the chapter in the docs folder (see Appendix B).

The second step of adding a new chapter is registering it in the enum `ChapterType`. The `getChapter` method of the class `Chapter` should be updated to return a localized string with the chapter name.

Chapter 6

Testing

Keeping the code quality high is a top priority in this project. Correct feedback is an essential component of educational software. We have attached importance to well-structured and organized code in order to achieve high quality. The second major measure to achieve correctness is unit testing. The tests should be run frequently. The angular command `ng test` opens a browser window and displays the results of the tests. The command monitors changes to the code and re-runs the tests based on the changes. These immediate test results help detect bugs very early. The code coverage is calculated every time the tests are run. For the project, we set a goal of at least 80 percent branch, line, function, and statement coverage.

6.1 Coverage

An essential fundament of code quality is testing. Therefore, we have put great effort into unit testing. Angular conveniently supports the Karma framework [33] and Jasmine unit tests [19]. We designed good tests to ensure correctness. We use white-box testing for testing the structure of the program. The tests cover over 90 percent of the statements, functions, and lines. The achieved branch coverage is around 81 percent. However, this value is mainly lower because we have many type checks due to the nature of the TypeScript language. With our tests, we meet our goal of 80 percent coverage. Many tests are needed to achieve this high coverage. In our case, approximately a third of the TypeScript code base is tests.

6.2 Exercise Components

The exercise type component's tests simulate the user's actions. The tree drawing exercises include emitting drag&drop mouse and touch events to test the tree's response. The arithmetic coding tests emit clicks and check

Listing 17 An excerpt of the arithmetic coding component tests.

```
it('should react to mouse moves and clicks', fakeAsync(async () => {
  // Initialization code omitted for this listing

  // Move mouse and click elements. Afterwards check, that the
  // right message was set.
  dispatchMouseMove(canvas, 544, 135);
  dispatchMouseMove(canvas, 600, 131);
  dispatchMouseMove(canvas, 775, 126);
  dispatchMouseMove(canvas, 525, 120);
  dispatchMouseClick(canvas, 526, 121);
  tick(100);
  expect(component.message).toBe('B');

  // Code omitted for this listing
}));
```

that the correct message gets selected. An example of an arithmetic coding test is shown in Listing 17. The exercise components are tested at least with three different conditions: with the state with the correct solution, with a state with a wrong solution, and without a state. This testing procedure guarantees that the exercise can be interrupted and later be restored. We test the `checkSolution` method very intensively for every exercise component as it is of utter importance. Part of these tests is a check that the feedback for the different states is appropriate.

6.3 Solution Checking

The implementation of checking Shannon-Fano, Max-Balance, and Huffman trees is located in separate files. This implementation is crucial for the correctness of the exercises. The exercise component's tests only perform a few tests checking that the trees are graded correctly. These tests focus on the interaction and the feedback for the user. However, the logic behind the trees is tested carefully at a lower level. Many tests check corner cases. In the case of Shannon-Fano and Max-Balance, we implemented the following checks.

- *Small trees:* These tests check that small trees with just one or two letters are graded correctly.
- *Correct trees:* We have many correct trees, including huge ones, that are tested.
- *Incorrect trees:* It is not sufficient to test that correct trees are accepted.

We need tests that check that faulty trees are rejected. Therefore, we prepared several mistaken trees to check against our implementation to see whether our solution reviewing implementation detected those errors.

- *Incomplete encodings:* The trees of that category are missing some letters. Let us assume that we have a letter frequency distribution for the letters A-E. Then, we create a tree that would be correct regarding Shannon-Fano or Max-Balance, but it is missing a letter, e.g., A. The solution checking algorithm must detect that.
- *Invalid letters:* We add at least one letter that does not appear in the letter frequency distribution in these tests. For the above example, this could be the letter F.
- *Double letters:* In those trees, we add at least one letter twice or even more times. This invalid tree structure must not crash the system or trigger the wrong feedback. Therefore, we test that the corresponding error is shown for this kind of tree.
- *Wrong Shannon-Fano or Max-Balance:* In those categories, we have several trees that fulfill the syntactical properties of a tree but contain errors regarding the Shannon-Fano or Max-Balance criteria. Therefore, we test that these trees are reported to be incorrect. The GUI should ensure that only this type of error is possible at all[1].

The Huffman solution checking tests are structured similarly. The solution checking for the decimal and binary conversion exercises and the interval exercise is more straightforward and therefore handled and tested in the exercise components themselves. The arithmetic coding math is, similarly to Shannon-Fano or Huffman, put into separate files. The tests for the arithmetic coding exercise are divided into user interaction tests and logic tests. An example of those user interaction tests is shown in Listing 17. The logic tests check that the interval bounds of the arithmetic coding implementation are set correctly.

6.4 Layout Components

The layout components are tested accordingly. We test loading different configurations, valid and invalid checksums, incomplete arguments, no arguments, and redirection regarding the book and chapter overview compo-

[1] This requirement is part of the exercise component tests. The GUI ensures that a letter cannot appear multiple times in a tree or invalid letters appear in it. Nevertheless, we do test for the errors listed above so that the solution checking software could potentially be used in other contexts.

nent. The main focus of the layout components' tests is the correct navigation between the components or exercises.

6.5 Other tests

For almost every TypeScript file, we provide a corresponding test file containing specific unit tests. In this section, we are going to highlight the most critical tests.

6.5.1 Random Generator

The random generator is fundamental to create an exercise given a seed. Therefore, we have tests that guarantee that independent random generator objects output the same values for the same seed. We also test that the generated values are inside the specified bounds.

6.5.2 Rational Numbers

Our implementation of arbitrarily precise rational numbers is tested carefully as well. This implementation must work as expected as it is the basis of arithmetic coding. Therefore, we have several tests for rational numbers.

- *Comparison:* The comparison between two rational numbers is essential in arithmetic coding. Hence, these tests are particularly comprehensive.
- *Equality:* We designed specialized tests to check the equality functionality of the rational numbers.
- *Transformation to machine precision floating-point numbers:* To display the numbers to the users, we need to transform the rational numbers to decimal floating-point numbers. As machine precision is limited, we check that this transformation works without too large deviations.
- *Basic operators:* We thoroughly test the following operations.
  - addition
  - subtraction
  - multiplication
  - division
  - abs
  - sign

6.5.3 Exercise Configuration

One of the most extensively tested classes is the exercise configuration class. The primary purpose of those tests is to check that the different exercise instantiations based on the JSON configuration files are correctly implemented. The tests provide many different exercise configurations that contain minor errors. These tests guarantee that the exercise configuration class only accepts correct exercise configurations and returns the corresponding feedback. When, e.g., teachers create those files and want to import them, it is crucial to report the exact location where the error has happened. Therefore, we have put great effort into testing this particular piece of code intensively with more than 500 lines of test code.

Chapter 7

Evaluation

To evaluate the application, we conducted two user studies to gather feedback. The surveys can be found in Appendices F and G. Overall, the feedback has been very positive. The quality of being easy to use, automatic scaling of trees, and the clarity of the tree representations are the main positive points of the feedback from the first user study. The surveys also lead to various helpful hints and could, therefore, significantly improve the application. Some resulting improvements are using a different mouse handle for drag&drop actions, removing up/down arrows in binary input fields, and applying different colors in the top-down trees. In the second user study, the results were also very positive. Only a few suggestions for improvement were mentioned. The short number representation of the barriers in the arithmetic coding exercise led to some confusion, and the feedback for the reverse Max-Balance tree exercise was considered too unclear.

As an excellent reference to reflect on educational software, we can use the criteria defined by Raimond Reichert and Werner Hartmann [23]:

- *Content Based on Fundamental Ideas:* As the production of good educational content is expensive, the creation of long-lived content is important. The textbook defines the educational material of this thesis, and its selection is not part of this thesis in contrast to the selection of the exercise types. However, the exercise types do not introduce new material as they are an addition to the textbook and therefore do not introduce new content.
- *Incorporating Different Cognitive Levels:* Benjamin Bloom described six cognitive levels in his taxonomy [5]: knowledge, comprehension, application, analysis, synthesis, and evaluation. Software for educational purposes should target the high levels in particular. In the following, we analyze our tool in terms of these six levels. The cognitive level of *knowledge*, i.e., a test whether a student has memorized specific infor-

mation, is not covered by our application. No exercise task asks for knowledge that consists of memorizing dates, names, grammar rules, or anything of this kind. Similar to the previous cognitive level, *comprehension* is not a clear part of our tool. All the exercise types of this work start at this level. They are all exercises that expect the students to solve tasks by *applying* the knowledge of the textbook. The *analysis* level is used in the reverse Max-Balance exercise. In this exercise type, the students are expected to analyze the compression quality of Max-Balance. Additionally, they need to find a solution that explicitly uses the gained information of the analysis. This advanced task also has some aspects of the cognitive level *synthesis*, where students are expected to get information from facts and draw their conclusions out of it. Therefore, we can conclude that our application covers several different cognitive levels, although we do not reach the highest level, the *evaluation*.

- *High Degree of Interactivity:* Our reflection regarding interactivity is based on Rolf Schulmeister's six levels of interaction [25].

  1. *Viewing Objects and Receiving:* Except for the simple number conversion exercises, this level is reached by every exercise type. As objects, we have trees for the reverse exercises and tree fragments for the tree drawing exercises. In the arithmetic coding exercise, the bar is the shown object.

  2. *Watching and Receiving Multiple Representations:* As we have exercise types for the topic compression, this leads automatically to at least two different representations – the original message and the compressed version. However, these two different representations are only reached towards the end of solving the exercise, as constructing the second representation is part of the exercise. Building the Huffman tree, for example, is basically constructing a second representation of the letter frequency distribution. We are given the encoded message in arithmetic coding, consisting of the rational number and the message length. While solving the exercise, two different representations are constructed. On the one hand, we have the bars construct, where the students click them through, and on the other hand, we have the actual message string.

  3. *Varying the Form of Representation:* This level is the first one, where the user gets the actual feeling of interacting with the system. As explained in Level 2, this is the actual exercise solving process. For arithmetic coding, it is only clicks, but for Shannon-Fano, Max-Balance, and Huffman, changing the representation also includes drag&drop.

4. *Manipulating the Component Content:* While in the previous level, the users changed the form of the content, i.e. the *representation*, this level includes changing the actual *content*. The tree drawing exercises, in some sense, are mainly working on changing the representation, but not the content, as the system prevents illegal node movements. However, this can also be seen as content changes, as three subtrees are different from two subtrees in the context of Huffman, for example. There is content manipulation going on in the reverse tree exercises, as trees get drawn depending on the entered numbers. In arithmetic coding, the content also can be changed by selecting different messages.

5. *Constructing the Object or Representation Contents:* This level describes the essential parts of all the exercises. Each exercise aims to construct something, whether it is a tree fulfilling specific construction rules, a letter frequency distribution that leads to a particular tree, or a text message.

6. *Constructing the Object or Contents of the Representation and Receiving Intelligent Feedback from the System through Manipulative Action:* Feedback is a relevant part of the application, and therefore every exercise type provides feedback for the current state. Therefore, our application's exercises fulfill the six levels of interactivity.

- *Feedback:* We have implicit feedback for every exercise type. A symbol displaying whether the exercise has been solved correctly or not is always shown. For some exercise types, e.g., in reverse Huffman, we have explicit feedback. Figure 7.1 shows the Huffman tree, where the student has made a mistake. The wrong "decision" is marked red to indicate that the values of C and D are too large compared to the other values (in this case of A and B).

- *Visualization and Usability:* The operation of the application should not be its focus. Students should not need to spend time on learning *how* to use the application. To reduce this amount of time to a minimum, we have taken several measures. One of the essential paradigms was reducing the number of GUI elements to a minimum to reduce visual clutter. Buttons and help texts that are only needed from time to time do not need to be shown all the time and disturb the students. Another design choice is to adhere to standard practices for user interfaces. For example, a help text for an element can be shown when hovering over it. Different cursor styles may help find an element that can be dragged, and visual hints define where it can be dropped. A consistent menu structure and the use of standard HTML elements are other vital factors in boosting usability. This consistency is achieved by using widely used material design HTML elements and a clean tree

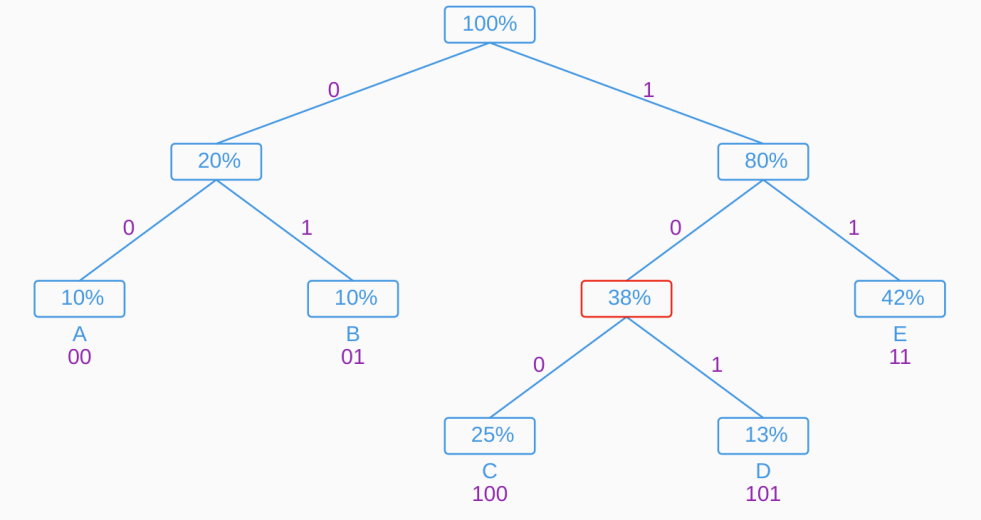


Figure 7.1: The Huffman tree where the student has entered some frequencies.

representation throughout the application. Consistent colors also help provide some context to elements without taking too much space. For example, values used for the encodings are all displayed in purple.

Anaraki [4] lists some of the most common problems in eLearning systems. Especially, the following points are mentioned.

- *Text-based learning materials:* Many eLearning systems contain mainly text-based material. Often they lack good graphics and figures. Sometimes the assignments are just simple multiple-choice questions that check for some knowledge of the script or book. In our work, this is not the case at all. We tried to reduce the number of visible texts to reduce visual clutter. Also, the exercise development focused on designing interactive exercises where the knowledge must be applied. Therefore, we can exclude this issue from our environment.
- *Lack of rich content for good understanding:* Some eLearning systems only provide PowerPoint slides that might not be enough for a deep understanding of the content. This critique cannot be applied to our work, as we only intend to provide additional exercises that apply the theory, but we do not provide extensive explanations of the theory behind it.
- *Insufficient interactivity or flexibility:* Many eLearning systems do not provide the classical interactivity with teachers as classroom learning provides. It is much more time-consuming for students if they need to watch instruction videos or read manuals instead of just asking for the precise information they need at some point. Our system is not designed to replace classroom learning – on the contrary – it can be used explicitly there. Therefore, we do not need to replace the teacher

interaction. However, our tool handles some of the teacher's responsibilities: giving helpful feedback to the students. In this sense, our tool is highly interactive, as for every action a student makes, feedback is available.

- *Unstructured and isolated multimedia instructions:* This point also mainly concerns the provision of learning material and does not necessarily apply to pure exercise collections.

Chapter 8

Conclusion

In this chapter, we first conclude our work and state the main outcomes. Then we discuss additional features that could be implemented in the future.

8.1 Conclusion

This thesis has three primary goals, as described in the introduction. The first objective is to create an easily extensible and maintainable platform-independent application. Maintainability is given as the latest and prevalent technologies are used. Angular is still widely supported and under active development to keep up to date regarding security and device support. It is widely used in practice, and therefore it is unlikely that further Angular updates break the system or require considerable effort to migrate the project to the newer version.

Platform independence is given as the device compatibility is ensured by the web browser implementation of the devices. The application has been tested on Windows, macOS, Linux, Android, and iOS. Chrome, Edge, Firefox, and Safari do work well with the application. With the support of those browsers, almost no Internet-capable device is excluded. Additionally, many other browsers are based on either Chrome's or Firefox's javascript engine, raising the chance to be compatible.

The codebase is kept clean, concise, reusable, well documented, and thoroughly tested. There is documentation provided on how to add new chapters and exercise types. Therefore, ease of extensibility should be granted. However, we will only know this with certainty once the first more considerable extension is complete.

The second goal is to find suitable additional exercises for the chapter "compression" and the third goal is to implement those. The textbook has not yet been used in schools since it has not yet been finished. Therefore, it is not

possible to test the application in schools since the theoretical background has not yet been taught. Therefore, the achievement of the second goal cannot be guaranteed, but at least strongly suspected by the tests within the ABZ. The user surveys inside of ABZ show that the application is usable, and the implementation looks pretty prosperous. Nevertheless, the evaluation of the application holds the most significant potential for optimization. Much more effort could have been spent on testing the real-life use in classrooms, resulting in further adjustments.

Supporting different platforms can be challenging – especially when working with touch screens or custom input fields for numbers. While this is not particularly surprising, it is nevertheless a remarkable finding. Supporting touch screens across multiple browsers takes considerably longer than expected, since each browser interprets touch screens differently. The browsers also behave differently when it comes to input field bounds and steps. It is worthwhile to schedule more time for such seemingly trivial problems for similar projects.

Another important lesson learned from the thesis is that unit testing is essential. The unit tests helped to maintain the correctness of the code. For the motivation of students to learn, providing correct feedback is crucial. Unit tests helped detect code changes that led to inaccurate exercise feedback.

8.2 Future Work

In this section, we present different further ideas that could be implemented.

More Chapters and Exercises

An obvious extension is adding more chapters. To make this additional application more attractive, it should provide interactive exercises for several if not all chapters of the textbook this application is based on.

Graphical Exercise Creation

The process of creating a new configuration graphically within the browser might be easier than writing a configuration file manually. The GUI could be easier to use, especially when creating a completely new configuration. Additionally, every task related to this application could be accomplished without leaving the application and switching to another editor or tool. However, it is expected that personalized configurations will be created quite rarely and therefore this may not be the most important feature.

Playground

A playground functionality could enhance our application. The students are free to add and remove letters as they wish and assign frequencies to them. The framework would then generate Shannon-Fano, Max-Balance, and Huffman trees. A comparison criterion could be provided by displaying additional information such as the average number of bits per letter. The students could input a message and view the encoding with different trees as well as with arithmetic coding.

Difficulty Level Indicator

Each exercise is different in difficulty. This might unsettle weaker students. Students who struggle with more challenging exercises might benefit from an indicator of the difficulty level that helps them focus on more straightforward and more elementary tasks before attempting the harder ones. Strong students could focus more on challenging exercises. As both the exercise type and the exercise instance can differ in difficulty, it may make sense to integrate the difficulty level in the configuration.

Login and Online State Storage

A login functionality could help students by making the state savable in a more convenient way. Saving files would no longer be necessary to accomplish this. Disadvantages are the significantly increased implementation and deployment efforts and another website login for students to manage. Additionally, we would need to deal with data protection and privacy.

Direct Submission

The submission process could also benefit from a login function. Teachers could access all the hand-ins of theirs students in a central place directly in the tool. Students might benefit from the feedback of the teacher inside the framework.

Examinations

An examination mode is not part of the goals of this thesis but may be implemented in the future. Students could solve the provided exercises and get graded after submitting those. The immediate feedback would be disabled to prevent the students from solving the exercises with trial and error.

Background Information

This application contains additional exercises without explaining the theory and its background. A possible extension could also include the theory so that this tool could work without additional information. For every exercise type, there could be a section explaining the theory. Examples and background information from the textbook could be provided within this application.

Chat and Forum

There is no collaboration between the students in this tool. An integrated chat or forum could be helpful for students to get help or tips from others. Students could read frequently asked questions or help others by answering their questions. Discussions about the quality of their solutions or possible improvements could take place in such a chat or forum.

Keyboard Shortcuts

The application supports only two keyboard shortcuts for the undo and redo functionality. The keyboard shortcuts can be further extended. For example, a keyboard shortcut could switch between exercises or reset the current state. The drawing exercises are designed to use with a mouse or touchscreen. Therefore, keyboard shortcuts may not be helpful or intuitive to use there.

Statistics

One could extend this website to collect user metrics. The time spent per exercise and student could be tracked. Common mistakes or errors could be detected and reported to the teacher automatically. This would allow the teacher to repeat or clarify some topics. It could also be a good indicator of how laborious a particular exercise is. Some statistics could even be helpful to improve the application.

Appendix A

README file of Code Repository

Gymnasium

This project is a web environment for interactive exercises as an addition to a computer science textbook for secondary schools. It provides intuitively solvable exercises.

Features

Exercise Feedback

The tool guides the students to the correct solution of an exercise. For students who are on the wrong track, helpful feedback is displayed that points out the mistake.

Graphical exercises

The exercises are presented in a modern and clean look, and they are designed to visualize the theory graphically.

Touch support

The exercises support touch gestures so that they can also be used on tablets.

Configurable

The exercise collection can be easily configured and deployed. Various exercise sets can be offered to meet the individual student's needs.

Chapters and Exercises

Currently, there is one chapter, Compression, available. More information about the chapter can be found here.

User Manual

Navigation

The main view of the application shows the different available chapters. There is a navigation sidebar on the left side, where every single exercise per chapter is shown. This is also the place where students can switch to the next exercise. The navigation sidebar can also be hidden with the button in the top left.

Actions

In the header bar, there are different possible actions (from left to right):

- The navigation bar toggle button hides or shows the left navigation sidebar.
- For some exercise types, a reset option (trash bin icon) is provided. It resets the current state of the exercise.
- The exercise feedback button is displayed as a check mark when the current state is correct. An exclamation mark is shown otherwise. A click on the feedback button shows feedback to the current exercise state.
- Import (in the drop-down menu) can be used to import an exercise state. It accepts files that have been exported before. This file must belong to the same configuration. Otherwise, it will be rejected.
- Export (in the drop-down menu) saves the current state into a file. If the file has been manually edited, it cannot be imported again (to prevent errors).

Defining the exercises

With a configuration file, the specific exercise can be created for the existing exercise types. The default configuration file is included in the application. Other specialized configurations (e.g., more challenging exercises) can be distributed by simply making the configuration file accessible to the students (cloud, e-mail, chat, etc.). Another possibility is to generate an URL (by importing the file into the app) and then distribute this link. The generated URL will contain all the necessary information.

Project Maintenance

Translation

This app is translated with the internationalization features of Angular.

The supported languages need to be specified in the `i18n` section of `angular.json`.

With `npm run update-i18n`, the localized strings are exported and merged with the existing translation file. Merge options can set in the `xliffmerge.json` file. The generated XLIFF file can then be translated with an XLIFF tool.

Development server

Run `ng serve` for a dev server. Navigate to `http://localhost:4200/`. The app will automatically reload if any of the source files are changed. With `ng serve --configuration de`, the app can be served (for development) in German.

Running unit tests

Run `ng test` to execute the unit tests via Karma. The tests are an essential part of the application. They are designed to report changes (possible errors) in the application as quickly as possible. Running them regularly (or keep them running in the background) is highly recommended.

Build and Setup

The logic used in the application is all done on the client-side. Therefore, this application needs to be deployed on a static web server (see below for more details). A default 'configuration', i.e., a set of predefined chapters and exercises, is included in the application.

Build

Run `ng build` to build the project. The build artifacts will be stored in the `dist/` directory. Use `ng build --configuration production` for the production build.

Deployment

For the deployment, you may need to specify the base href with the additional build flag `--base-href /path/to/app/`. The server needs to be configured to return the index.html file when a file is requested that does not exist. This is due to the Angular routing, which is done completely on the client-side. For more information, please check out the Angular deployment guide.

Further help

To get more help on the Angular CLI, use `ng help` or go check out the Angular CLI Overview and Command Reference page.

Appendix B

Short Chapter Documentation

Compression

The main topics of the chapter compression are Shannon-Fano and Huffman trees and Arithmetic Encodings. This platform provides the following nine additional exercise types.

Draw Shannon-Fano Tree

In this exercise type, a letter frequency distribution is given, and the students are expected to draw the resulting Shannon-Fano tree. A tree skeleton is shown, and the letters need to be split into two groups (left and right) with drag & drop (as in Shannon-Fano).

Draw Max-Balance Tree

This exercise looks the same as the Shannon-Fano exercise, but the nodes must be split according to the maximum balance criterion. The sum of the frequencies must be maximally balanced on the right and left sides.

Draw Huffman Tree

Similar to the previous exercise types, the students are expected to draw a Huffman tree based on a given letter frequency distribution. This is achieved again with drag & drop, i.e., the two smallest nodes need to be dragged on each other to merge them to a subtree.

Reverse Max-Balance

The students need to assign the frequencies (in %) to the letters so that the shown tree is the Max-Balance tree of the frequency distribution. An additional constraint is that the Max-Balance tree should not be as good as

the resulting Huffman tree. The resulting Huffman tree is also shown. The table contains additional information like the encodings and the average encoding lengths. This is especially interesting to see how Max-Balance and Huffman differ in terms of optimality.

Reverse Huffman

In this exercis, the goal is again to find a letter frequency distribution. In this case, the letter frequencies should result in the shown Huffman tree. There aren't any other constraints, except that the shown tree must be the Huffman tree based on the chosen letter frequency distribution.

Perfect Reverse Huffman

This is the same exercise type as the one above, but there is an additional constraint. We look for a letter frequency distribution, such that the shown Huffman tree is also optimal amongst compression algorithms that do not compress each letter individually, e.g., arithmetic coding.

Binary to Decimal

This exercise shows a table with rational numbers in binary format. The goal of the exercise is to convert them into the commonly used decimal format. This is, together with the following two exercises, a preparation exercise for the arithmetic coding exercise.

Decimal to Binary

In contrast to the last exercise, the students are expected to write the decimal numbers in binary format.

Intervals

A similar table as in the last two exercises is shown here. However, the goal is to find the binary representation of a number inside a given interval so that it is the shortest possible number that lies inside the given interval.

Arithmetic coding

The goal is to find a message given its arithmetic encoding. The arithmetic encoding consists of the shortest binary number (in decimal format) in the target interval and the message length. The exercise framework does all the math: All the students need to do is click the intervals where the goal is included.

Appendix C

Configuration File Example

```
{
  "chapters": [
    {
      "type": "compression",
      "exercises": [
        {
          "type": "draw-shannon-fano",
          "typespecifics": {
            "frequencies": {
              "A": 30,
              "B": 12,
              "C": 19,
              "D": 21,
              "E": 8,
              "F": 10
            }
          }
        },
        {
          "type": "draw-max-balance",
          "typespecifics": {
            "frequencies": {
              "A": 42,
              "B": 3,
              "C": 18,
              "D": 21,
              "E": 7,
              "F": 9
            }
          }
```

```
},
{
  "type": "draw-huffman",
  "typespecifics": {
    "frequencies": {
      "A": 11,
      "B": 21,
      "C": 6,
      "D": 14,
      "E": 23,
      "F": 9,
      "G": 16
    }
  }
},
{
  "type": "suboptimal-reverse-max-balance",
  "typespecifics": {
    "tree": {
      "left": {
        "left": "A",
        "right": "B"
      },
      "right": {
        "left": "C",
        "right": {
          "left": "D",
          "right": "E"
        }
      }
    }
  }
},
{
  "type": "reverse-huffman",
  "typespecifics": {
    "tree": {
      "left": {
        "left": "A",
        "right": "B"
      },
      "right": {
        "left": {
          "left": "C",
```

```
          "right": "D"
        },
        "right": "E"
      }
    }
  }
},
{
  "type": "perfect-reverse-huffman",
  "typespecifics": {
    "tree": {
      "left": {
        "left": "A",
        "right": "B"
      },
      "right": {
        "left": {
          "left": "C",
          "right": {
            "left": "D",
            "right": "E"
          }
        },
        "right": {
          "left": "F",
          "right": "G"
        }
      }
    }
  }
},
{
  "type": "binary-to-decimal",
  "typespecifics": {
    "binaryNumbers": [
      "0.0",
      "0.1",
      "0.01",
      "0.101",
      "0.110",
      "101.111"
    ]
  }
},
```

```
{
  "type": "decimal-to-binary",
  "typespecifics": {
    "decimalNumbers": [
      0.25,
      0.125,
      3.5,
      0.375,
      0.875
    ]
  }
},
{
  "type": "interval",
  "typespecifics": {
    "intervals": [
      [
        0.3,
        0.5
      ],
      [
        0.7,
        0.8
      ],
      [
        0.6,
        0.8
      ],
      [
        0.6,
        0.7
      ],
      [
        0.1,
        0.12
      ],
      [
        0.1,
        0.5
      ],
      [
        0.7,
        0.74
      ],
```

```
            [
              0.2,
              0.25
            ]
          ]
        }
      },
      {
        "type": "arithmetic-coding",
        "typespecifics": {
          "encodingNumerator": 135,
          "encodingDenominator": 1000,
          "messageLength": 4,
          "frequencies": {
            "A": 32,
            "B": 27,
            "C": 21,
            "D": 20
          }
        }
      }
    ]
  }
 ]
}
```

Appendix D

Exported State Example


```
{
    "configHash": -1483024399,
    "states": [
        {
            "chapterID": 0,
            "exerciseID": 0,
            "stateHistory": {
                "states": [
                    {
                        "topDownTreeState": {
                            "children": [
                                {
                                    "letter": "A",
                                    "children": [],
                                    "frequency": 30
                                },
                                {
                                    "letter": "B",
                                    "children": [],
                                    "frequency": 12
                                },
                                {
                                    "letter": "C",
                                    "children": [],
                                    "frequency": 19
                                },
                                {
                                    "children": [
                                        {
                                            "children": [],
```

```
                                "frequency": 0
                            },
                            {
                                "children": [],
                                "frequency": 0
                            }
                        ],
                        "finalInnerNode": true,
                        "frequency": 0
                    },
                    {
                        "letter": "D",
                        "children": [],
                        "frequency": 21
                    },
                    {
                        "letter": "E",
                        "children": [],
                        "frequency": 8
                    },
                    {
                        "letter": "F",
                        "children": [],
                        "frequency": 10
                    }
                ],
                "frequency": 0
            }
        },
        {
            "topDownTreeState": {
                "children": [
                    {
                        "letter": "B",
                        "children": [],
                        "frequency": 12
                    },
                    {
                        "letter": "C",
                        "children": [],
                        "frequency": 19
                    },
                    {
                        "children": [
```

```
                        {
                            "letter": "A",
                            "children": [],
                            "leftNode": true,
                            "rightNode": false,
                            "frequency": 30
                        },
                        {
                            "children": [],
                            "frequency": 30
                        },
                        {
                            "children": [],
                            "frequency": 0
                        }
                    ],
                    "finalInnerNode": true,
                    "frequency": 30
                },
                {
                    "letter": "D",
                    "children": [],
                    "frequency": 21
                },
                {
                    "letter": "E",
                    "children": [],
                    "frequency": 8
                },
                {
                    "letter": "F",
                    "children": [],
                    "frequency": 10
                }
            ],
            "frequency": 0
        },
        "correct": false
    },
    {
        "topDownTreeState": {
            "children": [
                {
                    "letter": "C",
```

```
            "children": [],
            "frequency": 19
        },
        {
            "children": [
                {
                    "letter": "A",
                    "children": [],
                    "leftNode": true,
                    "rightNode": false,
                    "frequency": 30
                },
                {
                    "children": [],
                    "frequency": 30
                },
                {
                    "children": [],
                    "frequency": 12
                },
                {
                    "letter": "B",
                    "children": [],
                    "leftNode": false,
                    "rightNode": true,
                    "frequency": 12
                }
            ],
            "finalInnerNode": true,
            "frequency": 42
        },
        {
            "letter": "D",
            "children": [],
            "frequency": 21
        },
        {
            "letter": "E",
            "children": [],
            "frequency": 8
        },
        {
            "letter": "F",
            "children": [],
```

```
                            "frequency": 10
                        }
                    ],
                    "frequency": 0
                },
                "correct": false
            }
        ],
        "stateIndex": 2
    }
},
{
    "chapterID": 0,
    "exerciseID": 1,
    "stateHistory": {
        "states": [],
        "stateIndex": -1
    }
},
{
    "chapterID": 0,
    "exerciseID": 2,
    "stateHistory": {
        "states": [],
        "stateIndex": -1
    }
},
{
    "chapterID": 0,
    "exerciseID": 3,
    "stateHistory": {
        "states": [],
        "stateIndex": -1
    }
},
{
    "chapterID": 0,
    "exerciseID": 4,
    "stateHistory": {
        "states": [],
        "stateIndex": -1
    }
},
{
```

```
        "chapterID": 0,
        "exerciseID": 5,
        "stateHistory": {
            "states": [],
            "stateIndex": -1
        }
    },
    {
        "chapterID": 0,
        "exerciseID": 6,
        "stateHistory": {
            "states": [],
            "stateIndex": -1
        }
    },
    {
        "chapterID": 0,
        "exerciseID": 7,
        "stateHistory": {
            "states": [],
            "stateIndex": -1
        }
    },
    {
        "chapterID": 0,
        "exerciseID": 8,
        "stateHistory": {
            "states": [],
            "stateIndex": -1
        }
    },
    {
        "chapterID": 0,
        "exerciseID": 9,
        "stateHistory": {
            "states": [
                {
                    "message": "DB",
                    "correct": false
                }
            ],
            "stateIndex": 0
        }
    }
```

```
    ],
    "checksum": 435747168
}
```

Appendix E

Tests for Huffman Solution Checking

```
import { ExerciseFeedback, ExerciseFeedbackCorrect } from
    'src/app/general/exercise-feedback';
import { BinaryTree } from './binary-tree';
import { checkHuffman, getHuffmanTree, HuffmanError, HuffmanWrong }
    from './huffman';
import { LetterFrequencyDistribution } from './letter-frequencies';
import { getAsFrequencyTree } from './utils';

/**
 * Expects the feedback to be correct.
 * @param feedback the feedback
 */
function expectCorrect(feedback: ExerciseFeedback) {
    expect(feedback instanceof ExerciseFeedbackCorrect).toBeTrue();
}

/**
 * Expects the feedback to be wrong
 * @param feedback the feedback
 */
function expectWrong(feedback: ExerciseFeedback) {
    expect(feedback instanceof HuffmanWrong).toBeTrue();
}

/**
 * Expects the feedback to be an error (that should not be possible
 * to get with the GUI)
 * @param feedback the feedback
 */
function expectError(feedback: ExerciseFeedback) {
```

```
        expect(feedback instanceof HuffmanError).toBeTrue();
    }

    describe('HuffmanSolutionChecking', () => {
        const dicts: { [id: string]: LetterFrequencyDistribution; } = {
            'three': new LetterFrequencyDistribution(
                { 'A': 52, 'B': 29, 'C': 19 }),
            'four': new LetterFrequencyDistribution(
                { 'A': 70, 'B': 10, 'C': 10, 'D': 10 }),
            'five': new LetterFrequencyDistribution(
                { 'A': 20, 'B': 30, 'C': 12, 'D': 27, 'E': 11 }),
            'several_solutions': new LetterFrequencyDistribution(
                { 'A': 22, 'B': 22, 'C': 16, 'D': 11, 'E': 11, 'F': 9, 'G': 9 })
        };

        it('should correctly check small trees', () => {
            const tree = new BinaryTree(100);
            tree.left = new BinaryTree(52);
            tree.right = new BinaryTree(48);
            tree.right.left = new BinaryTree(29);
            tree.right.right = new BinaryTree(19);

            expectCorrect(checkHuffman(tree, dicts.three));

            const tmp = tree.left;
            tree.left = tree.right;
            tree.right = tmp;

            expectCorrect(checkHuffman(tree, dicts.three));

            tree.left = new BinaryTree(71);
            tree.left.left = new BinaryTree(19);
            tree.left.right = new BinaryTree(52);
            tree.right = new BinaryTree(29);

            expectWrong(checkHuffman(tree, dicts.three));

            tree.left = new BinaryTree(43);
            tree.left.left = new BinaryTree(20);
            tree.left.right = new BinaryTree(23);
            tree.left.right.left = new BinaryTree(12);
            tree.left.right.right = new BinaryTree(11);
            tree.right = new BinaryTree(57);
            tree.right.left = new BinaryTree(30);
```

```
    tree.right.right = new BinaryTree(27);

    expectCorrect(checkHuffman(tree, dicts.five));

    tree.left = new BinaryTree(30);
    tree.left.left = new BinaryTree(20);
    tree.left.left.left = new BinaryTree(10);
    tree.left.left.right = new BinaryTree(10);
    tree.left.right = new BinaryTree(10);
    tree.right = new BinaryTree(70);

    expectCorrect(checkHuffman(tree, dicts.four));

    tree.left = new BinaryTree(20);
    tree.left.left = new BinaryTree(10);
    tree.left.right = new BinaryTree(10);
    tree.right = new BinaryTree(80);
    tree.right.left = new BinaryTree(10);
    tree.right.right = new BinaryTree(70);

    expectWrong(checkHuffman(tree, dicts.four));
});

it('should accept multiple correct solutions', () => {
    const sol0 = new BinaryTree(100);
    sol0.left = new BinaryTree(56);
    sol0.left.left = new BinaryTree(34);
    sol0.left.right = new BinaryTree(22);
    sol0.left.left.left = new BinaryTree(18);
    sol0.left.left.right = new BinaryTree(16);
    sol0.left.left.left.left = new BinaryTree(9);
    sol0.left.left.left.right = new BinaryTree(9);

    sol0.right = new BinaryTree(44);
    sol0.right.left = new BinaryTree(22);
    sol0.right.right = new BinaryTree(22);
    sol0.right.left.left = new BinaryTree(11);
    sol0.right.left.right = new BinaryTree(11);

    expectCorrect(checkHuffman(sol0, dicts.several_solutions));

    const sol1 = new BinaryTree(100);
    sol1.left = new BinaryTree(56);
    sol1.left.left = new BinaryTree(34);
```

```
        sol1.left.left.left = new BinaryTree(18);
        sol1.left.left.right = new BinaryTree(16);
        sol1.left.left.left.left = new BinaryTree(9);
        sol1.left.left.left.right = new BinaryTree(9);
        sol1.left.right = new BinaryTree(22);
        sol1.left.right.left = new BinaryTree(11);
        sol1.left.right.right = new BinaryTree(11);
        sol1.right = new BinaryTree(44);
        sol1.right.left = new BinaryTree(22);
        sol1.right.right = new BinaryTree(22);

        expectCorrect(checkHuffman(sol1, dicts.several_solutions));

        const wrong0 = new BinaryTree(100);
        wrong0.left = new BinaryTree(40);
        wrong0.left.left = new BinaryTree(22);
        wrong0.left.left.left = new BinaryTree(11);
        wrong0.left.left.right = new BinaryTree(11);
        wrong0.left.right = new BinaryTree(18);
        wrong0.left.right.left = new BinaryTree(9);
        wrong0.left.right.right = new BinaryTree(9);
        wrong0.right = new BinaryTree(60);
        wrong0.right.left = new BinaryTree(38);
        wrong0.right.left.left = new BinaryTree(22);
        wrong0.right.left.right = new BinaryTree(16);
        wrong0.right.right = new BinaryTree(22);

        expectWrong(checkHuffman(wrong0, dicts.several_solutions));
    });

    it('should detect invalid trees', () => {
        const unused = new LetterFrequencyDistribution({ 'A': 100 });

        const invalid0 = new BinaryTree(100);
        invalid0.left = new BinaryTree(40); // wrong sum of children
        invalid0.left.left = new BinaryTree(20);
        invalid0.left.right = new BinaryTree(21);
        invalid0.right = new BinaryTree(60);
        invalid0.right.left = new BinaryTree(20);
        invalid0.right.right = new BinaryTree(40);

        spyOn(console, 'error');

        expectError(checkHuffman(invalid0, unused));
```

```
        expect(console.error).toHaveBeenCalled();

        const invalid1 = new BinaryTree(99); // not 100
        invalid1.left = new BinaryTree(87);
        invalid1.right = new BinaryTree(12);

        expectError(checkHuffman(invalid1, unused));

        const invalid2 = new BinaryTree(100);
        invalid2.left = new BinaryTree(50); // only has one child
        invalid2.left.left = new BinaryTree(50);
        invalid2.right = new BinaryTree(50);

        expectError(checkHuffman(invalid2, unused));
    });

    it('should generate correct trees', () => {
        const generateAndTestHuffman = (freq: LetterFrequencyDistribution) => {
            const huffmanTree = getHuffmanTree(freq);
            const frequencyTree = getAsFrequencyTree(huffmanTree, freq);
            expectCorrect(checkHuffman(frequencyTree, freq));
        };

        generateAndTestHuffman(dicts.three);
        generateAndTestHuffman(dicts.four);
        generateAndTestHuffman(dicts.five);
        generateAndTestHuffman(dicts.several_solutions);
    });

    it('should generate only optimal (correct) trees', () => {
        const bitsPerLetters = (f: LetterFrequencyDistribution, exp: number) => {
            const huffmanTree = getHuffmanTree(f);
            expect(freq.getAverageBitsPerLetters(huffmanTree)).toBe(exp);
        };

        bitsPerLetters(dicts.three, 1.48);
        bitsPerLetters(dicts.four, 1.50);
        bitsPerLetters(dicts.five, 2.23);
        bitsPerLetters(dicts.several_solutions, 2.74);
    });
});
```

Appendix F

First User Study Survey

Fragebogen
"Daten komprimieren"

Vielen Dank, dass Sie sich Zeit nehmen, um meine Masterarbeit zu testen. In meiner Arbeit geht es darum, zusätzliche interaktive Aufgaben als Ergänzung zum Thema Komprimierung auf gymnasialer Stufe zu erstellen. Die Übungen sollen als Ergänzung zum Buch dienen und die Theorie interaktiv und spielerisch vertiefen. Um die Aufgaben lösen zu können, muss man wissen, wie Shannon-Fano und Huffman Bäume aufgebaut werden. Ausserdem muss man das Konzept der arithmetischen Kodierung sowie die binäre Darstellung von Rationalen Zahlen kennen.

Die Übungen sollten selbsterklärend sein. Falls etwas unklar ist oder nicht funktioniert, bin ich froh um eine kurze Rückmeldung. Sie finden nachfolgend eine Liste mit Ideen, was es zu testen gibt. Sie können aber gerne auch wild drauflos experimentieren. Auf den nächsten Seiten finden Sie dann ein paar Fragen zur Evaluation. Sie können die Antworten an tobiasae@student.ethz.ch schicken. Dazu können Sie beispielsweise die Antworten in dieses editierbare PDF schreiben oder direkt im Mail jeweils die Nummer der Frage mit angeben. Vielen Dank!

1. Öffnen Sie die Applikation unter https://n.ethz.ch/~tobiasae/gymnasium/de/
2. Navigieren Sie zur "Shannon-Fano zeichnen" Aufgabe des Kapitels "Komprimierung".
3. Versuchen Sie die Aufgabe zu lösen.
4. Testen Sie nun die restlichen Aufgaben dieses Kapitels. Sie brauchen die Aufgaben nicht vollständig zu lösen. Oben rechts finden Sie Hinweise zum aktuellen Fortschritt einer Aufgabe (ein Ausrufezeichen, wenn die Aufgabe noch falsch bzw nicht gelöst ist und ein Gutzeichen, falls sie korrekt gelöst ist). Klicken Sie während dem Lösen der Aufgaben sporadisch darauf und lesen Sie das Feedback.
5. Wechseln Sie zu einer Aufgabe, die Sie schon bearbeitet haben und überprüfen Sie, ob die von Ihnen vorher gemachten Fortschritte immer noch vorhanden sind.
6. Exportieren Sie Ihren Fortschritt und speichern Sie ihn als Datei ab (drei Punkte oben rechts → exportieren).
7. Laden Sie die Seite neu[1] und importieren Sie die vorher gespeicherte Datei wieder. Überprüfen Sie kurz, ob die Aufgaben wieder soweit gelöst sind wie vorher.

[1] Möglicherweise wird die Seite beim neu laden nicht mehr gefunden. In diesem Fall können Sie wieder via https://n.ethz.ch/~tobiasae/gymnasium/de/ zur Seite gelangen.

1. Hat das Öffnen der Webseite funktioniert?

2. Gab es Unklarheiten in der Aufgabenstellung? Wenn ja, welche und bei welchen Aufgaben?

3. Bitte geben Sie an, wie gut Ihnen die einzelnen Aufgabentypen gefallen haben.

   (a) Shannon-Fano zeichnen
   nicht gut mittelmässig gut sehr gut

   (b) Huffman zeichnen
   nicht gut mittelmässig gut sehr gut

   (c) Reverse Shannon-Fano
   nicht gut mittelmässig gut sehr gut

   (d) Reverse Huffman 1
   nicht gut mittelmässig gut sehr gut

   (e) Reverse Huffman 2
   nicht gut mittelmässig gut sehr gut

   (f) Binär nach Dezimal
   nicht gut mittelmässig gut sehr gut

   (g) Dezimal nach Binär
   nicht gut mittelmässig gut sehr gut

   (h) Kürzeste Binärzahl
   nicht gut mittelmässig gut sehr gut

   (i) Arithmetische Kodierung
   nicht gut mittelmässig gut sehr gut

4. Gab es technische Schwierigkeiten bei einer Aufgabe? Wenn ja, welche und bei welchen Aufgaben?

5. War das Feedback zu Ihren Lösungen jeweils verständlich und hilfreich? Wenn nein, was könnte verbessert werden?

6. Haben Sie weitere Bemerkungen?

Appendix G

Second User Study Survey

Fragebogen
“Daten komprimieren”

Vielen Dank, dass Sie sich Zeit nehmen, um meine Masterarbeit im zweiten Testgang zu testen. Im folgenden finden Sie nochmals eine kurze Beschreibung der Arbeit.

In meiner Arbeit geht es darum, zusätzliche interaktive Aufgaben als Ergänzung zum Thema Komprimierung auf gymnasialer Stufe zu erstellen. Die Übungen sollen als Ergänzung zum Buch dienen und die Theorie interaktiv und spielerisch vertiefen. Um die Aufgaben lösen zu können, muss man wissen, wie Shannon-Fano und Huffman Bäume aufgebaut werden. Ausserdem muss man das Konzept der arithmetischen Kodierung sowie die binäre Darstellung von rationalen Zahlen kennen.

Die Übungen sollten selbsterklärend sein. Falls etwas unklar ist oder nicht funktioniert, bin ich froh um eine kurze Rückmeldung. Auf den nächsten Seiten finden Sie dann ein paar Fragen zur Evaluation. Sie können die Antworten an tobiasae@student.ethz.ch schicken. Dazu können Sie beispielsweise die Antworten in dieses editierbare PDF schreiben oder direkt im Mail jeweils die Nummer der Frage mit angeben. Vielen Dank!

1. Öffnen Sie die Applikation unter https://n.ethz.ch/~tobiasae/gymnasium/de/
2. Versuchen Sie ein paar Aufgaben zu lösen.
3. Gehen Sie zur Kategorie “Zufall” und lassen Sie ein paar mal eine zufällige Aufgabe generieren.

1. Hat das Öffnen der Webseite funktioniert?

2. Gab es Unklarheiten in der Aufgabenstellung? Wenn ja, welche und bei welchen Aufgaben?

3. Bitte geben Sie an, wie gut Ihnen die einzelnen Aufgabentypen gefallen haben.

   (a) Shannon-Fano zeichnen
   nicht gut mittelmässig gut sehr gut

   (b) Max-Balancierung zeichnen
   nicht gut mittelmässig gut sehr gut

   (c) Huffman zeichnen
   nicht gut mittelmässig gut sehr gut

   (d) Reverse Max-Balancierung
   nicht gut mittelmässig gut sehr gut

   (e) Reverse Huffman 1
   nicht gut mittelmässig gut sehr gut

   (f) Reverse Huffman 2
   nicht gut mittelmässig gut sehr gut

   (g) Binär nach Dezimal
   nicht gut mittelmässig gut sehr gut

   (h) Dezimal nach Binär
   nicht gut mittelmässig gut sehr gut

   (i) Kürzeste Binärzahl
   nicht gut mittelmässig gut sehr gut

   (j) Arithmetische Kodierung
   nicht gut mittelmässig gut sehr gut

4. Gab es technische Schwierigkeiten bei einer Aufgabe? Wenn ja, welche und bei welchen Aufgaben?

5. War das Feedback zu Ihren Lösungen jeweils verständlich und hilfreich? Wenn nein, was könnte verbessert werden?

6. Hat die Zufallsfunktion vernünftige zufällige Aufgaben generiert?

7. Haben Sie weitere Bemerkungen?

Bibliography

[1] ABZ. einfachinformatik. https://web.archive.org/web/20210825120432/https://einfachinformatik.inf.ethz.ch/. Accessed: 2021-08-25.

[2] ABZ. Webtigerjython. https://web.archive.org/web/20210825140320/https://webtigerjython.ethz.ch/. Accessed: 2021-08-25.

[3] ABZ. Xlogoonline. https://web.archive.org/web/20210825135816/https://xlogo.inf.ethz.ch/release/latest/. Accessed: 2021-08-25.

[4] Firouz Anaraki. Developing an effective and efficient elearning platform. *International Journal of the computer, the internet and management*, 12(2):57–63, 2004.

[5] Benjamin S Bloom et al. Taxonomy of educational objectives. vol. 1: Cognitive domain. *New York: McKay*, 20(24):1, 1956.

[6] Sonja Tabea Blum. A platform independent, computer-based learning environment to a textbook for computer science. Master's thesis, ETH Zurich, Department of Computer Science, 2018.

[7] George EP Box. A note on the generation of random normal deviates. *Ann. Math. Statist.*, 29:610–611, 1958.

[8] Inc. Brilliant Worldwide. Brilliant. https://web.archive.org/web/20210916093522/https://brilliant.org/. Accessed: 2021-09-16.

[9] Michelene TH Chi and Ruth Wylie. The icap framework: Linking cognitive engagement to active learning outcomes. *Educational psychologist*, 49(4):219–243, 2014.

[10] Douglas Crockford. Json. https://web.archive.org/web/20210902212546/https://www.json.org/json-en.html. Accessed: 2021-09-03.

[11] Robert M Fano. *The transmission of information*. Massachusetts Institute of Technology, Research Laboratory of Electronics . . . , 1949.

[12] Google. Angular. https://web.archive.org/web/20210817140115if_/https://angular.io/. Accessed: 2021-08-16.

[13] Google. Blockly. https://web.archive.org/web/20210825160506/https://developers.google.com/blockly/. Accessed: 2021-08-25.

[14] Juraj Hromkovič and Björn Steffen. Why teaching informatics in schools is as important as teaching mathematics and natural sciences. In *International Conference on Informatics in Schools: Situation, Evolution, and Perspectives*, pages 21–30. Springer, 2011.

[15] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[16] Facebook Inc. Jsx. https://web.archive.org/web/20210902155942/https://facebook.github.io/jsx/. Accessed: 2021-09-02.

[17] Facebook Inc. React. https://web.archive.org/save/https://reactjs.org/. Accessed: 2021-08-16.

[18] Donald E Knuth. *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional, 2014.

[19] Pivotal Labs. Jasmine. https://web.archive.org/web/20210902213408/https://github.com/jasmine/jasmine. Accessed: 2021-09-07.

[20] MIT. Scratch. https://web.archive.org/web/20210825154758/https://scratch.mit.edu/. Accessed: 2021-08-25.

[21] University of Washington. Practice-it. https://web.archive.org/web/20210421205338/https://practiceit.cs.washington.edu/. Accessed: 2021-08-25.

[22] Stuart Reges and Marty Stepp. *Building Java Programs*. Pearson, 2014.

[23] Raimond Reichert and Werner Hartmann. On the learning in e-learning. In *EdMedia+ Innovate Learning*, pages 1590–1595. Association for the Advancement of Computing in Education (AACE), 2004.

[24] Ricarose Vallarta Roque. *OpenBlocks: an extendable framework for graphical block programming systems*. PhD thesis, Massachusetts Institute of Technology, 2007.

[25] Rolf Schulmeister. Taxonomy of multimedia component interactivity. a contribution to the current metadata debate. *Studies in Communication Sciences. Studi di scienze della communicazione*, 3(1):61–80, 2003.

[26] Claude Elwood Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[27] Gary Stix. Encoding the "neatness" of ones and zeroes. https://web.archive.org/web/20210706123652/https://www.huffmancoding.com/my-uncle/scientific-american. Accessed: 2021-07-06.

[28] Nikolai Tillmann, Michal Moskal, Jonathan De Halleux, and Manuel Fahndrich. Touchdevelop: Programming cloud-connected mobile devices via touchscreen. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, pages 49–60, 2011.

[29] Tynker. https://web.archive.org/web/20210815050416/https://www.tynker.com/. Accessed: 2021-08-25.

[30] David Weintrop. Block-based programming in computer science education. *Communications of the ACM*, 62(8):22–25, 2019.

[31] David Weintrop, David C Shepherd, Patrick Francis, and Diana Franklin. Blockly goes to work: Block-based programming for industrial robots. In *2017 IEEE Blocks and Beyond Workshop (B&B)*, pages 29–36. IEEE, 2017.

[32] Evan You. Vue.js. https://web.archive.org/save/https://vuejs.org/. Accessed: 2021-08-16.

[33] Friedel Ziegelmayer. Karma. https://web.archive.org/web/20210908094529/https://karma-runner.github.io/6.3/index.html. Accessed: 2021-09-08.