Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A computer-based learning environment aimed for students at the 3rd and 4th grade level

Bachelor's Thesis

Florian Bütler
flbuetle@ethz.ch

Chair of Information Technology and Education
ETH Zürich

Supervisor:
Prof. Juraj Hromkovič

February 18, 2021

Acknowledgements

prof hrom kovic

team of abz for feedback

proofreaders

Todo:

Abstract

a brief introduction describing the discipline that the paper belongs to

a clear and concise statement of your problem

a brief explanation of your solution and its key ideas

a brief description of the results obtained and their impacts

Todo:

Contents

Chapter 1

Introduction

1.1 Motivation and Background

With the introduction of Lehrplan 21 Computer Science became an integral part of the Swiss education curriculum [**?**]. Pupils learn to understand the basic concepts of Computer Science and how to use them for problem solving. These concepts include methods on how to process, evaluate and summarize data, how to securely communicate and how to develop solution strategies for simple problems of information processing [**?**]. The Education and Counselling Center for Computer Science Education at ETH Zurich (ABZ) supports schools to teach these concepts among others by providing teaching resources and learning environments.

Todo: mention text book where the exercises are taken from

1.2 Goals

The main goal in this bachelor thesis is to implement tasks and riddles based on the textbook "einfach Informatik 3/4" in a computer-based learning environment that teaches the following concepts:

- representing information with symbols,
- keeping information secret and
- learning from data

for pupils in the second cycle. Along with solving tasks and riddles about the mentioned topics the ability of reading, writing, counting and calculating is trained as well.

1.3 Related Work

Todo: abz.inf.ethz.ch other learning environments

1.4 Outline

This report first explains how the aforementioned concepts are thought by hands-on exercises, then gives in-depth technical insight on how a learning environment is developed and how these exercises are implemented. Finally, the report ends with a conclusion with a review of the project.

Todo: update

Todo: disclaimer that shown implementation is not complete and sometimes simplified for the sake of readability

Chapter 2

Representing Information with Symbols

2.1 Similar Words

Representing information with symbols is a fundamental concept of Computer Science as information should be represented clear and concisely. Words can be seen as a sequence of symbols, namely a sequence of letters.

2.1.1 Exercises

Transmitting information includes representing it in a message, sending it to the destination and the receiver being able to make sense of it even tough the message might contain errors such as a spelling mistake. To achieve this sender and receiver agree to only send messages with a minimal editing distance [**?**] between each of them.

Editing distance

The editing distance is the amount of operations that need to be done to transform a message in another. Operations are deleting, inserting and changing a letter . A cost function exists that defines the cost of each operation, and in our case each operation has a cost of 1 (unit cost model). The editing distance is then the minimal cost to transform a message into another one by a sequence of operations. In our case a message is a word.

Todo: add some TI explanation

Example 2.1. The editing distance between `LIKE` and `BIKE` is 1, since changing the first letter from an `L` to and `B` transforms the first word into the second.

If sender and receiver agree to only transmit words with a minimal editing distance of e.g 3, then the receiver can still uniquely determine what word the sender has sent even when at most 1 spelling mistake has been made. The receiver

calculates the minimal editing distance between the received word and each of the agreed words and chooses the word with the least editing distance. The receiver assume that this was the word the sender wanted to sent. This of course work only if there are not more than 1 error in the word. If this happens the editing distance to another word is closer than to the original word and hence the word is misinterpreted. To counter this, sender and receiver might agree on a bigger minimal distance, but this comes with a tradeoff. When chosing a bigger minimal distance for a fixed alphabet, the number of words that can be used shrinks. To maintain the same amount of words the size of the lphabet can be increased too, resulting in longer words.

The purpose of the `Similar Words` exercises is to learn these operations. Therefore an exercise is dedicated to each opertion: adding, changing and removing a letter from a word. Additionally, an exercise about swapping adjacent letters in a word is included. Swapping adjacent letters is not part of the mentioned operations and usually consists of 2 operations: removing and adding or changing twice. However, when typing on a keyboard typing mistakes happen often and most of the time only two adjacent letters are swapped. This exercises is supposed to train the ability to spot these sort of mistakes.

Adding a letter

In the `adding a letter` exercise pupils are presented a word, the alphabet from A to Z and spaces between each letter of the word as well as at the beginnig and the end of the word, where a letter can be added. Pupils are supposed to choose a letter from the alphabet and add it to one of the mentioned spaces to form a new valid word.

Example 2.2. The word `PACE` is given. By adding the letter `S` before the first letter, the valid word `SPACE` is formed.

Changing a letter

In this exercise there is again a word and the alphabet from A to Z shown. Again pupils should select a letter from the alphabet, but this time, instead of adding it to a space, the selected letter should be replace a letter from the word itself to create a new valid word.

Example 2.3. The word `BIKE` is given. By changing the first letter from a `B` to a `L` the new valid word `LIKE` is formed.

Removing a letter

In this exercise pupils receive a word and they should select a character from within the word an move it to the trashcan to remove the letter from the word

itself to form a new valid word.

Example 2.4. The word `SPACE` is given. By removing the first letter the new valid word `PACE` is formed.

Swapping adjacent letters

To learn to recognize typing mistakes in a word pupils are presented a word with swapped adjacent letters and they are supposed to identify those and swap them back to restore to original word. Here are multiple difficulty levels possible, where on the easy level only one pair of adjacent letters is swapped and on harder levels multiple pairs of adjacent letters are swapped.

Example 2.5. The word `BKIE` is given. By swapping the second and third letter the original word `BIKE` is restored.

2.1.2 Implementation

Word Generation

The foundation of these exercises is to have a list of words and for each word a list of similar words. A similar word is a word to which the original word can be transformed to by either adding, changing or removing a letter. Generating a list of words is the easy part. Basically any list of words will do the jobs, but they should be understable for pupils. So the first step is to collect a list of nouns for children provided by a online service [].

Not much more difficult, but much more expensive is the computation of similar word. Given the children nouns list and the allowed words list, one can brute force a list of similar word by simply applying each transformation to every children noun and checking whether the transformed word exists in the allowed word lists. The script to generate a list of similar words for each word and each operation is given in listing **??**. To have multiple difficulty levels in the exercise `Changing Letters`, similar word with an editing distance of two are generated as well. All other operations only include words with an editing distance of one.

```
1  for word in children_words:
2    add = list()
3    for pos in range(len(word) + 1):
4      for letter in ALPHABET:
5        w = word[:pos] + letter + word[pos:]
6        if contains(allowed_words, w) and not w in add:
7          add.append(w.upper())
8    similar_words["add"][1][word.upper()] = add
9
10   remove = list()
11   for pos in range(len(word)):
```

```
12      w = word[:pos] + word[pos + 1 :]
13      if contains(allowed_words, w) and not w in remove:
14          remove.append(w.upper())
15  similar_words["remove"][1][word.upper()] = remove
16
17  change = list()
18  for pos in range(len(word)):
19    for letter in ALPHABET:
20      w = word[:pos] + letter + word[pos + 1 :]
21      if w != word and contains(allowed_words, w) and not w in
            change:
22        change.append(w.upper())
23  similar_words["change"][1][word.upper()] = change
24
25  change = list()
26  for left_pos in range(len(word)):
27    for right_pos in range(left_pos + 1, len(word)):
28      for left_letter in ALPHABET:
29        for right_letter in ALPHABET:
30          if word[left_pos] == left_letter or word[right_pos] ==
                right_letter:
31            continue
32          w = (
33            word[:left_pos]
34            + left_letter
35            + word[left_pos + 1 : right_pos]
36            + right_letter
37            + word[right_pos + 1 :]
38          )
39          if w != word and contains(allowed_words, w) and not w in
                change:
40            change.append(w.upper())
41  if len(change) >= MIN_SIMILAR_WORDS:
42    change.sort()
43    similar_words["change"][2][word.upper()] = change
```

Listing 2.1: Algorithm to generate a list of similar words

There is no need to generate a list of similar word for the `Swapping letters` exercise, since the letters can be swapped during the exercise creation.

Todo: maybe check for editing distances

Significantly more difficult is to generate a list of allowed words. Three different approaches were taken:

- Using a list of approximately ten thousand nouns
- Collecting a list from online dictionary of allowed words of the well known word game Scrabble [**?**]
- Generating a list from a spell checker

The first approach resulted in having a word list with which exercises could

be generated. However, the word list obviously did not cover all possible word pupils could find. Therefore the word list needed to be extended.

Hence the idea to collect Scrabble word came up. This seemed like a good idea until it showed that the online Scrabble dictionary did also not contain all valid words. Collecting words from the Scrabble dictionary resulted in approximately thirty thousand word between two and 8 charachters long.

Finally, the last approach was taken by generating a list of allowed word from a spell checker. However, spell checkers do not have a list of all allowed word, but rather of two files: the dictionary file and the affix file. The dictionary files contains a list of words and applicable rules for each word and the affix file contains a list explaining these rules. Each rule defines a prefix or suffix that can be applied to a word [?]. Luckily there exists a tool called `unmunch` in the hunspell ecosystem that generates a list of word from the dictionary file and the affix file as easy as `unmunch German`$_{de_CH}$.dic German$_{de_CH}$.aff > wordlist.txt[?].Intheendalistofaboutonemillionallowedwordswasgenerated(withoutanywordlength

Todo: example of dictionary and affix file

2.2 Number Systems

Todo: fix structure

2.2.1 Exercises

2.2.2 Representing Numbers like the Maya

Today, one is used to the decimal number system (base number 10), with 10 symbols. The Maya used a different number system with with a base number of 20 i.e the vigesimal number system. It is believed that the Maya used their ten fingers and ten toes to count adding up to twenty. But instead of having a symbol for each number as in the decimal system, the Maya used three different symbols: zero (a turtle shell, belly side up), one (a dot) and five (a bar) [?]. The mayan representation of the decimal number from 0 to 19 can be found in figure ??.

The following exercises are supposed to familiarize pupils with a new number system, the mayan number system. First converting numbers between the decimal number system and the mayan numbers is trained topped of an exercise about adding two mayan numbers.

Representing mayan numbers

A decimal number is presented to the pupils and they need to choose the correct amount of dots and bars representing the decimal number. For

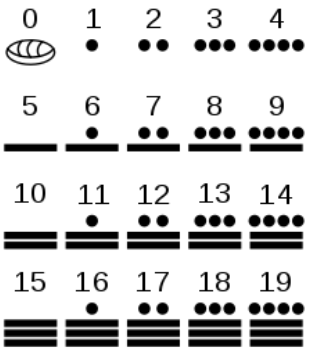


Figure 2.1: Representation of the Maya numbers from 0 to 19

the purpose of simplicity, only decimal numbers between 1 and 19 are chosen.

Example 2.6. The decimal number 17 is writte as three bars and two dots.

Understanding mayan numbers

In this exercise the pupils learn to understand mayan numbers. Again only decimal numbers between 1 and 19 are used and the pupils need to understand what mayan number is shown and write down the decimal equivalent.

Adding mayan numbers

To sum this section up, adding mayan numbers is learned. This exercises combines the previous two, since the pupils need to understand the summands given in the mayan number system, add them and write the sum down in either decimal number system or the mayan number system.

2.2.3 Representing Numbers with Coins

The following exercises introduce a new number system and trains an already learned one: the binary number system and the decimal number system. Both number systems are praticed with coins with numbers. The binary coins include the following numbers: 1, 2, 4, 8, 16, 32 and 64 (Fig. ??). The decimal coins include 1, 2, 5, 10, 20 and 50 (Fig. ??). Overall, the same concepts are praticed for both number systems with the limitation that every binary coin can at most be used once. For both number systems the following exercises are given:

- Conversion of a decimal number to its coin representation

Figure 2.2: Representation of the decimal coins

Figure 2.3: Representation of the binary coins

- Conversion of a number given in its coin representation to a decimal number

Additionally, for the decimal number system the following exercise is given:

- Reducing the number of decimal coins in a given set of decimal coins

Conversion of a decimal number to its coin representation

This exercise includes two difficulty levels:

- Converting a decimal number to a coin representation, where the sum of all coins are equal to the decimal number
- Converting a decimal number to a coin representation, where the sum of all coins are equal to the decimal number and the amount of used coins is minimal.

Conversion of a number given in its coin representation to a decimal number

This exercise is straigh forward. A number given in its coin representation, either binary or decimal coins, have to be converted to a decimal number.

Reducing the amount of decimal coins in a given set of decimal coins

Similar to the previous exercise, a number in its coin representation is given and pupils have to display the same number but with less coins.

2.2.4 Implementation

Chapter 3

Keeping Information Secret

The previous chapter ?? was about representing information with symbols. This section is about keeping information secret. Ciphers haven been used for thousands of years [?]. They are used to keep information secret from people, that are not supposed to have knowledge of it. Not encrypted information is called clear text. Once one encrypted a clear text, it is called a cipher text and only people who know how to decrypt the cipher text can read originally encrypted information. The exercises in this section are introducing pupils to the concepts of ciphers.

3.1 Cipher Texts from Reversed Letters

3.1.1 Exercises

The cipher used in these exercises is a simple mix up of letters and both directions are trained: encryption and decryption. In the decryption exercise, the pattern, on which the clear text was encrypted with, is shown. The pupils need to understand the pattern and move the letters in the cipher text accordingly to retrieve the clear text. The encryption exercise is set up analogously. Multiple difficulty levels are possible by changing the amount of moved letters.

Example 3.1. The cipher text is `TNOAM` and the pattern is shown in figure **??**. By moving the letter in the cipher text according to the pattern, the clear text can be retrieved: `MONAT`.

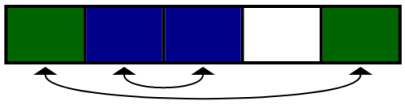


Figure 3.1: Pattern

3.1.2 Implementation

Both exercises have a similar implementation. Both need a drawn pattern that indicates the cipher. Fundamentally, the only difference is that for the decryption exercise the given text consists of reversed letters and the solution is compared to the original word, while the encryption exercise gives first the original word as a text and the solution is compared to match the pattern. The pattern is created by generating an array of tuples. Each tuple indicates two swapped letters. These two letters are selected randomly and have to be distinct. The exact algorithm used is given in Listing ??. Both encryption and decryption exercises have two difficulty levels. They differ by the amount of tuples:

- easy - one tuple i.e two swapped letters
- medium - two tuples, i.e in total four swapped letters if the the word has at least four letters, otherwise only two swapped letters

```
1  createPattern(text: string[], swapAmount: number): Array<[number,
       number]> {
2    const pattern = new Array<[number, number]>();
3    const letters = [...Array(text.length).keys()];
4    for (let j = 1; j <= swapAmount && 2 * j <= text.length; j++) {
5      const leftIndex = Math.floor(Math.random() * letters.length);
6      const left = letters[leftIndex];
7      letters.splice(leftIndex, 1);
8      const rightIndex = Math.floor(Math.random() * letters.length);
9      const right = letters[rightIndex];
10     letters.splice(rightIndex, 1);
11     pattern.push([Math.min(left, right), Math.max(left, right)]);
12   }
13   return pattern;
14 }
```

Listing 3.1: Algorithm to generate an array of distinc tuples of given size

The array of tuples needs to be graphically represented. For this purpose the <canvas> was introduced in HTML5. With the help of scripting language like JavaScript or TypeScript, one can drawn diagrams, edit images and create animations on canvas elements. A canvas element has a height and width and is basically a coordinate system with having the origin in the top left corner at (0, 0). Usually 1 unit corresponds to 1 pixel and all elements are drawn relative to the origin. Some basic operations on a canvas element are drawing rectangles, paths, straight lines, arcs, curves and moving the pen without drawing. Additionally, it can be choosen between filling the drawn structure with a color or only highlighting its border [?]. These operations already meet the requirements to draw a pattern and requires four steps:

Todo: maybe explain canvas in more detail

- drawing the grid - each cell represents a letter of the word
- colorizing the pairs - the pairs representing the two letters that need to be swapped are colorize in the same color
- drawing the arrow heads - below each cell, that is part of a pair, an arrow head is drawn
- drawing the arrow lines - connecting the pairs with lines without the lines overlapping or unnecessarily crossing each other

The implementation for these steps is given in listing ??.

Everything drawn outside of the canvas border is invisible, hence special attenention needs to be given to the first few lines in ?? (L 2-5). When drawing lines, the line width has to be taken into account as well. Therefore, the rectangle is drawn with a distance of half a line width to the actual border of the canvas, so the whole line width is drawn instead of only half of it.

Drawing the grid includes drawing a rectangle first and then the lines. To draw the lines, the pen needs to be moved to the start of the line, set down, moved to the end of the line and finally be drawn [??].

Drawing a arrow head includes more operations. The pen is first moved the the pointy end of the arrow, set down, moved to the two other corners and back to the pointy end. This time, instead of only highlighting just the border, the whole structure is filled with the black color [??].

```
1  drawPattern(cells: number, pairs: [number, number][]) {
2    const rectX = this.lineWidth / 2;
3    const rectY = this.lineWidth / 2;
4    const rectWidth = this.width - this.lineWidth;
5    const cellHeight = this.height / 2 - this.lineWidth;
6    const cellWidth = rectWidth / cells;
7
8    this.drawGrid(rectX, rectY, rectWidth, cellHeight, cells,
         cellWidth);
9
10   // sort to have it easier to draw the lines connecting the boxes
         on the correct height
11   pairs.sort(([a, b], [c, d]) => Math.abs(a - b) - Math.abs(c - d))
         ;
12   const arrowLevelY = this.calculateArrowLevelY(pairs);
13   for (let i = 0; i < pairs.length; i++) {
14     for (let j = 0; j < 2; j++) {
15       const pairIndex = pairs[i][j];
16
17       this.ctx.fillStyle = this.colors[i % this.colors.length];
18       this.fillBox(rectX, rectY, cellHeight, cellWidth, pairIndex);
19
```

```
20        const centerX = rectX + cellWidth * pairIndex + cellWidth /
              2;
21        this.ctx.fillStyle = "black";
22        this.drawArrowHead(
23          centerX,
24          rectY + cellHeight + 5,
25          Math.min(30, cellWidth),
26          10
27        );
28      }
29
30      this.drawArrowLine(
31        cellHeight,
32        cellWidth,
33        rectY,
34        rectX,
35        arrowLevelY.get(JSON.stringify(pairs[i])),
36        cells,
37        pairs[i]
38      );
39    }
40  }
```

Listing 3.2: Implementation to draw the pattern on a canvas element

```
drawGrid(
  rectX: number,
  rectY: number,
  rectWidth: number,
  cellHeight: number,
  cells: number,
  cellWidth: number
) {
  // draw box
  this.ctx.strokeRect(rectX, rectY, rectWidth, cellHeight);
  // draw walls
  for (let i = 1; i < cells; i++) {
    this.ctx.beginPath();
    this.ctx.moveTo(rectX + cellWidth * i, rectY);
    this.ctx.lineTo(rectX + cellWidth * i, rectY + cellHeight);
    this.ctx.stroke();
  }
}
```

```
drawArrowHead(
  headX: number,
  headY: number,
  headWidth: number,
  headHeight: number
) {
  this.ctx.beginPath();
  this.ctx.moveTo(headX, headY);
  this.ctx.lineTo(headX - headWidth / 2, headY + headHeight);
  this.ctx.lineTo(headX + headWidth / 2, headY + headHeight);
```

Figure 3.2: Cipher text of an encrypted number

	·	:	⋮	⁘
□	0	1	2	3
○	4	5	6	7
△	8	9		

Figure 3.3: Symbol table to encrypt numbers

```
11    this.ctx.lineTo(headX, headY);
12    this.ctx.fill();
13  }
```

Example 3.2. The pattern in figure **??** is generated by calling `drawPattern(5, [[0, 4], [1, 2]])` with `5` as `cells` and `[[0, 4], [1, 2]]` as `pairs`.

3.2 Cipher Texts from New Characters

3.2.1 Exercises

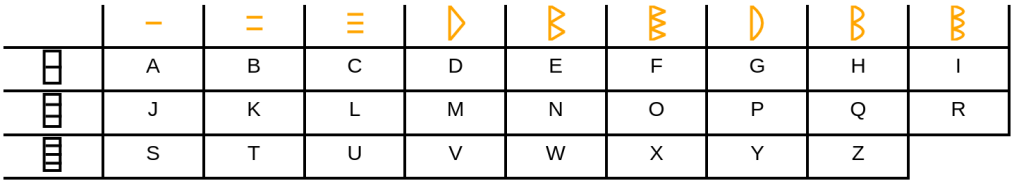
Sometimes, only moving symbols in not enough to keep information secret. A better why is to substitute symbols with new symbols. These symbols may be letters, numbers or completly new symbols, that are solely invented for the purpose of encrypting information. In the following exercises the last approach is followed. Again both direction, encryption and decryption, are trained. But this time, instead of having a pattern, there is a symbol table showing how the letters are encrypted.

Example 3.3. The cipher text is shown in figure **??** and the symbol table in figure **??**. By using the symbol table one can decrypt the cipher text to `52`.

3.2.2 Implementation

The implementations of the encryption and decryption exercises using symbols are similar to the previous exercises using patterns. The main difference is he use of a symbol table showing what alphanumerical letter is encrypted with what symbol. A symbol is composed of two shapes and

Figure 3.4: Cipher text of an encrypted text



	A	B	C	D	E	F	G	H	I
	J	K	L	M	N	O	P	Q	R
	S	T	U	V	W	X	Y	Z	

Figure 3.5: Symbol table to encrypt and decrypt letters

the composition configuration is shown in the symbol table. In the decryption exercise a with symbols encrypted text is shown. Pupils have to use the symbol table to translate symbols to its alphanumerical counter part, decrypting the text. In the encryption exercise, it is the other way around. Pupils have to encrypt a text with the help of the symbol table. The symbol table is implemented in the SymbolTable.vue component. It accepts a two-dimensional array representing the table content and generates this table with the help of canvas elements. Each cell in the table is a canvas element. Each shape consist of multipe elements e.g the yellow shape in the symbols for numbers consists of multiple points. If there are three or less points, the points are drawn in a vertical line and for four points, they are drawn in grid style. Other shapes are more complex, like the yellow shapes in the symbols for letters consisting of multiple arcs. The implementation for drawing these shapes takes as an argument the amount of arcs that need to be drawn and first draws the straight vertical lines and then the specified amount of arcs. This way this implementation can be used for all three arc shapes [??] used in symbol table [??]. All other shapes are implemented in a similar way, so there is as less code as possible.

```
1  drawArcs(arcs: number) {
2    this.ctx.strokeStyle = "#ffa500";
3    if (arcs <= 0) {
4      return;
5    }
6    this.ctx.lineJoin = "bevel";
7    this.ctx.beginPath();
8    this.ctx.moveTo(this.lineWidth / 2, this.height - this.lineWidth
       / 2);
9    this.ctx.lineTo(this.lineWidth / 2, this.lineWidth / 2);
10   const diffY = (this.height - this.lineWidth) / (2 * arcs);
11   for (let i = 1; i <= arcs; i++) {
12     this.ctx.quadraticCurveTo(
```

```
13        this.width - this.lineWidth / 2,
14        (2 * i - 1) * diffY,
15        this.lineWidth / 2,
16        2 * i * diffY
17      );
18    }
19    this.ctx.stroke();
20  }
```

Listing 3.3: Implementation of drawing a variable amount of arc symbols

Chapter 4

Learning from Data

The following exercises are logic-based, combinatorial puzzles. The idea is that pupils need to conclude the solution by a partially completed grid.

4.1 Row of Trees

4.1.1 Exercises

The row of trees is a one dimensional grid i.e a row either of size 3 or 4. In every row there is exactly one tree of every height between 1 and 3 (Fig. ??) or 4 (Fig. ??) respectively. At both ends of the row, the amount of visible trees from this end of the row is given. In the following the amount of visible trees from one end is referred to as view. Pupils are given an empty row with only the number of trees seen from both ends of the row and need to place a tree of each height such the aforementioned rules are met.

Example 4.1. For a row of size 3, if given the value 2 for both ends of the row, then one possible solution to the puzzle is shown in figure **??**.

Figure 4.1: Trees from height 1 to 3

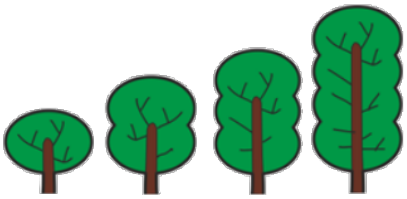


Figure 4.2: Trees from height 1 to 4

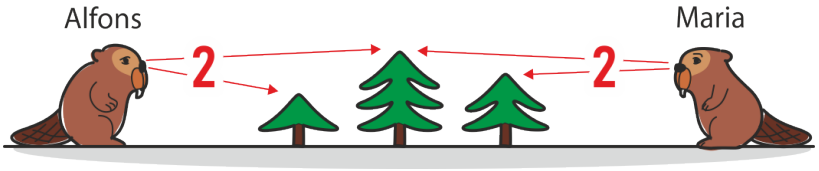


Figure 4.3: Visualization of the puzzle solution from the tree row example

4.1.2 Implementation

The Row.vue component is implemented to work with any row size. However, since only images of trees for a row size of three and four are present, it makes only sense to use it for those row sizes. The internal representation of a tree is its height e.g a tree of height 2 is represented as 2 internally. All possible exercise instance are given by all permuations of a monotone increasing list of values of the given size starting with 1 e.g $1, 2, 3$ for size 3. Hence the approach taken here is to generate an array with these values, shuffle the array and calculate the the views from both end of the row. Listing ?? shows this approach and returns the view from the left end and the view from the right end.

```
1 generate(): [number, number] {
2   const values = Array.from({ length: this.size }, (_, i) => i + 1)
        ;
3   this.shuffle(values);
4   return [
5     this.getVisibleTrees(values),
6     this.getVisibleTrees(values.slice().reverse()),
7   ];
8 }
```

Listing 4.1: Algorithm to generate a row of trees exercise instance of `this.size`

Interesting functions in ?? are shuffle and getVisibleTrees. Apprently, there exists no built-in function in TypeScript, that shuffles an array. Therefore, a own function needs to be implemented. This is not as easy as it seems since, naive approaches may result in different permutations having different probabilities to appear. An implementation that shuffles an array, where all permuatations are approximately equal is given in

Listing ?? [?]. Another crucial part of the implementation of this exercise is the calculation of the amount of trees one sees from one end of the row of trees. An algorithm for calculating this is given in Listing ??. To calculate the view from the other side, the same function can be used and only the input needs to be reversed as shown in Listing ??.

```
shuffle(arr: number[]): void {
  for (let i = arr.length; i >= 0; i--) {
    const randomIndex = Math.floor(Math.random() * i);

    const temp = arr[i];
    arr[i] = arr[randomIndex];
    arr[randomIndex] = temp;
  }
}
```

Listing 4.2: Algorithm to shuffle an array

```
getVisibleTrees(values: number[]): number {
  let min = 0;
  let visible = 0;
  for (let i = 0; i < values.length; i++) {
    if (values[i] > min) {
      visible++;
      min = values[i];
    }
  }
  return visible;
}
```

Listing 4.3: Algorithm to calculate the amount of visible tree from one end

You may have spotted the use of the custom type row. This custom type is the internal respresentation of the row and its definition is given in Listing ??. A row is an array of the custom type rowField and the field value saves what tree was placed on this field.

```
type rowField = {
  id: number;
  value: number;
};
type row = rowField[];
```

Listing 4.4: Definition of the custom row and rowField type

To determine wheter a given solution by a pupils is correct or incorrect, the amount of visible trees in the row of placed trees from both end is again calculated and compared to the indicated amount. If both are the same, then the solutions is evaluated as correct [??].

```
isCorrect(): boolean {
    const row = this.values.map((field) => field.value);
    const visibleLeft = this.getVisibleTrees(row);
```

Figure 4.4: Row of tree example exercise

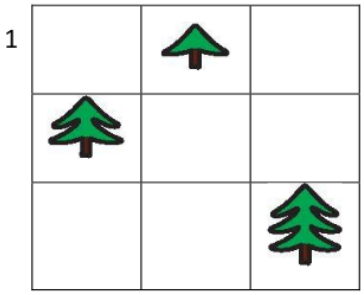


Figure 4.5: Example of an unsolved 3x3 tree sudoku

```
4       const visibleRight = this.getVisibleTrees(row.slice().reverse()
            );
5       return !(visibleLeft !== this.leftView || visibleRight !== this
            .rightView);
6   }
```

Listing 4.5: Correctness check of a row of tree exercise solution

4.2 Tree Sudoku

4.2.1 Exercises

Tree sudoku is similar to the well known traditional sudoku with the difference that trees of different heights are placed instead of numbers, and for both ends of every row and collumn the amount of visible trees is given. Again, the amount of visible trees from on end of a row or colums is referred to as a view. Otherwise, the puzzle follows the same rules as the row of trees and is either of size 3x3 or 4x4.

Example 4.2. An example of a 3x3 tree sudoku is shown in figure **??**

4.2.2 Implementation

Again the Sudoku.vue component is implement in a way such that it works with different sizes. In general, a similar approach for the tree sudoku exercise is taken like for the row of trees exercise ??. The internal representation is of a tree of height e.g 2 is 2 and some custom types are used: sudoku and sudokuField [??]. The major difference between these two exercises is that the tree sudoku is two-dimensional. Therefore a two-dimensional array is used for the internal representation of the tree sudoku grid. A new field used in the sudokuField comapred to the rowField is locked. This field indicates that this sudoku field is part of the initial generation and when the same tree sudoku should be restarted, only not locked fields are erased.

```
1 type sudokuField = {
2     id: number;
3     value: number;
4     locked: boolean;
5 };
6
7 type sudoku = sudokuField[][];
```

Listing 4.6: Definition of the custom sudoku and sudokuField type

The generation of an instance of the tree suduoku exercise is a bit more complicated, since the additional constraints of a sudoku have to be considered. The approach taken here is to first generate an empty tree sudoku. Then in each round, a random tree is placed at a random position or a random value of visible trees is placed at a random end of a row or column until a solvable configuration is found.

The most basic operation on a tree sudoku is to decide whether it is valid or invalid i.e whether in each row and column every tree height does appear exactly once and whether the views match the placed trees. To check the validity the tree heights placed in each row and each column are collected. If some height appear more than once in a row or colums it is evaluated as invalid. Additionally, since the same algorithm to check the validity of an instance during the generation and when a pupils hands its solution in should be used, one has to differentiate the handling of empty field in those scenarios. During the generation of an instance, the instance should not be evaluated as invalid, when there are still empty field. However, it has to be evaluated as invalid upon checking a solution from a pupil. For each row and column the amount of visible trees is calculated for both ends and if these values match the views the instance is evaluated as valid. The algorithm to check the validity can be found in Listing ??.

```
1 isValid(values: sudoku, views: number[][], complete: boolean):
      boolean {
2   for (let i = 0; i < values.length; i++) {
```

```
3       const rowSeen = new Set<number>();
4       const colSeen = new Set<number>();
5       for (let j = 0; j < values[i].length; j++) {
6         const traversal: [number, Set<number>][] = [
7           [values[i][j].value, rowSeen],
8           [values[j][i].value, colSeen],
9         ];
10        for (let k = 0; k < traversal.length; k++) {
11          const [el, seen] = traversal[k];
12          if (seen.has(el)) {
13            return false;
14          } else if (el === 0) {
15            if (complete) {
16              return false;
17            }
18            continue;
19          } else {
20            seen.add(el);
21          }
22        }
23      }
24
25      if (rowSeen.size === this.size) {
26        const row = values[i].map((el) => el.value);
27        const visible = this.getVisibleTrees(row);
28        const visibleRev = this.getVisibleTrees(row.slice().reverse()
              );
29        if (
30          (views[1][i] !== 0 && views[1][i] !== visible) ||
31          (views[2][i] !== 0 && views[2][i] !== visibleRev)
32        ) {
33          return false;
34        }
35      }
36      if (colSeen.size === this.size) {
37        const col: number[] = [];
38        for (let k = 0; k < values.length; k++) {
39          col[k] = values[k][i].value;
40        }
41        const visible = this.getVisibleTrees(col);
42        const visibleRev = this.getVisibleTrees(col.slice().reverse()
              );
43        if (
44          (views[0][i] !== 0 && views[0][i] !== visible) ||
45          (views[3][i] !== 0 && views[3][i] !== visibleRev)
46        ) {
47          return false;
48        }
49      }
50    }
51    return true;
52  }
```

Listing 4.7: Validation algorithm for a tree sudoku instance

A tree sudoku generator needs a tree sudoku solver to check whether an instance is actually solvable. Solving a tree sudoku means that one needs to place trees in fields such that the tree sudoku is still valid. First an empty field has to be found, then tree heights are placed in the empty field until a tree height is found such that the tree sudoku is still valid. This procedure is repeated until the tree sudoku is completed. This would already conclude a tree sudoku solver, but in the end this solver is going to be used to generate a tree sudoku. Therefore one needs to know how many solutions are available for a given tree sudoku solution. Hence the solving algorithm shown in ?? counts how many solutions exist and returns it.

```
1  solve(values: sudoku, views: number[][]): number {
2    const [complete, emptyValueSlotRow, emptyValueSlotCol] = this.
       findEmptySlot(
3      values.map((row) => row.map((col) => col.value))
4    );
5    if (complete) {
6      this.valuesSolution = JSON.parse(JSON.stringify(values)) as
         sudoku; // deep copy
7      return 1;
8    }
9
10   let solutions = 0;
11   for (let i = 1; i <= this.size; i++) {
12     values[emptyValueSlotRow][emptyValueSlotCol].value = i;
13     if (this.isValid(values, views, false)) {
14       solutions += this.solve(values, views);
15       if (solutions > 1) {
16         break;
17       }
18     }
19   }
20   values[emptyValueSlotRow][emptyValueSlotCol].value = 0;
21   return solutions;
22 }
```

Listing 4.8: Solving algorithm for a tree sudoku instance

Generating a tree sudoku is now straigh forward. First a random empty field and a random empty view needs to be chosen, then with a probabilty of 0.5 either the empty field or the empty view is chosen and a random number is put in it. After that, the validity of the tree sudoku is checked and if it is valid, the amount of possible solutions for the tree sudoku is calculated. If no solution is found, a different number has to be tried. If more than one solutions are found, we start all over again by chosing a random empty field or view. If the tree sudoku has exactly one solution, the algorithm stops. An implementation of the here mentioned algorithm can be found in Listing ??.

```
generate(values: sudoku, views: number[][]): [sudoku, number[][]] {
  const [
    valuesComplete,
    emptyValueSlotRow,
    emptyValueSlotCol,
  ] = this.findEmptySlot(values.map((row) => row.map((col) => col.
      value)));
  if (valuesComplete) {
    return [values, views];
  }
  const [
    viewsComplete,
    emptyViewSlotRow,
    emptyViewSlotCol,
  ] = this.findEmptySlot(views);
  if (viewsComplete) {
    return [values, views];
  }

  const numbers: number[] = [];
  for (let i = 0; i < this.size; i++) {
    numbers[i] = i + 1;
  }
  this.shuffle(numbers);
  for (let i = 0; i < numbers.length; i++) {
    if (
      this.randomNumber(1) <= 0.5 ||
      this.LevelsWithNoViews.includes(this.currentDifficultyLevel)
    ) {
      values[emptyValueSlotRow][emptyValueSlotCol].value = numbers[
          i];
      values[emptyValueSlotRow][emptyValueSlotCol].locked = true;
    } else {
      views[emptyViewSlotRow][emptyViewSlotCol] = numbers[i];
    }
    if (this.isValid(values, views, false)) {
      const solutions = this.solve(values, views);
      if (solutions === 0) {
        values[emptyValueSlotRow][emptyValueSlotCol].value = 0;
        values[emptyValueSlotRow][emptyValueSlotCol].locked = false
            ;
        views[emptyViewSlotRow][emptyViewSlotCol] = 0;
        continue;
      } else if (solutions === 1) {
        return [values, views];
      } else {
        return this.generate(values, views);
      }
    } else {
      values[emptyValueSlotRow][emptyValueSlotCol].value = 0;
      values[emptyValueSlotRow][emptyValueSlotCol].locked = false;
      views[emptyViewSlotRow][emptyViewSlotCol] = 0;
    }
  }
```

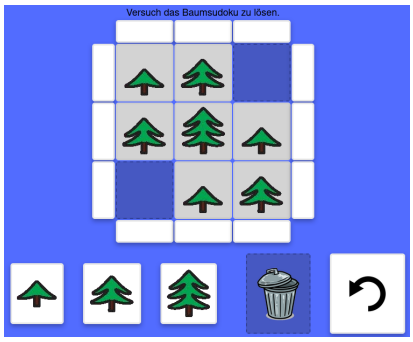


Figure 4.6: Tree sudoku example exercise

```
52   return [values, views];
53 }
```

Listing 4.9: Generation algorithm for a tree sudoku instance

Additionally, the tree sudoku exercise has three different difficulty levels. The difference between the levels is the amount of initially given information for an tree sudoku instance:

- easy - no views are given and at least 75% of the trees are placed
- medium - some views are given and at least 50% of the trees are placed
- hard - some views are given and only the minimal amount of information to solve the tree sudoku instance is given

Todo: explain coverage algorithm

To check whether a solution to tree sudoku exerise handed in by a pupils, the algorithm from Listing ?? can be reused without allowing empty fields.

Chapter 5

Testing and Continous Integration

Implemented funtionality should always be tested. Not only to ensure that is currently does what it is supposed to do, but also to avoid regression. Nothing is worse than introducing a bug by implementing a new feature with noticing it. Therefore in this project we use unit, snapshot and End-to-End test to avoid regression. Moreover, all tests are always run upon each commit to the GitLab repository.

5.1 Testing

For unit and snapshot testing Jest is used. Jest is a JavaScript testing framwork focusing on simplicity and is maintained by Facebook. It is supposed to require zero config and runs tests isolated and therefore can be paralellized [?]. For End-to-End testing Nightwatch is used. Nighwatch is a Node.js powered End-to-End testing framework for web applications and websites. It supports testing in Chrome, Firefox and Edge, supports page object to easily abstract the content of a page and allows extending the framwork with custom commands [?].

5.1.1 Unit Testing

Unit tests are usually used to verify the functionality of functions and work with the following concept: the tester has an input to a function and knows what the function should output for this input. The tester then compares the actual ouput of the function and compares it to the expected output. If those are not the same the test is evaluated as failed, and succeeded otherwise. There are mainly two different categories of unit tests: black-box and white-box testing.

Back-box and White-box Testing

Black-box testing is a testing method, where the tester does not know how a feature is implemented and has to think of cases by just using ones understanding of the feature. White-box testing on the other hand is a testing method, where the tester does know the implementation of the feature. Usually both kind of software testing is used. But since there are no software tester (testers without any knowledge of the impelementation) involved in this project, only white-box testing is applied [?].

Test Plan

General purpose components are tested in its specialized functionality i.e wheter they react correctly on different user inputs. The compon representing an exercises are tested for:

Todo: maybe elaborate on that?

- Initial conditions hold
- Various exercise specific user inputs are handled correctly
- Restarting the exercise works as expected
- Starting next example restores the initial conditions
- Correct answer is accepted
- Incorrect answer is rejected

Code Coverage

As mentioned before, unit test usually test a feature on the function level. To catch as many possible code paths and corner cases, the same function is tested multiple times with different inputs. A metric to judge the usefullness of a test suite is the code coverage. The code coverage shows how well a test suite covers the functionality. Usually, a coverage report includes:

- Statement coverage - The percentage of statements that have been executed
- Branch coverage - The percentage of branches of the constrol structures (e.g. if and loop statements) that have been executed
- Function coverage - The percentage of functions that have been called

- Lines coverage - The percentage of line of the source code that have been tested

High coverage percentage does not imply that it is a good test suite. Some critical paths can still be untested. It is generally accecpted that a code coverage of 80% is a desirable goal. Going above 80% of code coverage is usually costly and does not provide any valuable benefit [?].

The code coverage for the whole project and each component is given in table ??. Most of the components are well tested with a code coverage of above 80% in most categories. Some percentages indicate bad code coverage

Todo: elaborate on components with bad code coverage

Non-determinism

A problem that has to be tackled for testing is how to handle determinism. Some exercises are based on random behaviour to achieve a rich user experience. However, if random behaviour is used, then the actual output may not be the same as the expected output and the test is evaluated as failed. To circumvent this issue, one can mock the function that is used to generate a random numbers. Mock functions allow to overwrite the actual implementation of the function by intercepting calls to this function and return values configured by the test suite [?].

Todo: maybe refer to game mixin

5.1.2 Snapshot Testing

Snapshot tests ensures correctness of the user interface and that it does not change unexpectedly. More precicely, snapshot tests renders the template of a component to HTML, takes a snapshot and compares the snapshot with the previous saved reference snapshot. These two snapshots are compared and if they do not match, the test fails or succeds otherwise. If the changes made to the source code were intended to change the UI, the snapshot has to be updated. This requires an initial snapshot that is known to be correct. Hence snapshot should be commited to the repository as well [?].

Every component in this project is snapshot tested and is therefore save from unintended UI changes.

5.1.3 End-to-End Testing

End-to-End tests are used to test the entire application flow through an application. The main goal is to test it from a users perspective

Table 5.1: Test coverage of all components

File	% Stmts	% Branch	% Funcs	% Lines
All files	86.71	73.11	86.73	86.56
App.vue	100	100	100	100
Footer.vue	100	100	100	100
Header.vue	100	100	100	100
Home.vue	100	100	100	100
GameMixins.vue	55.56	100	27.27	55.56
Buttonmenu.vue	66.67	100	0	66.67
Difficulty.vue	87.5	62.5	100	87.5
ItemSelection.vue	100	100	100	100
Modal.vue	100	100	100	100
Trashcan.vue	100	100	100	100
Tutorial.vue	100	100	100	100
Undo.vue	100	100	100	100
PatternDecryption.vue	78.79	50	80	77.42
PatternEncryption.vue	79.41	50	81.82	78.13
From.vue	100	100	100	100
ItemDropzone.vue	100	100	100	100
ItemGroup.vue	100	100	100	100
NumbersystemsMixin.vue	75.41	62.5	84.62	77.59
Swap.vue	88.89	83.33	88.89	91.18
To.vue	90.24	45.45	90	92.11
Row.vue	89.04	65.71	100	88.89
Sudoku.vue	91.12	76.09	94.12	90.26
TreesMixin.vue	100	100	100	100
Add.vue	98.33	100	95.24	98.28
Change.vue	91.43	83.33	93.33	91.18
Remove.vue	100	100	100	100
Swap.vue	61.22	87.5	76	59.78

by simulating a user scenario [?].

Todo: maybe elaborate a bit more on that

Nighwatch allows to run End-to-End tests for Chrome, Firefox and Edge to ensure the application works across browser-borders. The End-to-End tests for this project make use of page objects. A page object is an abstraction of a page represented as an object. The goal of a page object is to simplify the End-to-End test by showing the same one can see when one visites the page [?].

The End-to-End tests in this project only covers the visibilty of all parts of an exersise one needs to see to solve it. The reason behind this is that many exercises are based on non-determinism and since a user cannot influence that the End-to-End tests are not able to do that as well.

5.2 Continous Integration

This project make use of the GitLab Continous Integration (CI). When pushing a commit to the projects GitLab repository, the CI pipeline is run. The CI pipeline is specified in the .gitlab-ci.yml file in the projects root folder and specifies what jobs need to be done. Usually, these jobs build, test and validate changes made to the source code and allow to easily catch bugs and errors. The CI pipline of this project consists of two parts:

- Unit and Snapshot tests - runs the unit and snapshot tests ?? ??
- End-to-End tests - runs the End-to-End tests in Firefox ??

Chapter 6

Conclusion, Drawbacks and Further Work

6.1 Conclusion

6.2 Drawbacks

- Not all exercises per topic
- Own learning environment

6.3 Further Work

- Inegrate into existing learning environment
- Add additional exercises

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

Verfasst von (in Druckschrift):
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**

Ich bestätige mit meiner Unterschrift:
- Ich habe keine im Merkblatt „Zitier-Knigge“ beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.

Appendix A

First Chapter Title

```
Lorem ipsum dolor sit amet
```