My Success Squared - Developing and Integrating a Smart Generator and Corrector of Maths exercises

Auteur : Wéry, Victor
Promoteur(s) : Donnet, Benoît
Faculté : Faculté des Sciences appliquées
Diplôme : Master : ingénieur civil en informatique, à finalité spécialisée en "computer systems security"
Année académique : 2023-2024
URI/URL : http://hdl.handle.net/2268.2/21156

Master thesis

Civil Engineering in Computer Science

My Success Squared

Developing and Integrating a Smart Generator and Corrector of Maths Exercises

Author: Victor Wéry

Promoter: Prof. Benoît Donnet

Liege University:

Academic Year 2023-2024

Abstract

My Success Squared is an application to support Mathematics learning and practice in a playful way for secondary school students. However, in its current state, exercises statements and the corresponding solutions have to be created and added by hand.
The first objective of this project is thus to provide a smart generator capable of creating exercises of desired types (equations, fraction simplification, etc) and with desired properties (complexity, parameters, etc).
As it is an application for learning students, it would be desirable to be able to deliver consistent feedback depending on the answer those students will provide and the second objective is thus to provide a smart corrector that will be able to find what mistake has been made based on the answer and/or step-by-step reasoning provided by a student and to provide corresponding feedback.

Acknowledgements

I want to thank Pr. Donnet and Ms. Brieven for their insights, help and patience. Their guidance has been essential to this project's completion.

Table of contents

Chapter 1

Introduction

Learning mathematics is often not very exciting, especially in the first years of secondary school when it mainly consists in drilling the same exercises on simple concepts such as simple equations, fraction calculations, factorization, etc for those to be assimilated.
As a result, it is not always easy for teachers to keep these phases interesting for their students.

For that purpose, My Success Squared was created.
It allows to perform those basic concepts learning and drilling in a "gamified" way. The different chapters are associated to islands that have to be explored, containing tutorials to learn the concepts and exercises to practice on them. It includes a competitive mode as well in which students can compete against each other and try to become the presidents of the islands described above.

However, in its current state, the application is unable to generate any exercise and those have to be added by hand by the teachers wanting to create a lesson.
Moreover, for the application to be a good tool that can be used in the context of learning, it is essential that the students using the application and solving exercises on it are given consistent feedback when they fail. That is, not just knowing if their answer is right or wrong but what mistake they made if their answer is wrong.

That is the context of this thesis and the two big topics that we will have to address and solve are the automated generation of exercises of various types (exercise generator) and the generation of consistent feedback for any exercise and that is adapted to each specific student's case (feedback generator). The chosen solution should be as general as possible as to allow to easily extend them and bring new exercise types.

More specifically, we will have a brief overview of the different ways in which similar problems have already been addressed.
Then we will have a quick look at the tools that were used to carry out this project.
We will then get in the heart of the matter with a detailed explanation on how both exercise and feedback generators have been implemented for both first degree equation and fraction simplification exercises.
Finally, we will have a guide through the actual files that were written for this project and how to use them.

Chapter 2

State of the art

We are not the first ones to address the problem of automated exercise and feedback generation. Numerous works have already been done on the subject though not that much on the specific topic of mathematics exercises. The topic of programming exercises for example is much more often addressed. Valentin Baum did a complete state of the art analysis on this subject in his work "Implementing an Automatic Pointers Exercises Generator in CAFÉ 2.0"(2023)[3] and although it is not precisely our subject, it still brings quite pertinent elements that can help us to find a solution to our problem.

2.1 Exercise generation

2.1.1 Template

Template generation is a generation method that uses generic templates defining the global structure of exercises. The generator then replaces the parameters with random values. This is explored in "Math exercise generation and smart assessment"(Almeida et al., 2013)[1] with the system Passarola which provides a simple and powerful language to define templates ("Exercise generation with the system Passarola"(Almeida et al., 2013)[2]).

2.1.2 CLP

CLP (Constraint Logic Programming) is a grammar based scheme where exercises formats are defined with context free grammars and constraints are then used during generation to prefer some choices over others. This allows to have a general solution while keeping the possibility to tune the exercise properties thanks to the constraints. This method is studied in "A CLP-based tool for computer aided generation and solving of maths exercises"(Tomás et al., 2002)[5] and used to generate math quizzes in "Automatic generation and delivery of multiple-choice math quizzes"(Tomás et al., 2013)[6] with the help of AGILMAT which is web application aiming at facilitating math education ("AGILMAT–a Web Application for Math Education", Tomás et al., [7]).

2.1.3 Others

In the work of Valentin Baum [3] we can learn about other methods that could be used for mathematics exercise generation:

- **Mutation** : Consists in starting from an existing exercise and create new ones by slightly modifying it.
- **Construction from solution** : Consists in generating the solution first and then build the exercise on top of that solution.

2.2 Feedback generation

Not much has been made on the topic of generating feedback for mathematics exercises.
In "Math exercise generation and smart assessment"(Almeida et al., 2013)[1] (template based generation), they use the fact that a solution to an exercise could be represented by a sequence of operators applied to data vectors. If the operators are reversible, starting form an erroneous answer, one could revert to the state in which the error has been made and thus find the appropriate feedback.
Others like in "Feedback design patterns for math online learning systems"(Inventado et al., 2017)[4] implement a feedback generation scheme based on history of students errors to define typical errors and associated feedback.

Again, Valentin Baum[3] brings additional information:

- **Path construction** : Consists in creating a graph that links the exercise statement to the possible solutions through nodes representing intermediate states and edges representing the transition between those nodes.
- **Transformation model** : Consists defining the answer to the exercise and some transitions with associated feedback that link wrong answers to the right one. This allows to find the correct feedback to provide given the student's answer by performing all the transitions on it and matching with the correct answer.

2.3 Chosen methods

2.3.1 Exercise generation method

In the context of this thesis, we will generate first degree equation and fraction simplification exercises which are two very different types of exercises. We thus chose to use different methods for each type:

- **First degree equation** : We use the template based model. The exact scheme is detailed in section 4.2.
- **Fraction simplification** : For that exercise type, we chose the construction from solution model. The reasoning that yielded this conclusion is detailed in section 4.3.3.

2.3.2 Feedback generation method

As we need to be able to provide feedback to students from the beginning, the errors history model is therefore inadequate.
So is the reversible operators scheme as it only allows to generate feedback for a restricted set of exercise types.

We then consider two possible models:

- **Transformation model** : A bit different from the one described earlier, it consists in applying transformations to the exercise statement that each represents a possible error made somewhere in the reasoning.
- **Path construction**

The transformation model has the advantage of not having to represent even correct reasoning steps. However path construction model allows for more specific feedback which is a key feature when generating feedback for young students who may not know the equivalence between the error $A + B = C \rightarrow A = B + C$ and the error $A + B = C \rightarrow A - B = C$.

The chosen feedback generation scheme is thus path construction as it is the model that allows for the greatest precision in feedback.

Chapter 3

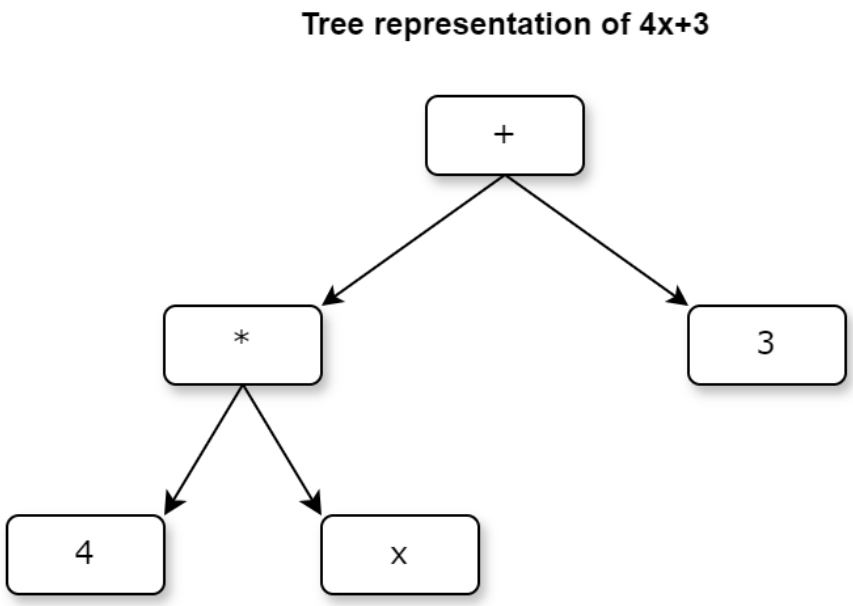
Introduction to SymPy

Sympy is a Pyhon library that is very useful to perform symbolic calculations.
It allows to define and maintain mathematical expressions ($expression = a + b$) and to easily perform arithmetic operations on them ($expression + 3 = a + b + 3$).
It defines the following operators:

- The multiplication operator "$*$"/"Mul" ($\frac{a}{b} = a * b^{-1}$).
- The addition operator "$+$"/"Add" ($a - b = a + (-b)$).
- The power operator "$**$"/"Pow".

Many very useful functions are defined to solve equations, simplify or factorize an expression, etc.

Sympy expressions are built in tree structures



which will be very useful both for exercise and feedback generations.
A Sympy expression has two attributes:

- *func* which is the operator at the root of the expression tree.
- *args* which is the list of the arguments on which the operator is applied and that are Sympy expressions as well. These correspond to the sub-trees whose roots are the children of the tree root.

$$\begin{aligned}&expression = 4 * x + 3\\&expression.func = Add\\&expression.args = (4 * x, 3)\end{aligned}$$

For expressions that do not contain an operator, their *func* attribute is Number or Symbol etc and their *args* is an empty list.

Sympy spares us the tasks of defining structures for representing mathematical operations, parsers, and algorithms for solving, simplifying, etc, and allows us to easily perform many operations on mathematical expressions.

Chapter 4

Exercise generation

We want to offer the teachers a way to order bunches of exercises for their lessons without having to define the exercise statements or answers themselves.
However, they need to keep control on the exercise properties such as complexity, difficulty level, etc.
For that purpose, the exercise generator takes as input a json file containing a set of parameters which values are set by the teachers.

4.1 General

In this section, we will discuss the general concepts and mechanisms that may apply to any type of exercise.

4.1.1 Parameters

Let's consider the fact that any exercise statement of any type is composed of mathematical expressions. These expressions are composed of operators (+, *, ...) applied to other sub-expressions or tokens. These tokens can be numbers or parameters (a, b, c, x, y, ...). Teachers may want to have some control over these tokens:

- They may or may not want parameters to get involved in their exercises. They may indeed want to order fraction simplification exercises for beginners and keep it simple with only numbers for example:

$$\text{Simplify: } \frac{-60}{-360}$$

  or introduce parameters for more advanced exercises:

$$\text{Simplify: } \frac{4b^2}{-4ab^2+2b^3}$$

  They may as well want to vary the number of parameters involved in the exercises.

  For that purpose, we define the *nb_param_range* parameter composed of two numbers (*[lower_bound, higher_bound]*) bounding the number of parameters that will appear in the exercise. If the teachers want to define the exact number of parameters to appear, they can simply use that number for both bounds, and use 0 if they don't want any parameter in their exercises.

- We also give the teachers the opportunity to chose which parameters may appear in their exercises with the *possible_params* parameter.

- They may want to bound the value of the numbers that are introduced during the exercise generation because it may lead to unnecessarily complex expressions and difficulty in terms of mental calculation. As an example, here are two generated fraction simplification exercises with 2 factors bounded in absolute value respectively by 19 and by 5:

$$\text{Unnecessarily complex: } \frac{-1404}{-24024} = \frac{-12\times11\times9}{-12\times11\times154} = \frac{9}{154}$$

$$\text{Acceptable: } \frac{200}{32} = \frac{4\times2\times25}{4\times2\times4} = \frac{25}{4}$$

4.1.2 Generation

The generation of the mathematical expressions and the enforcement of the constraints on the token that constitutes them are implemented by a general generator class from which each specialized generator will inherit.

First, this generator is able to generate complex mathematical expressions.
What we call a complex expression in an expression composed of an operator applied to one or more operands as opposed to tokens which are composed of a single parameter or number.

Complex expression: $a + b$ composed of tokens: a, b

A complexity value is associated to each expression and computed as follow:

$$complexity(expression) = \begin{cases} 0 & \text{if expression is token} \\ 1 + \text{complexity(expression.args)} & \text{if expression is complex} \end{cases}$$

where complexity(expression.args) is the sum of the complexities of expression's operands.
As for example, the complexity of $(a + b)c$ is:

$$\begin{aligned} complexity((a+b)c) &= 1 + complexity(a+b) + complexity(c) \\ &= 1 + (1 + complexity(a) + complexity(b)) + 0 \\ &= 1 + (1 + 0 + 0) + 0 \\ &= 2 \end{aligned}$$

Parameters selection

The number n of parameters to appear in the exercise to be generated is a random number bounded by the *nb_param_range* parameter. The n first parameters in *possible_params* then form the list of parameters to appear in the exercise.

Complex expression generation

From there we can define a complex expression generation algorithm:

Algorithm 1 Generate Complex Expression

```
procedure GenerateComplexExpression($complexity$)
    if $complexity == 0$ then
        return $DrawRandomToken()$
    else
        $complexity \leftarrow complexity - 1$
        $operator \leftarrow$ DrawRandomOperator()

        $first\_arg\_complexity \leftarrow$ random($0, complexity$)
        $second\_arg\_complexity \leftarrow complexity - first\_arg\_complexity$

        $first\_arg \leftarrow$ GenerateComplexExpression($first\_arg\_complexity$)
        $second\_arg \leftarrow$ GenerateComplexExpression($second\_arg\_complexity$)

        return $operator(first\_arg, second\_arg)$
    end if
end procedure
```

In other words, given the complexity of the expression to generate, if that complexity is 0, then the expression is a randomly drawn token.
If it is greater than 0, then an operator is randomly drawn as well as the complexities of the operand. This is done in such a way that their sum is equal to the initial complexity-1. Operands are then generated with this same algorithm with their corresponding complexities. Finally the resulting expression is the chosen operator applied to the generated operands.

However this is a generalized view and some tuning is still required:

- Depending on the context, certain operators may or may not be desirable. For example we may not want the power operator to appear in low level exercises as to keep them simple. We thus need to specify the list of acceptable operators to the algorithm.

- For the same reason, we may not want every operators to appear equally often. We thus need to dynamically associate a probability of appearance to each operator and adapt the operator drawing algorithm accordingly.

- Not every operator should be treated in the same way. Except in very specific cases, the power operator for example should only have integers as power value and the algorithm should be adapted accordingly.

- In order to prevent high powers from appearing in exercises that we would want to keep simple, we should as well specify bounds to the power value of power operations.

Token generation

The token generation is more simple. A token can be either a parameter or a number which itself can be an integer or a fraction (bringing decimal point numbers into the exercises would be of no interest). The generation algorithm requires to specify:

- The list of parameters that can appear in the exercise.

- The probability for a token to be a fraction. As fraction operations can be difficult for younger students, it is important to be able to specify whether or not fractions can be included and how often.

- Whether 0 can be drawn as it cannot be the denominator of a fraction or would nullify any multiplication operation, etc.

- Whether 1 can be drawn as it is neutral as a factor or denominator, etc.

The algorithm works as follow:

Decide whether the token will be a fraction or not in accordance with the associated probability.
If it is, generate both numerator and denominator as non-fraction tokens with this same algorithm.
If not, choose, among the possible parameters or an integer between 0, 1 or 2 (depending on whether 0 and 1 are allowed) and the maximum value defined in the general parameters. The sign of the token is then randomly chosen.

Parameter number enforcement

One problem persists with the token generation algorithm described above. Indeed, although one could have asked for having at least n parameters in his exercises, there is a chance for an exercise with m<n parameters to be generated. This could be just because at least one parameter has never been chosen during token generation. We thus need to enforce that each parameter that has been asked is present at least once in the exercise.
We can do that by checking for missing parameters in a given exercise and for each one of them, adding it to a random sub-expression of the exercise.

4.2 First degree equation exercises

4.2.1 Exercise format

A first degree equation is typically of the form $ax + b = c$, with x the equation's unknown and a, b and c numbers:

$$\text{Solve: } 3x + 7 = -1$$

But we could complexify it into the form $a(bx + c) + d = e$:

$$\text{Solve: } 5(3x - 1) - 1 = 0$$

From now on, we will call the $a(bx + c) + d$ block a complex **factor** (not term).

Furthermore, we could increase the complexity by adding several unknown dependent terms in the equation:

$$\text{Solve: } 5 - (x - 3) = 4x - (3x - 8)$$

Finally, the reason why we call a $a(bx + c) + d$ block a complex factor instead of a complex term is that we could have several unknown dependent factors multiplying. Indeed, although that would mean having powers of x greater than 1, as long as these powers of x cancel out, we would still have a first degree equation:

$$\begin{gathered}(x + 2)(x + 1) = (x + 4)(x - 5) \\ x^2 + 3x + 2 = x^2 + x - 20 \\ 3x + 2 = x - 20\end{gathered}$$

The simpler forms $ax + b = c$ and $a(bx + c) + d = e$ are typically for beginner students who may have difficulties with fraction calculations. We thus want to ensure that the answers to such exercises are integers.

For the same reasons, it is important to be able to control the frequency of fraction token appearance.

4.2.2 Parameters

In addition to the general parameters, we define 6 specific parameters to tune the equations form:

- ***possible_unknowns*** defines the list of symbols that can be used as the equation's unknown.

- ***fraction_token_proba*** is the probability for each token other that the unknown to be a fraction token.

- ***complex_ factor_ proba*** is the probability for each unknown dependent factor to be a complex factor ($a(bx + c) + d$ instead of $ax + b$).

- ***add_ unknown_ terms_ range*** defines the lower and upper bounds to the number of additional unknown dependent terms (thus in addition to the one necessarily present by default).

$$\begin{aligned}[0, 0] :& 5 - (x - 3) = 9\\ [1, 1] :& 5 - (x - 3) = 4x - (3x - 8)\end{aligned}$$

- ***add_ unknown_ factors_ range*** defines the lower and upper bounds to the number of additional unknown dependent factors per term (thus in addition to the one necessarily present by default in each term)

$$\begin{aligned}[0, 0] :& 5 - (x - 3) = 3x - 8\\ [1, 1] :& (5 - (x - 3))(3x + 7) = (3x - 8)(-x - 2)\end{aligned}$$

- ***unknown_ division_ proba*** is the probability for an unknown dependent factor to be put at the denominator.

$$\frac{-2x}{x-6} = \frac{8x}{3-4x}$$

However, generating such forms of equations appeared to be quite complicated. As it was only a secondary requirement, we thus chose not to implement it and the parameter is left unused in the current generator version.

4.2.3 Generation

The generation process for first degree equations is as follow:

First of all, we obviously need to choose the parameters and equation's unknown accordingly with *possible_unknowns* and with the help of the selection process discussed in section 4.1.2.

Then we can proceed with the equation generation.

Equation generation

Starting with both sides of the equation equal to 0, randomly draw the number of unknown dependent terms to generate. This is done accordingly with *add_unknown_terms_range*. Then, generate these terms and add them to a randomly chosen side of the equation.

If one side of the equation is left zero, replace it with a random token generated accordingly with *fraction_token_proba* and allowing 0 and 1.

If there are some parameters in the equation, ensure that all chosen parameters do appear in the equation thanks to methods discussed in section 4.1.2.

If *nb_param_range*, *fraction_token_proba*, *add_unknown_terms_range* and *add_unknown_factors_range* are all zero, consider that the exercise is of the simplest form and should have an integer solution as discussed in section 4.2.1.
The solution that has been chosen is the following:

We can rewrite any first degree equation into their homogeneous form:

$$\begin{aligned} 2x - (x-2) + 3 &= x - 4 \\ 2x + x + 2 + 3 &= x - 4 \\ 3x + 5 &= x - 4 \\ 2x + 9 &= 0 \end{aligned}$$

Which is of the form $ax + b = 0$ and whose solution is $\frac{-b}{a}$. Thankfully, the SymPy library defines the Poly function that yields a polynomial if applied to the subtraction of the right hand side from the left hand side of the equation. Solving the homogeneous equation is then equivalent to equalizing this polynomial to 0.

Now b is not necessarily divisible by a so we need to change the equation as to have the solution $\frac{c}{a}$ with c divisible by a.
We can obtain that by defining $c = -b + adj$ where $adj = b \mod a$ is an adjustment and $b \mod a$ is the modulo of b by a (which is equal to the remainder of the integer division of b by a. Ex: $7 \mod 3 = 1$ because $3/7 = 2$ remaining 1). This ensures by definition of the modulo operator that c is divisible by a.
We thus simply need to add adj to the equation (even in its initial, non-homogeneous form).

Finally, to ensure the greatest diversity and randomness, we can add to adj $n \times a$ with n randomly chosen between 0 and *max_int_value* and with random sign. This is correct as $n \times a$ is of course divisible by a.
Furthermore, we can randomly chose whether to add it to the right hand side of the equation or subtract it from its left hand side.

Unknown dependent term generation

The unknown dependent term generation is as follow:

Choose the number of unknown dependent factor the term will be composed of accordingly with *add_unknown_factors_range*. Then generate these factors and define the new term as their product.

However, as we have seen, some terms are composed of more than 1 factor, higher powers of x will appear and we need them to cancel out as to keep a first degree equation.
We thus need to ensure that the last term is such that it cancels out each power of x greater than 1. The chosen solution to that problem is the following:

Start with the new term equal to 0.
Reduce the equation to its homogeneous form and for each term of degree >1, add it to the new term.

Then, to add randomness, complete the new term by generating an unknown dependent factor and adding it to the term.
Randomly choose whether to add it to the right hand side or subtract it from the left hand side of the equation.

Equation before adding new term:

$$(3x+5)(2x-4) = -3x(2x-5)$$
$$6x^2 - 12x + 10x - 20 = -6x^2 + 15x$$
$$12x^2 - 17x - 20 = 0$$

New term: $12x^2 - x - 2$

Equation after adding new term:

$$(3x+5)(2x-4) = -3x(2x-5) + 12x^2 - x - 2$$

Unknown dependent factor generation

The unknown dependent factor generation is as follow:

Choose whether to generate a simple factor $(ax+b)$ or a complex factor $(c(dx+e)+f)$ in accordance with *complex_factor_proba*.
Then replace the parameters values by random tokens generated accordingly with *fraction_token_proba* and allowing 1(except for c) and 0 (except for a, c, d, e).

4.2.4 Evaluation

We want to make sure that the implementation of the generator allows for a maximum diversity in the exercises. We don't want the students to be able to exploit patterns in the exercise generation and be able to guess the answers without properly solving the exercises.

A simple way to make sure that will not be the case is by checking the diversity in the exercise answers. The closer we are to the case where each exercise has an answer not shared by any other exercise, the better.

Before analyzing the results for the 2 most important parameters in first degree equation generation *fraction_token_proba* and *complex_factor_proba*, let's first clearly define the notions addressed in figures 4.1, 4.2, 4.3 and 4.4 and how they were made.

Graphs explanation

Graphs 4.1a and 4.2a allow to compare the occurrence rates of respectively fraction tokens and complex factors to the values of *fraction_token_proba* and *complex_factor_proba*. If the generator implementation is correct, both values should be approximately equal.

Graphs 4.1b and 4.2b display as a function of their respective parameter the measured diversity in the sampled exercises. That diversity is defined as the ratio between the number of different answers encountered and the number of exercises in the sample.

Figures 4.3 and 4.4 show for different values of their respective parameter the computed probabilities for an answer to be shared by several exercises. As an example, Percentage = 0.6 for Nb of exercises sharing the common answer = 3 means that 60% of encountered answers were shared by exactly 3 different exercises in the sample.
For better readability, only non-zero percentages are displayed.

Those graphs were generated by, for each considered value of the respective parameter, generating 20 batches of 50 exercises (thus samples of 1000 exercises).
For each batch b_i, the mean occurrence rate μ_i was computed, and the final average occurrence rate μ was computed over all the batches.
The error err was computed as $\frac{\sigma(\mu_1,...,\mu_n)}{\sqrt{n}}$ with $\sigma(\mu_1, ..., \mu_n)$ the standard deviation over μ_i and $n = 20$ (The errors are present on graphs 4.1a and 4.2a but are so tiny that they can barely be seen).

The exercises were generated with:

Parameter	Value
nb_param_range	[0, 0]
max_int_value	5
add_unknown_terms_range	[0, 2]
add_unknown_factors_range	[0, 0]

as those are parameter values that will most typically be chosen by teachers.

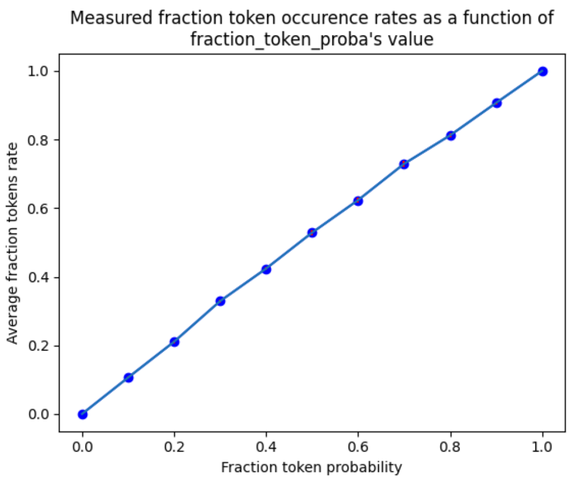
Graphs analysis

First thing we can observe on graphs 4.1a and 4.2a is that the real occurrence rates correspond to the values of the respective parameters as expected.

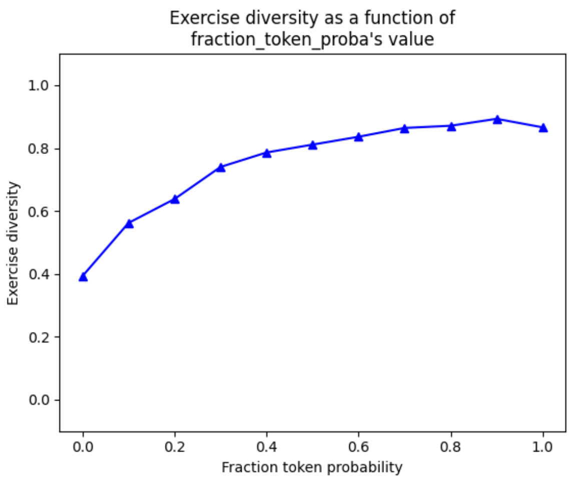
Second, graphs 4.1b and 4.2b show an increase in exercise diversity with the respective parameter.
It is expected for *fraction_token_proba* as a fraction token can take many more values that other tokens. It is thus logical that exercises with more fraction tokens generate a greater answer diversity.
Likewise, as complex factors include more tokens than simple factors, the increase in exercise diversity with the value of *complex_factor_proba* is also expected.

However the exercise diversities for low values of the parameters seem quite low, especially for *fraction_token_proba*.
A diversity of 0.4 as in graph 4.1 for *fraction_token_proba* = 0 could be problematic. Indeed, for 1000 exercises, this means that there are only 400 different answers.
If it is because some answers are shared by not too many exercises, this is acceptable. However, if it is because very few answers are shared by many exercises (for example, 1 answer shared by 601 exercises and 399 other answers shared by only one exercise) then this is problematic.

Thankfully, we can see on figures 4.3 and 4.4 that we are in the acceptable case. Indeed, for any value of the parameter, the percentage of answers that are shared by several exercises decrease really fast in a decreasing exponential fashion with the number of exercises sharing the answer.
As the worst case is a tiny percentage of the answers (probably corresponding to a single answer) shared by 43 exercises, that is, not even 1 exercise out of 20, it is safe to conclude that the generator satisfies the properties we expected.

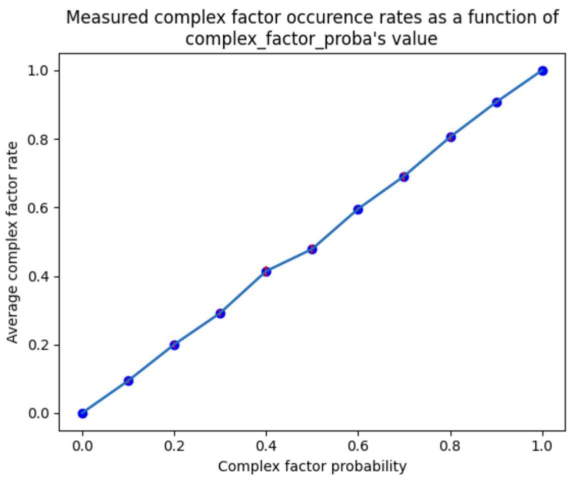


(a)

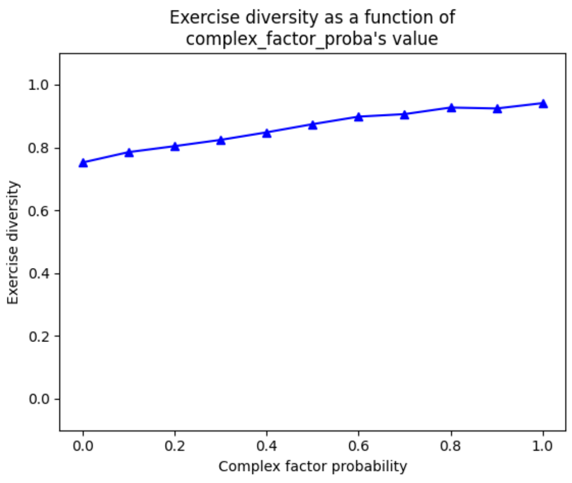


(b)

Figure 4.1: Generator evaluation regarding *fraction_token_proba* over samples of 1000 exercises

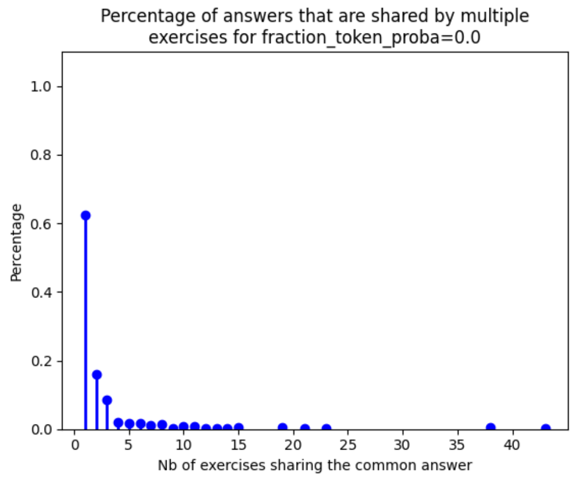


(a)

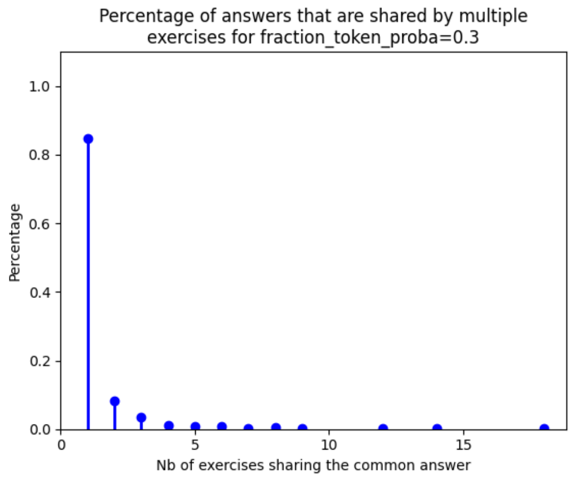


(b)

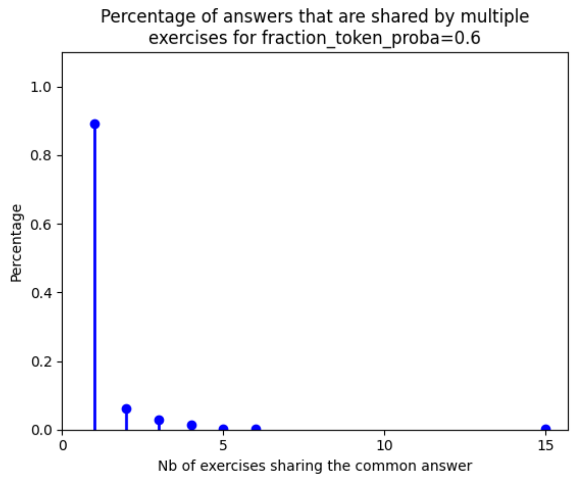
Figure 4.2: Generator evaluation regarding *complex_factor_proba* over samples of 1000 exercises



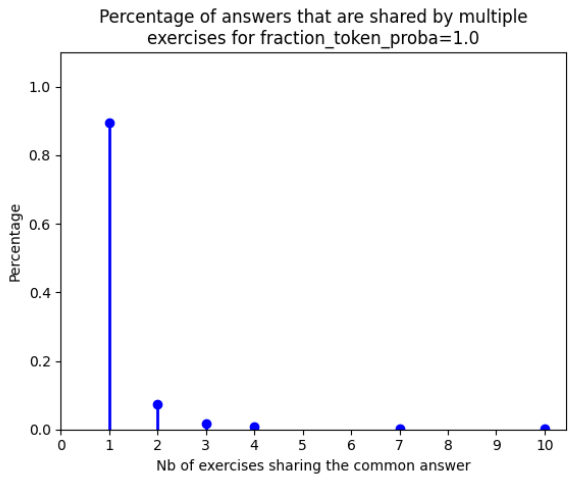
(a)



(b)

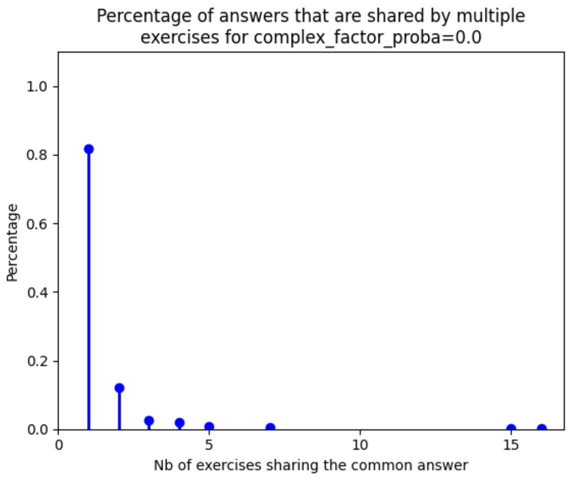


(c)

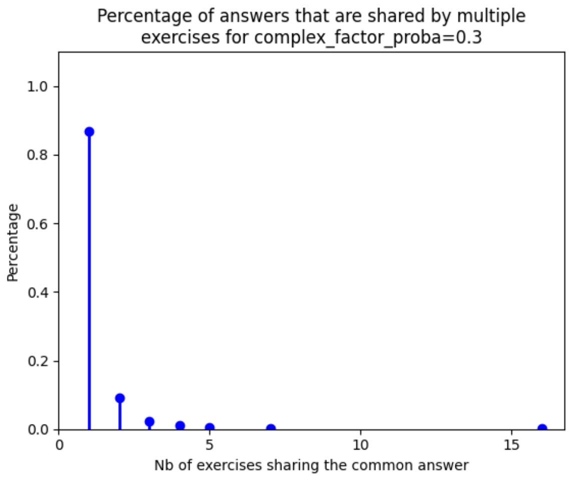


(d)

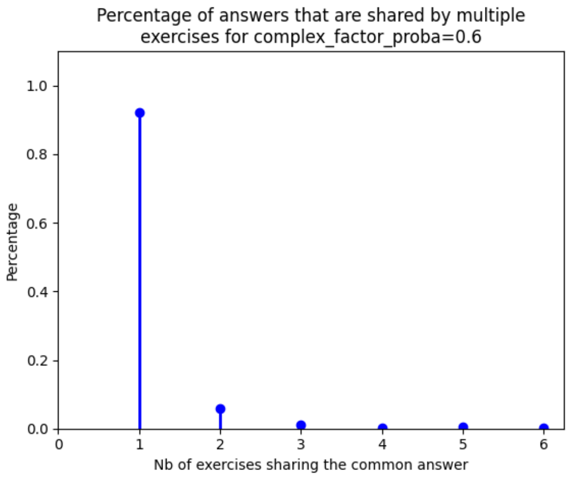
Figure 4.3: Diversity analysis with different values of *fraction_token_proba* over samples of 1000 exercises



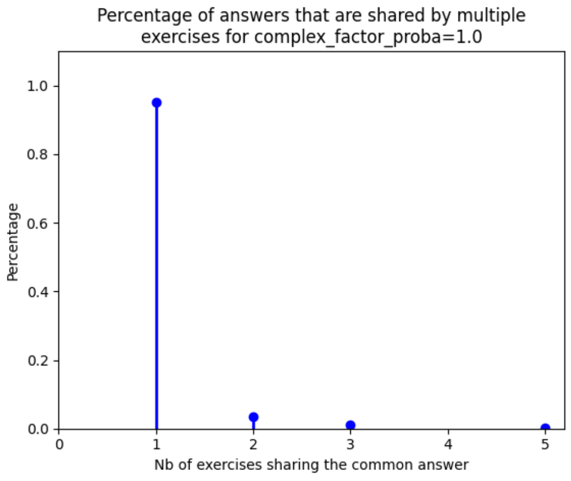
(a)



(b)



(c)



(d)

Figure 4.4: Diversity analysis with different values of *complex_factor_proba* over samples of 1000 exercises

4.3 Fraction simplification exercises

4.3.1 Exercise format

The simplest form an exercise can take is a fraction with integers as numerator and denominator that are not prime to each other (thus such that it can be simplified).

$$\text{Simplify :}\frac{36}{100}$$

We can then complexify the exercise by adding parameters:

$$\text{Simplify :}\frac{10x^2y^3}{12xy^2}$$

And using more complex expressions:

$$\text{Simplify :}\frac{10x^2y^3+4x^3y}{12xy^2}$$

Finally, we could even have several fraction that need to be put back together:

$$\text{Simplify :}\frac{9}{3(x^2-y^2)}+\frac{x}{x(y-x)}$$

But, as explained in section 4.3.3, the current version of the generator does not support the generation of such form of exercise.

We consider an answer to be as simplified as possible when there are no more common factor between the fully factorized numerator and denominator. We will not set any more constraint on the form of the answer.
Indeed, asking for the answer to be as expanded as possible would mean rejecting $\frac{3x(x^2+y)}{y}$ as an answer and accepting $\frac{3x^3+3xy}{y}$ which would seem counter-intuitive for many students.
On the other hand, asking for the answer to be as factorized as possible would mean asking to factorize $-12x^2+23x+1$ into $(24x-23-\sqrt{577})(-24x+23-\sqrt{577})$ as an example, which is not the aim of the exercise either.
Obviously, any constraint of the form "factorize but not too much" cannot be formally defined and could not be implemented in an automatized generator/corrector.

4.3.2 Parameters

In addition to the general parameters, we define 4 specific parameters to tune the fractions form:

- ***answer_ num_ complexity_ range*** is the expression complexity of the numerator of the answer.

- ***answer_ denom_ complexity_ range*** is the expression complexity of the denominator of the answer.

- ***nb_ factor_ range*** defines the lower and upper bounds to the number of factors that multiply both numerator and denominator of the answer to form the exercise.

- ***factor_ complexity_ range*** defines the lower and upper bounds to the expression complexity of each additional factor forming the exercise.

Those parameters are linked to the chosen way of generating exercises discussed in next section.

However, as explained in section 4.3.4, total compliance can be ensured only with *nb_factor_ range* while only partial compliance can be ensured with the other three parameters.

4.3.3 Generation

Choosing the generation method

There are 2 possible ways to generate a fraction simplification exercise:

- Generate random expressions for both numerator and denominator. This allows to easily generate any form of exercise, even those with several fractions that need to be put together.
  However, control over how much the fraction can be simplified is impossible and we cannot ensure that the generated fraction is even simplifiable.

$$\begin{aligned}\frac{3x+6}{x^2-y^2}+\frac{x+y}{4x(y-x)} &= \frac{(3x+6)4x(y-x)}{(x^2-y^2)4x(y-x)}+\frac{(x+y)(x^2-y^2)}{(x^2-y^2)4x(y-x)}\\ &= \frac{(3x+6)4x(y-x)+(x+y)(x^2-y^2)}{(x^2-y^2)4x(y-x)}\\ &= \frac{12x(x+2)(y-x)+(x+y)(x+y)(x-y)}{4x(y-x)(x+y)(x-y)}\\ &= \frac{-12x(x+2)+(x+y)(x+y)}{4x(y-x)(x+y)}\\ &= \frac{-12x^2-24x+x^2+2xy+y^2}{4x(y-x)(x+y)}\\ &= \frac{-11x^2+y^2+2xy-24x}{4x(y-x)(x+y)}\end{aligned}$$

- Generate the exercise's answer and then generate factors that will multiply both answer numerator and denominator to form the final exercise.

$$\begin{aligned}\textbf{Answer:}\ & \frac{3x}{y}\\ \textbf{First factor:}\ & x+y\\ \textbf{Second factor:}\ & 7y\\ \textbf{Final fraction:}\ & \frac{21x^2y+21xy^2}{7xy^2+7y^3}\end{aligned}$$

However, it would be difficult to generate exercises with several fractions that need to be put together with such a method.

The second method was chosen because being able to tune the exercises properties is too important whereas being able to generate the most complex forms of exercises is not a priority.

Exercise generation

The first step of fraction simplification exercise generation is to chose the parameters if any, that will appear in the exercise.

Then generate the answer expression by generating complex expressions for the numerator and denominator with randomly chosen complexities in accordance with *answer_num_complexity_range* and *answer_denom_complexity_range* and with the methods discussed in section 4.1.2.

We then must ensure that all chosen parameters are present in the answer with methods discussed in section 4.1.2.
We could do that at the end of the generation as all parameters do not need to appear in the answer but only in the final expression. That would however disturb the whole structure of the exercise and that is not acceptable.

$$\begin{aligned}\textbf{Exercise: } & \frac{3a^3+6a}{9a}\\ \textbf{Answer: } & \frac{a^2+2}{3}\\ \textbf{Factorizable: } & 3a\end{aligned}$$

Exercise with parameter

inserted in answer: $\dfrac{3(a+b)^3+6a}{9a+3ab}$

Answer: $\dfrac{a^2+2}{3+b}$

Factorizable: $3a$

Exercise with parameter

inserted at the end: $\dfrac{3(a+b)^3+6a}{9a}$

Answer: $\dfrac{3(a+b)^3+2a}{3a}$

Factorizable: 3

The price to pay however is that the complexity of numerator and/or denominator is slightly changed and exact compliance with *answer_num_complexity_range* and *answer_denom_complexity_range* cannot be ensured.

Then we can complexify the answer with several factors multiplying both numerator and denominator to form the final exercise.
The number of factors and their complexities are randomly drawn accordingly with *nb_factor_range* and *factor_complexity_range.*

Finally the exercise expression is expanded so as to make solving the exercise challenging.

4.3.4 Evaluation

Once again we want to ensure that the generated exercises are satisfactorily diversified.

The answer diversity is thus important and we will analyze it for different values of the *answer_num_complexity_range* and *answer_denom_complexity_range* parameters.

Moreover, diversity in the factors that can be factorized from the exercise expression is important as well. We do not want students to be able to systematically guess how they can factorize the fraction without searching by themselves. We will thus analyze that factor diversity with different values of *factor_complexity_range.*

We will see that the results greatly depend on the number of parameters involved in the exercises and thus repeat the analysis listed above for several values of *nb_param_range.*

Graphs explanation

Only notions that have not already been discussed in section 4.2.4 will be discussed in this section.

Figure 4.5 is about the notion of answer complexity which in this case is defined as the sum of both answer numerator and denominator complexities. If we have *answer_num_complexity_range*=2 and *answer_denom_complexity_range*=2, we thus expect the answer complexity to be equal to $2 + 2 = 4$.
Keep in mind however that on figure 4.5 the value of *Answer numerator/denominator complexity* is the value of separately *answer_num_complexity_range* and *answer_denom_complexity_range* and not their sum.

Figure 4.6 shows an analysis of factors diversity. This notion is quite similar to answer diversity and is defined as the ratio between the number of different factor values that have been encountered and the number of factors that were considered.

The graphs were generated in the same fashion as described in section 4.2.4.
Both evaluations were performed on the same sampled exercises as their answers and factors values do not influence one another. Those evaluations were performed for three values of *nb_param_range*: [0, 0], [1, 1], [2, 2].

Only graphs with the parameter=[0, 0] and [1, 1] are displayed in figures 4.7 to 4.12 as those are the most interesting results.

The chosen values for *answer_num_complexity_range*, *answer_denom_complexity_range* and *factor_complexity_range* were [0, 0], [1, 1], [2, 2] and [3, 3] as those are the most likely to be used when ordering exercises.

The only static parameters' values were *nb_factor_range*=[1, 1] as more would have been useless and without consequence on the analysis and *max_int_value*=5 as such a value is likely to be chosen when ordering exercises.

Graphs analysis

First of all we can see on graphs 4.5a and 4.6a that the measured complexity is zero no matter what the parameter's value is when there is no parameter in the exercise. This is because in that case, we only deal with numbers which naturally associate into a single number no matter what operators are applied to them.

$$\frac{5+3\times 6}{7-3} = \frac{23}{4}$$
$$\text{Complexity} = 0$$

In the case where there are no parameter in the exercise, complexity parameters' values thus cannot be reflected by the actual expressions' complexities as defined in section 4.1.2 but only in the sizes of the numbers involved.

For the same reason, we can observe on those same graphs that even with more than zero parameter, the measured complexities are substantially lower than expected (for an *Answer numerator/denominator complexity* value of 3, we expect an answer complexity of 6).

$$\frac{a+3b}{7a}$$
$$\text{Complexity} = 2+1 = 3$$

$$\frac{a+3\times 6}{7a} = \frac{a+18}{7a}$$
$$\text{Complexity} = 1+1 = 2$$

Another thing that we can note on graph 4.5a is that for an expected complexity of 0 however, with some parameters in the exercise, the measured complexity is non-zero. This is because with such a value for *answer_num_complexity_range* and *answer_denom_complexity_range* there should only be 2 tokens forming the answer (one at the numerator and one at the denominator). However, as soon as 2 numbers (1 if *nb_param_range*=[2, 2]) are chosen over the exercise's parameters during token generation, at least one asked parameter cannot appear in the exercise answer, and is thus inserted by force.

$$\text{Expected complexity} = 0$$
$$\text{nb\_param\_range} = [2, 2]$$
$$\text{Generated answer :} \frac{a}{7}$$
$$\text{Parameter insertion :} \frac{a+b}{7}$$
$$\text{Complexity} = 1+0 = 1$$

Next, we can see on graphs 4.5b and 4.6b that for a zero complexity, the diversity is close to 0, which seems pretty bad.
The reason is that for such a complexity, any factor or answer's numerator/denominator can only take the possible integers or parameters, negative or positive, as a value.
As an example, the numerator's value can only be ± 1, ± 2, ± 3, ± 4, ± 5, $\pm a$ or $\pm b$ with *nb_param_range*=[2, 2] and *max_int_value*=5, that is 14 possible values.

Considering the fact that some possible values of the answer are redundant due to simplification ($\frac{5}{5} = \frac{4}{4} = ...$), or that a factor cannot be equal to ± 1 as it would be without impact on the expression, that yields at most a few tens of possible values for 1000 exercises/factors and that explains the results on graphs 4.5b and 4.6b.

We can note on the other hand, that the general tendency of the diversity to increase with the number of parameters is expected. Indeed, more parameters means more possible values and values that are not miscible.

$$\frac{1+4}{3} = \frac{2+3}{3} = \frac{5}{3}$$
$$\frac{1+a}{3} \neq \frac{1+b}{3}$$

Let's now take a closer look into the answers/factors' values distribution over the sampled exercises/factors on figures 4.7 to 4.12.

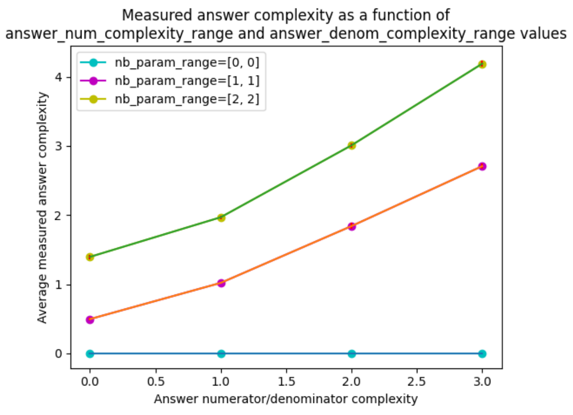
We can see on graph 4.10a that the factors' values are all shared by a similar amount of sample factors. This is expected as in that case, there are only 8 possible values (± 2, ± 3, ± 4 and ± 5) which are all equally likely to be drawn. We thus have for each value representing $\frac{1}{8} = 12.5\%$, $\frac{1000}{8} = 125$ factors sharing it which is roughly what we can observe on the graph.

As for the other graphs, we can make the observation that most of answers/factors' values are shared by relatively few exercises/factors and only a few extreme values are shared by a greater number of exercises/factors.
Let's take graph 4.7a. We can see that there are 2 values shared by about 100 exercises. This is due to the answers' possible values redundancy discussed above, making 1 and -1 far most likely as an answer than other values.
If we add a parameter, on graph 4.8a, then because of the token generation drawing any parameter as likely as a number in general (here 50% chance of drawing $\pm a$ and 50% chance of drawing a number in ± 1, ± 2, ± 3, ± 4 or ± 5), $\pm a$ will be far more redundant than any other value as a token. That explains the two isolated values around 140.

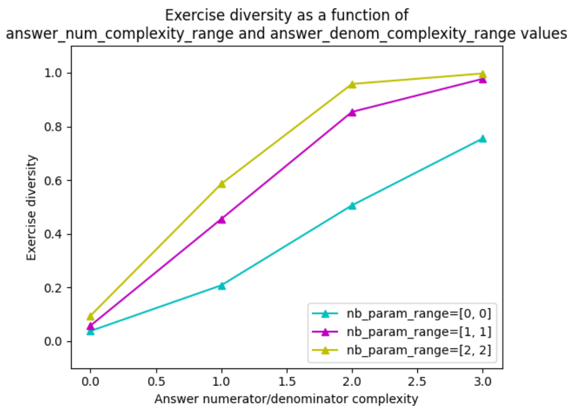
Another interesting case is graph 4.11b. The single isolated value at about 150 is due to both facts that the single parameter is far more likely to be drawn than any other number and the fact that with a complexity of 1, expressions $a \times a$, $-a \times -a$, a^2 and $(-a)^2$ all yield the same value (a^2). It thus becomes in those conditions the most likely value for a factor by far.

All those combined mechanisms explain the other isolated points on the other graphs. They are all natural consequences of the way the generator is implemented.
Although there are these unexpected at first but explainable extreme cases, they do not seem to be extreme enough to generate real problems.

Should this special cases however prove to be truly problematic (by testing through practice), one could remedy the problem by tuning the probability of appearance of the Power operator, or the probability distribution between parameters and numbers in token generation for example.

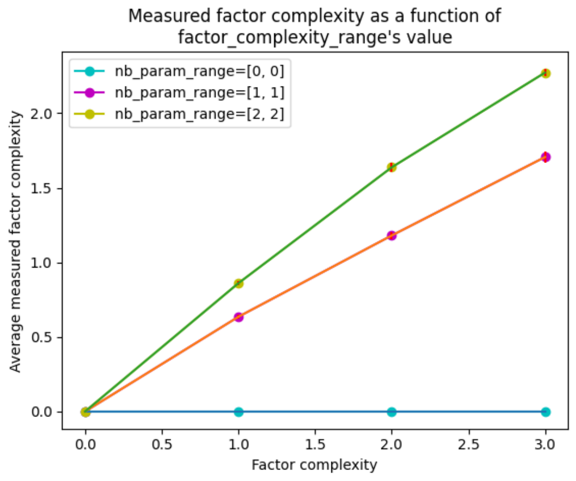


(a)

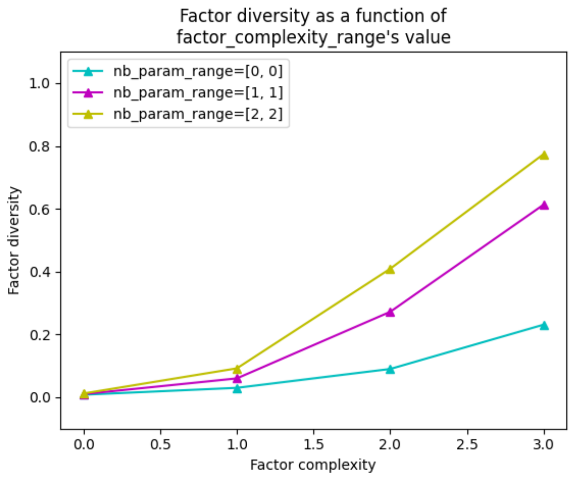


(b)

Figure 4.5: Generator evaluation regarding *answer_num_complexity_range* and *answer_denom_complexity_range* over samples of 1000 exercises

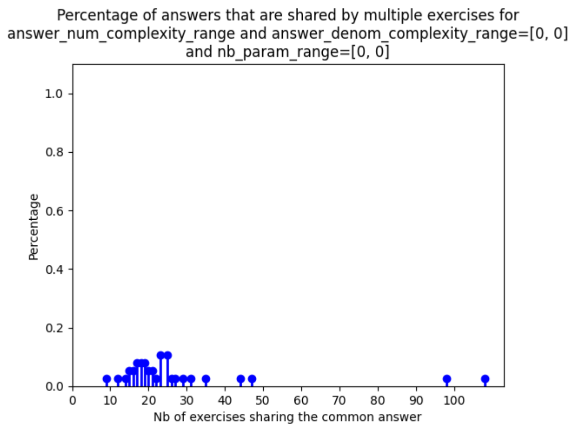


(a)

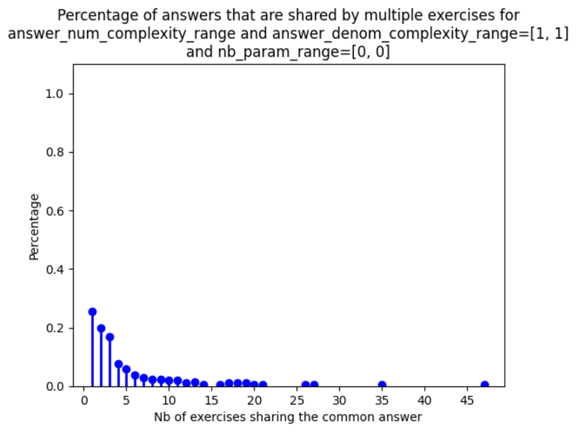


(b)

Figure 4.6: Generator evaluation regarding *factor_complexity_range* over samples of 1000 factors

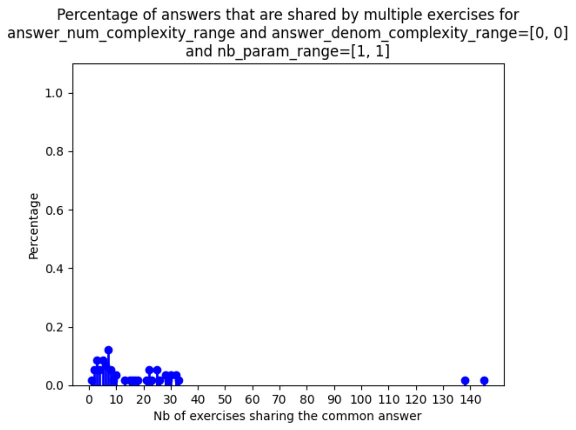


(a)

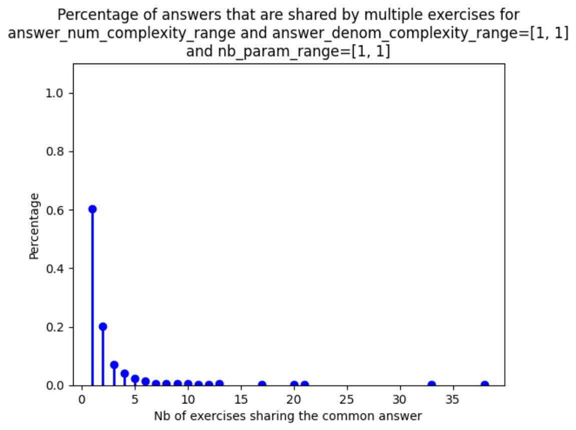


(b)

Figure 4.7: Diversity analysis with different values of *answer_num_complexity_range/answer_denom_complexity_range*, 0 parameter and over samples of 1000 exercises

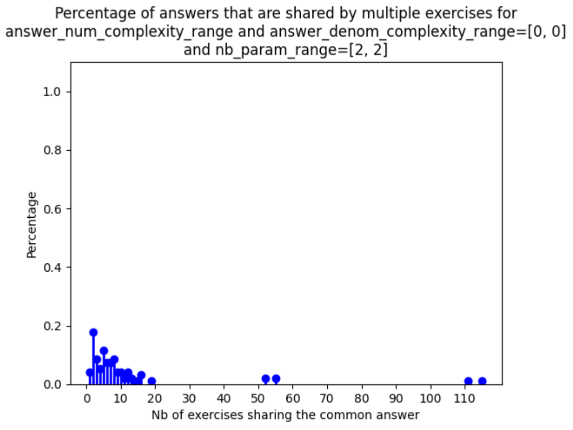


(a)

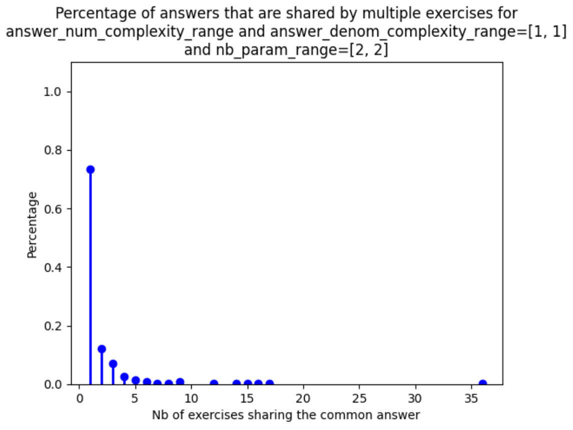


(b)

Figure 4.8: Diversity analysis with different values of *answer_num_complexity_range/answer_denom_complexity_range*,, 1 parameter and over samples of 1000 exercises

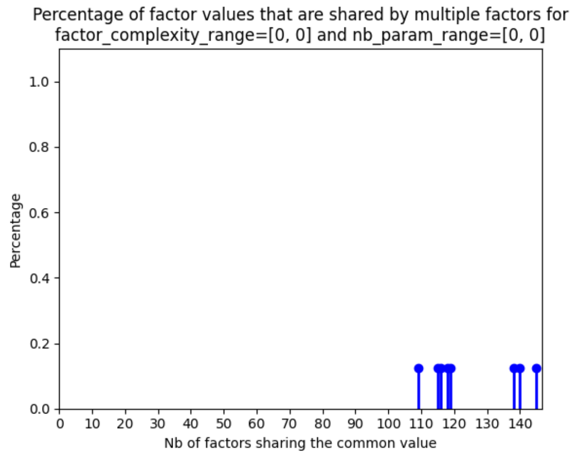


(a)

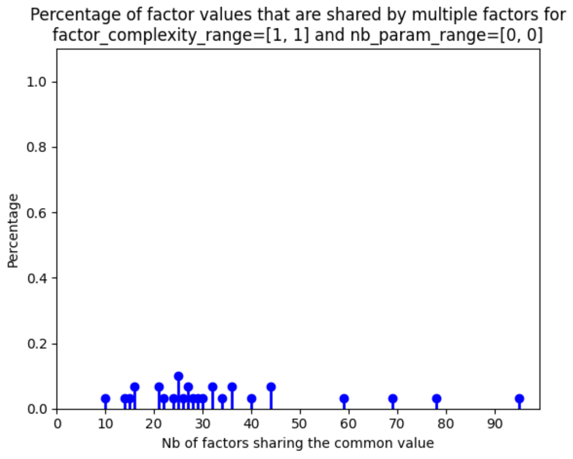


(b)

Figure 4.9: Diversity analysis with different values of *answer_num_complexity_range/answer_denom_complexity_range*, 2 parameters and over samples of 1000 exercises

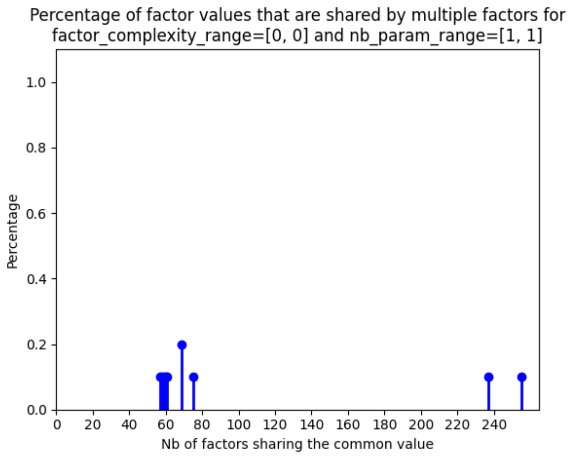


(a)

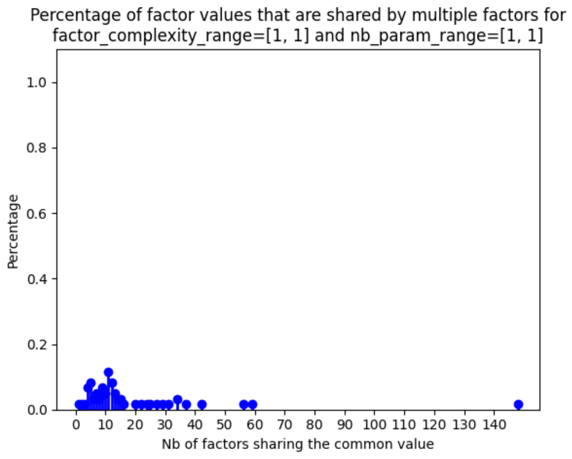


(b)

Figure 4.10: Diversity analysis with different values of *factor_complexity_range*, 0 parameter and over samples of 1000 exercises

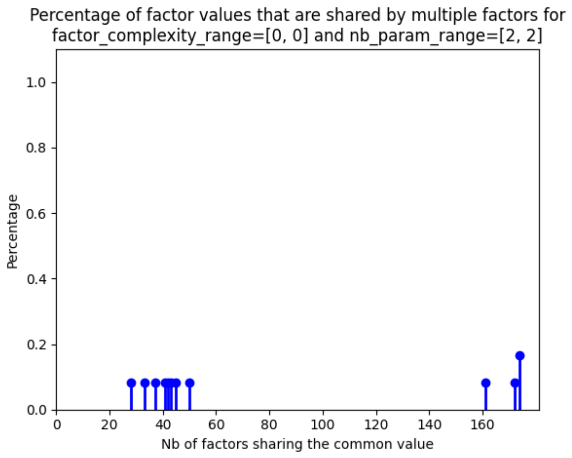


(a)

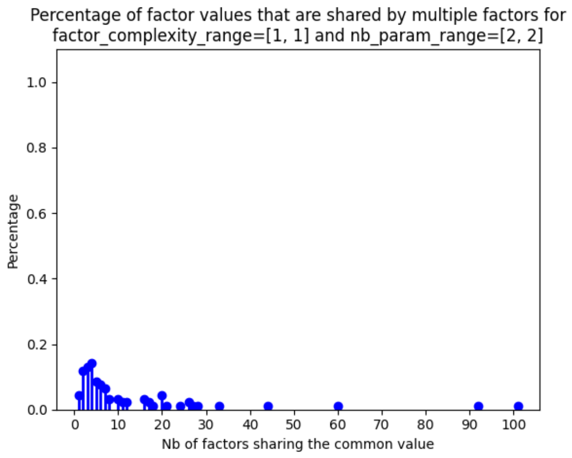


(b)

Figure 4.11: Diversity analysis with different values of *factor_complexity_range*, 1 parameter and over samples of 1000 exercises



(a)



(b)

Figure 4.12: Diversity analysis with different values of *factor_complexity_range*, 2 parameters and over samples of 1000 exercises

4.4 Extensibility

What has already been implemented for those two generators may serve as well for the generation of new types of exercises.

Factorization exercise generation could reuse factor generation defined for equation simplification exercises.
Second degree equations may be build from the multiplication of two first degree factors whose generation is already implemented by the first degree equation generator.

Chapter 5

Feedback generation

We want to be able to provide consistent feedback to students when they provide wrong answers to the exercises.

The chosen way to do that is by defining a set of mistakes that can be performed by a student associated with their corresponding feedback and compute all the ways to introduce those mistakes at any point of the exercise resolution. The resulting wrong answers are compiled with their respective feedback and ready to be sent along with the exercise to the students.
Their application will then be able to provide the correct feedback in addition to detecting whether the provided answer is right or wrong.
This is asynchronous feedback generation.

This allows to propose MCQ exercises to the students as the application only has to chose some already computed and plausible wrong answers along with the correct one and propose them to the students. If a wrong answer is chosen by the student, the corresponding feedback can directly be returned.

However, this is not sufficient for open question exercises. Indeed, as explained in the next section, computing the possible wrong answers in an exhaustive manner is impossible. The sets of wrong answers provided with the exercises to the application will thus be incomplete and a student may then make a mistake for which the application will be unable to provide any feedback.

To solve this problem, we provide as well the possibility to reach back to the server and ask it to detect the mistake and provide a consistent feedback based on the written reasoning of the student.
This is synchronous feedback generation.

The reason why not everything is done synchronously is that this would mean that the server hosting My Success Squared would have to manage each and every student that is active on the app and solving exercises. This could cause a major bottleneck problem, with long delays to receive feedback and an overloaded server. Such a model would thus clearly not be scalable.

5.1 Asynchronous feedback generation

5.1.1 General principles

The asynchronous feedback generator should provide the set of possible wrong answers associated with their respective feedback for a given exercise.

To produce that set, we can build a tree structure whose root corresponds to the initial expression of the exercise and each other node of the tree corresponds to the result of the application of a certain step (correct or not) to the expression of the parent node. The leafs of the tree then correspond to the different answers resulting of the path between the root and themselves.
This tree structure represents all the ways to solve an exercises, with or without mistakes, that is, all the possible reasonings for solving the exercise. We thus call that structure a reasoning tree.

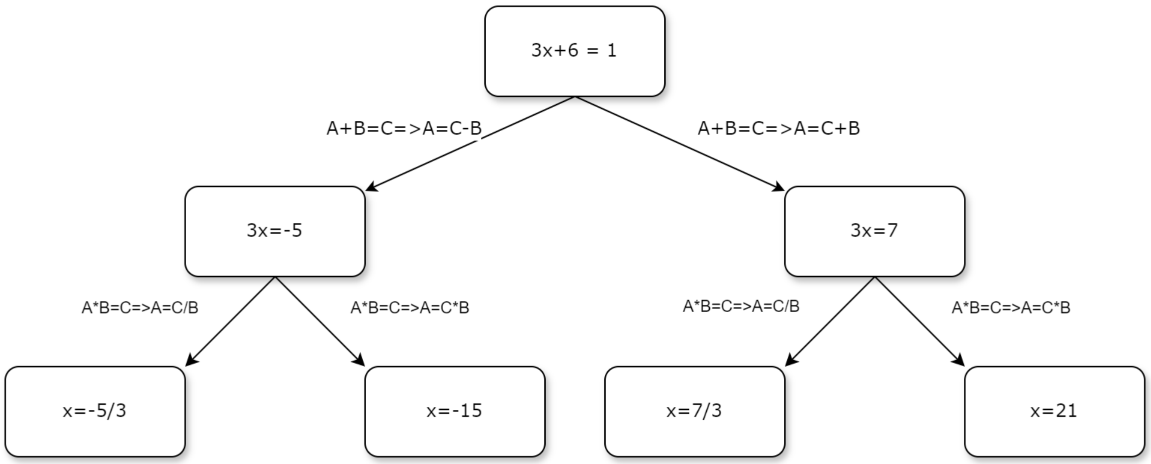


Figure 5.1: Simple reasoning tree example

Then for each leaf, we can retrieve the corresponding feedback by checking the path from the root to the leaf.

Answer : -15
Feedback :"You don't maintain equality. On the left-hand side, you divide by B, while on the right-hand side you multiply by B."

5.1.2 Reasoning steps

We need to define a reasoning step set that allows to explore all the possible ways to introduce the mistakes under consideration.
For now we define the steps:

Correct:

$$\begin{gathered} A+B=C \rightarrow A=C-B \\ AB=C \rightarrow A=\frac{B}{C} \\ \frac{AB}{AC} \rightarrow \frac{B}{C} \end{gathered}$$

Incorrect:

$$\begin{gathered} A+B=C \rightarrow A=B+C \\ AB=C \rightarrow A=BC \\ AB=C \rightarrow A=B+C \\ A+B=C \rightarrow A=\frac{B}{C} \\ \frac{A}{B}=\frac{C}{D}E \rightarrow \frac{A+C}{B+D}=E \\ AB+AC \rightarrow A(B+AC) \\ \frac{AB}{AC} \rightarrow \frac{AB}{C} \\ \frac{AB}{AC} \rightarrow \frac{B}{AC} \end{gathered}$$

Some of them are incorrect and thus define the mistakes that are considered. The others are necessary to be able to introduce mistakes later in the development instead of right at the beginning (see figure 5.1).
We can also note that some involve the equality sign (equation step) and some do not (non-equation step).

These steps are defined in the most general way such that it is not necessary to consider whether the expression mapped to A for example involves or not the unknown of the equation, etc.

In order to be able to distinguish more probable mistakes from rarer ones, we also associate a weight to those steps (the higher the rarer).

5.1.3 Reasoning tree

Basic algorithm

The tree generation algorithm is theoretically very simple.

Algorithm 2 Reasoning tree building

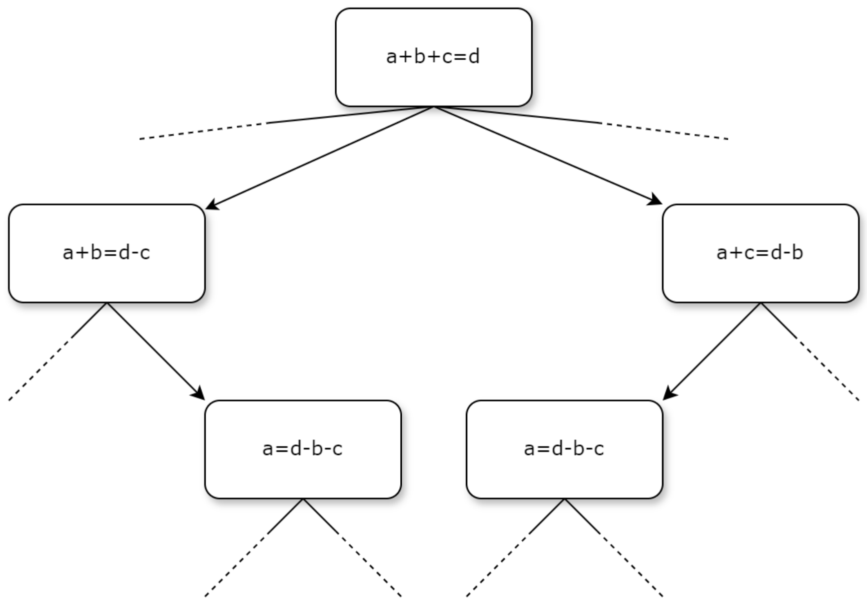
```
procedure ExpandTree($tree$, $fromNode$)
    if $fromNode$ is leaf then
        return $tree$
    else
        for $step$ in $steps$ do
            $resultingExpressions \leftarrow$ ApplyStep($step, fromNode$)
            for $expression$ in $resultingExpressions$ do
                $parent \leftarrow fromNode$
                $child \leftarrow$ NewChild($expression, parent$)
                $tree \leftarrow$ ExpandTree($tree, child$)
            end for
        end for
    end if
end procedure

$tree \leftarrow$ Tree()
$root \leftarrow$ SetRoot($exersiseExpression, tree$)
$tree \leftarrow$ ExpandTree($tree, root$)
```

We apply the tree expansion algorithm starting at the root of the tree, thus the initial exercise expression :
We compute the expressions resulting from all the possible ways to apply each step to the considered node's expression.
We then create a new node for each one of these expressions and add them to the tree as children of the considered node.
We then recursively apply this algorithm to those children nodes until we get to the leafs.

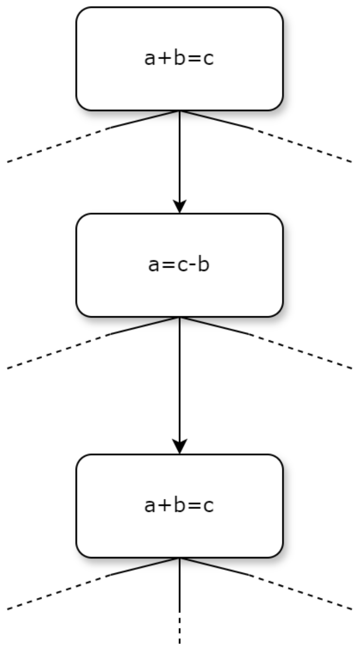
However, this is a simplified scheme and there are some phenomena that force us to complexify it.

Ensuring termination

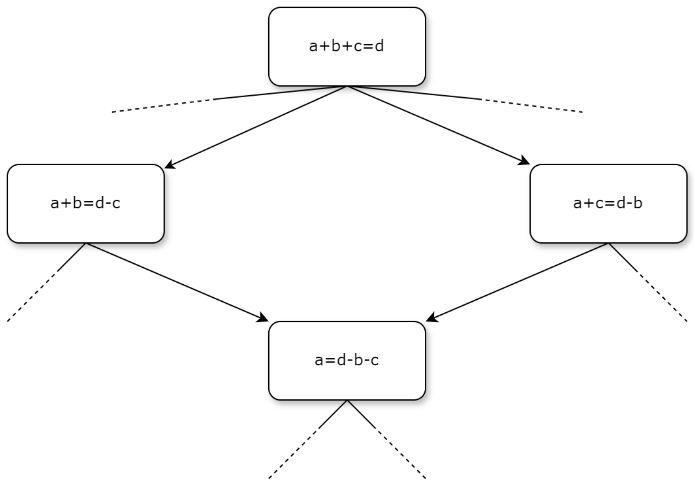
First of all, several path may lead to the same expression and thus the same node. A first consequence is that the same computations may be performed several times in different places in the tree, which is not very efficient.

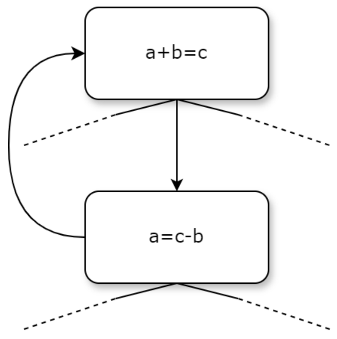


The other far more problematic consequence is that even loops can appear, generating an infinite tree.

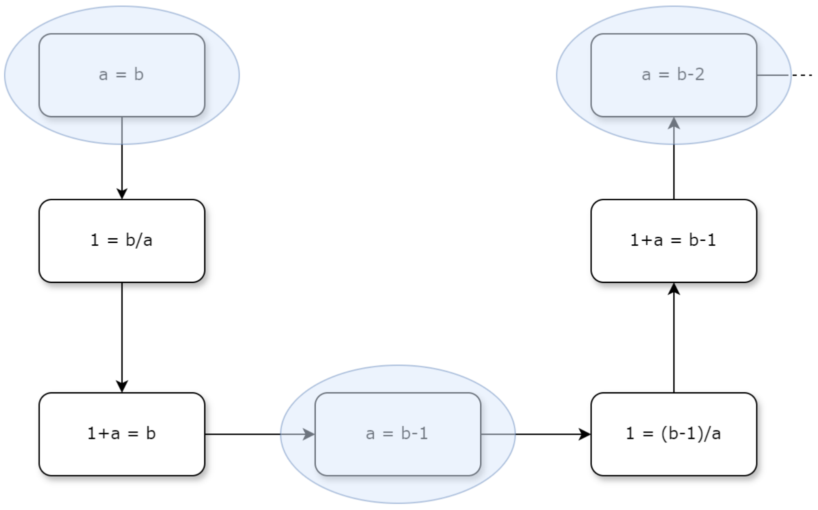


The solution to that first problem is to stop extending the tree as soon as a node that is identical to an already existing node is reached and merge both identical nodes.





There is another infinite loop problem caused by the fact that some series of erroneous steps may come back to an expression similar to the initial one, but with different values. The resulting expression's node is thus not already existing and we are yet again in an infinite tree extension scheme.



The only way to prevent that is to bound the tree to a certain depth. This depth however should be high enough for the tree to be able to reach the end of a reasoning so that mistakes introduced close to the end of a reasoning can be taken into account as well.

We now have an algorithm for generating the tree that is guaranteed to terminate.
Indeed the first level of the tree only contains 1 (the root) and thus, a finite number of nodes.
At each level, a finite set of steps is applied to each node which thus have a finite set of children.
Any level of the tree thus contains a finite set of nodes and as the tree depth is bounded (finite), the number of nodes in the final tree is finite and the tree building thus necessarily terminates.

To be exact, this is true if we ensure that each step may apply to any expression in a finite set of ways and that the algorithm to compute this set and apply its element to any expression terminates. This will be the matter of section 5.1.5.

Time and space constraints enforcement

Termination is not the only criteria that has to be met. We have to ensure a certain time and space efficiency. We would not want the feedback generation to take hours or an exercise to come along with Gb of feedback.

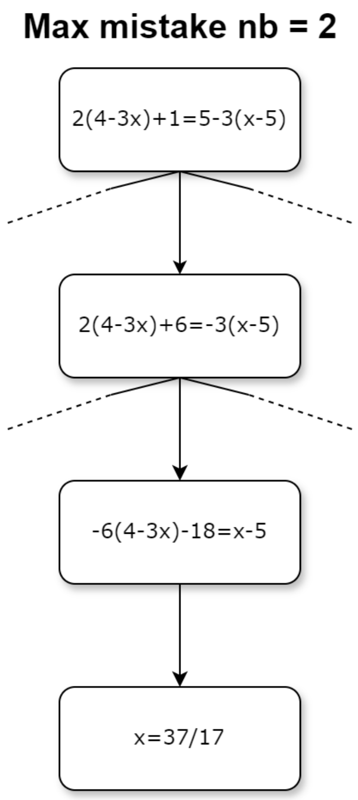
As for the time constraint, we decided that feedback generation for an exercise should not take an average time greater than 1 minute. Indeed, although it may seem quite long, as this is asynchronous generation and the exercises are thus generated in advance, it remains acceptable.

The algorithm as designed until now is not able to comply with the time constraint. Indeed, feedback generation can take several tens of minutes or more for relatively complex exercises.
This is due to the fact that the size of a full tree structure is $n^{depth} - 1$, with n the number of child per non-leaf node, and depth the depth of the tree.
In the case of a reasoning tree, the depth grows with the exercise complexity as more steps are needed to solve the exercise ($depth = f(complexity)$ with f a strictly increasing function) whereas n is proportional to the number of different possible steps step_nb that can be applied to each node multiplied with the average number of ways to apply those steps to a given expression. This again increases with the exercise complexity ($n = step\_nb \times g(complexity)$ with g a strictly increasing function). The size of a reasoning tree can thus be approximated to :

$$(step\_nb \times g(complexity))^{f(complexity)} = e^{f(complexity) \times (ln(step\_nb) + ln(g(complexity)))}$$

We can see that the complexity of an exercise has a great impact on the size of the tree as $f(complexity)$ might not be precisely defined but is most probably proportional to $complexity$. On the other hand, $step\_nb$ has a smaller yet significant impact on the tree size.
As the nodes have to be added one by one, the time to build the tree is thus proportional to its size.

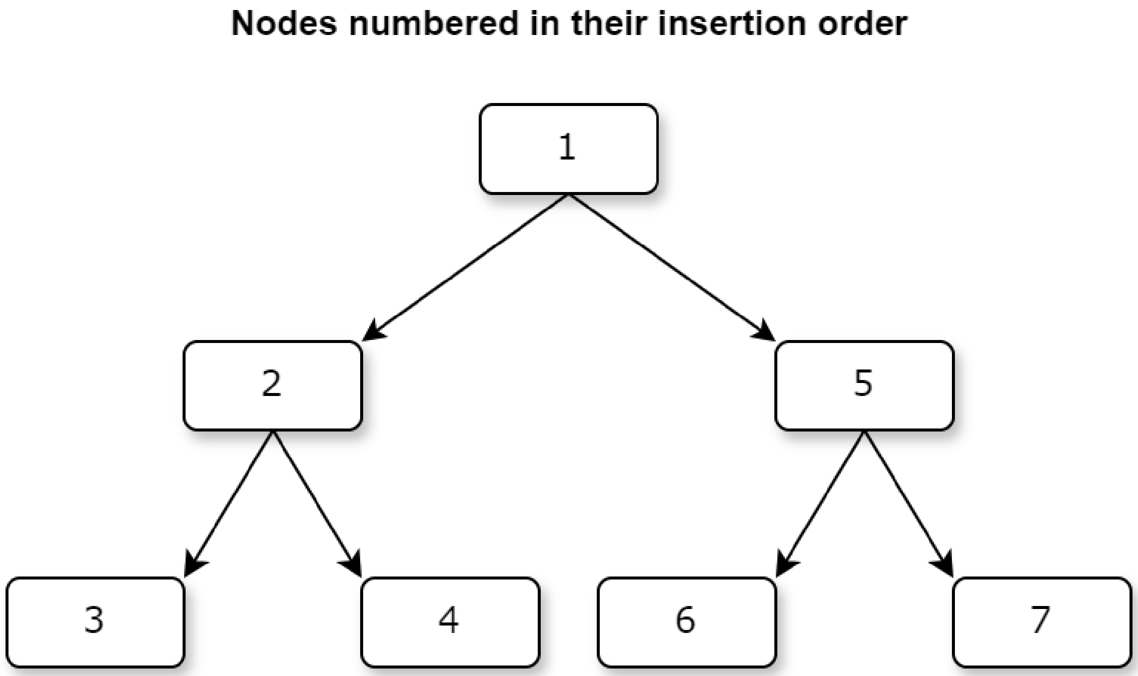
A first solution is to define a maximum number of mistakes that can be introduced in a reasoning. Indeed, we can consider that students have passed the tutorial and will not make more than a few mistakes.
This is a way to reduce the tree's depth as the tree is not full anymore and the average depth is thus decreased.
We arbitrarily set that maximum number to 3.
Correct steps are applied during tree extension only to be able to reach further into the reasoning and consider more possible mistakes. We can thus stop the tree extension as soon as the maximum number of mistakes have been reached in the considered path and directly link the last added node to the corresponding result (tree leaf).



This can greatly reduce the size of the tree and thus the computation time.

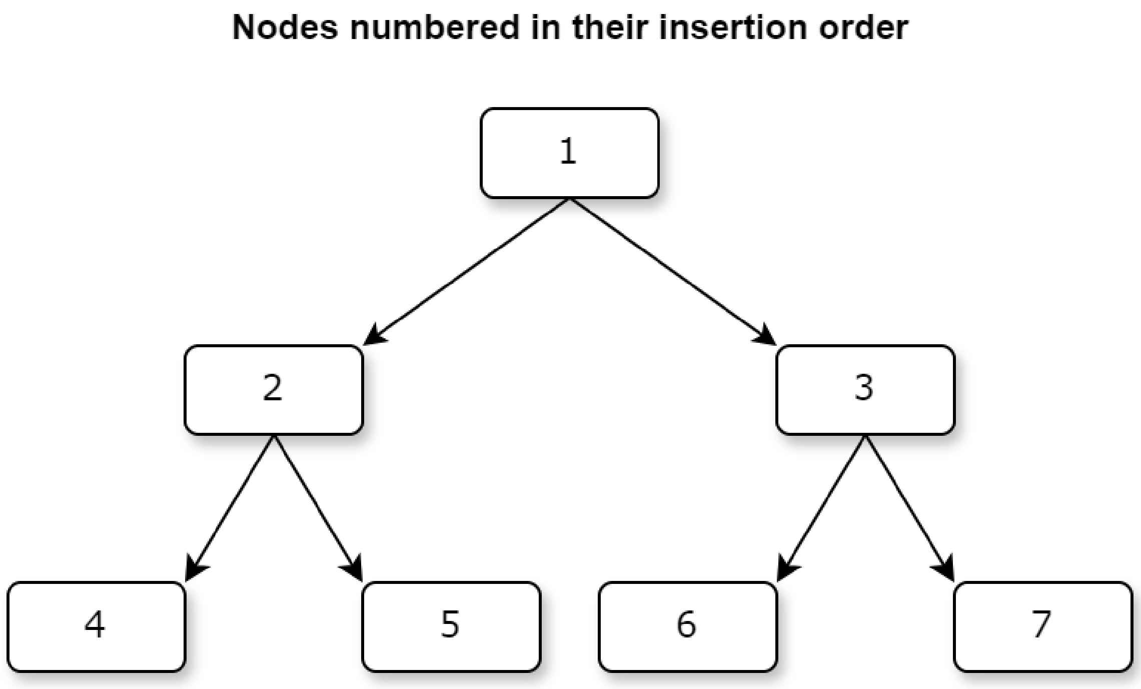
However, this is not enough yet as feedback generation for complex exercises still take several tens of minutes.
As we were not able to identify any unnecessary tree development scheme that could be eliminated, the solution we chose is to bound the number of wrong answers considered. The principle is simple, as soon as the number of leafs in the tree (which are unique as redundant nodes are merged together) exceeds a certain threshold, we stop the extension of the tree. This allows to efficiently manage the time complexity of the generation as well as its space complexity. Setting 500 as the maximum threshold should ensure for most of the exercises that the feedback generation takes less than a minute and guarantees an acceptable use of space per exercise.

There is one problem however with this solution as it is. Indeed, the tree is extended in a depth first manner.



As a consequence, if an early branch ends up in so much different leaves that the threshold is exceeded, other early branches will not be explored at all. The result is then 500 feedbacks corresponding to reasonings that all begin in the same way which is a serious lack of diversity and thus quality.
We prefer a feedback generator that misses evenly distributed possibilities over a feedback generator that is complete for a very specific category of reasoning but useless for the rest of the possible reasonings.
Moreover, with such a scheme, the number of generated reasonings containing n mistakes increases with n although we would like it to be the opposite as it is considered to be less likely for a student to make many mistakes than just a few.

The solution would be to change the tree extension algorithm into more of a width-first scheme.



More precisely, we will first generate only reasonings containing one single mistake. Then if all the possibilities have been explored and the maximum leaf number threshold has not been exceeded yet, resume the tree extension. Starting from the erroneous nodes generated during last step, generate all the reasonings containing two mistakes. Repeat this last step until either the threshold is exceeded or the maximum number of mistakes to be considered per reasoning is reached.
We thus need to keep the list of erroneous nodes generated during previous step in order to be able to perform next step.

In order to prevent the generated feedback to depend on the order in which the reasoning steps have been defined, the erroneous nodes from which to resume the tree extension at each step are chosen randomly.
Also, as the tree extension is not complete anymore, while applying reasoning steps in a random order we should still give the priority to reasoning steps that have a lower weight.

5.1.4 Associate feedback to answer

Once the tree is built, we still need a way to find the correct feedback to associate to each wrong answer (leaves of the tree).
For that purpose, each node associates to each one of its parents (there may be several paths from the root to a same node) the step by which the node results. We can then know for each path the mistakes that were introduced and thus the corresponding feedback.

Each leaf is linked to the root through several paths, i.e. through several possible reasonings. For each leaf, we can then select the most probable path that we define as the path for which the sum of its mistakes' weights is the smallest (thus few mistakes and/or probable ones).

However, visiting each one of the possible paths can be very long and inefficient.
Instead, we run Dijkstra's algorithm to efficiently find the shortest weighted path between a leaf and the root without visiting each path from end to end.
It is important to run Dijkstra's algorithm from the leaf and to the root and not the opposite. Indeed, only the corresponding branch gathering all the possible paths is reachable from the leaf before reaching the root whereas the entire tree is reachable from the root. Added the fact that the tree is mainly composed of nodes resulting from non-erroneous steps that have a zero weight and the one who knows Dijkstra's algorithm will understand that it applied to the root is not so different from the exclusive path comparison.

The feedback generator associates to each considered answer the feedbacks associated to all the mistakes that appear in the reasoning and in their order of appearance.
It also gives for each mistake the generic step expression as well as the corresponding specific expression in the computed reasoning.

5.1.5 Reasoning step application

We still need to address the problem of applying a reasoning step represented by a generic expression to an exercise expression.

$$\begin{aligned}\textbf{Expression before :}& 4x + 3 = 1\\ \textbf{Step expression :}& A + B = C \rightarrow A = C - B\\ \textbf{Expression after :}& 4x = -2\end{aligned}$$

Mapping generic expression to specific expression

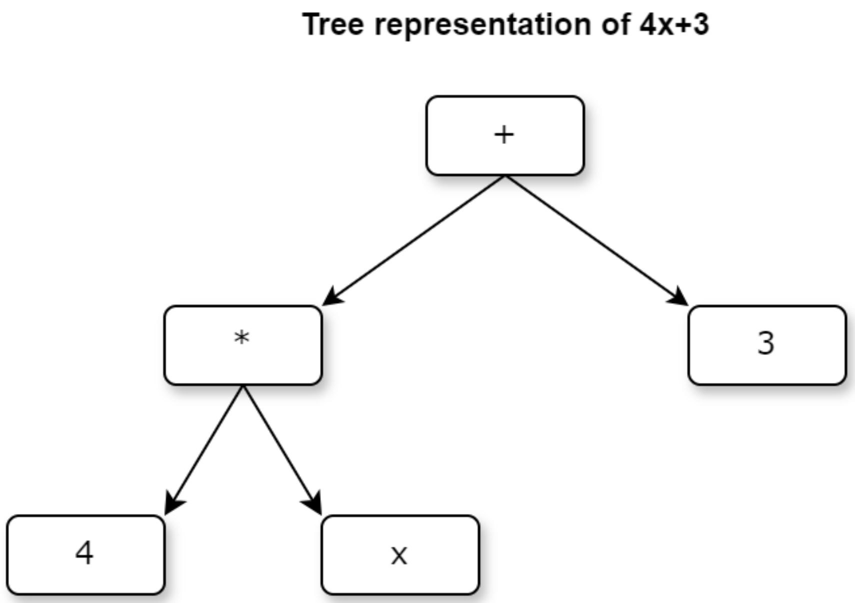
To do that, we need a way to map the symbols in the step's generic expression to sub-expressions of the exercise.

$$\begin{aligned}\textbf{Expression before :}& 4x + 3 = 1\\ \textbf{Prior step expression :}& A + B = C\\ \textbf{Mapping :}& \{A = 4x, B = 3, C = 1\}\end{aligned}$$

Once we have that mapping, we can rebuild the resulting specific expression.

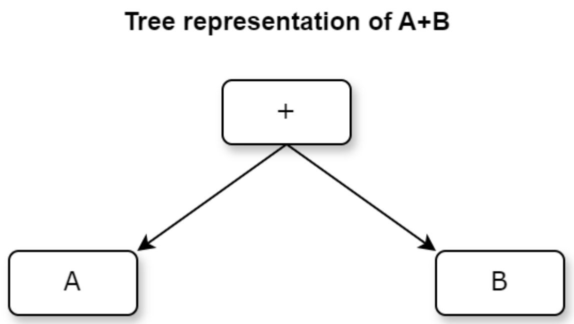
$$\begin{aligned}\textbf{Posterior step expression :}& A = C - B\\ \textbf{Mapping :}& \{A = 4x, B = 3, C = 1\}\\ \textbf{Expression after :}& 4x = -2\end{aligned}$$

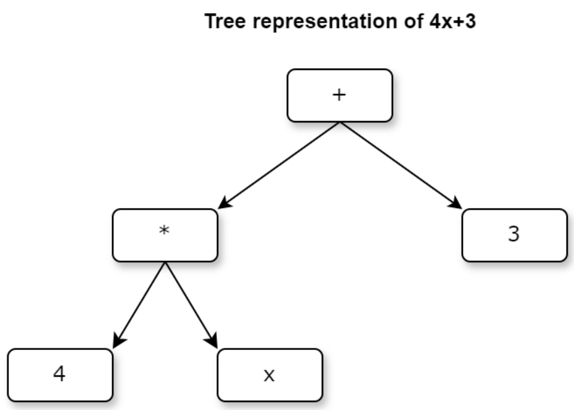
In order to build the mapping, we can use the fact that in Sympy, expressions are represented as tree structures.



(In Sympy, equations cannot be represented as such and we thus divide it into 2 expressions, one for each side of the equation.)

We can then recursively explore both generic and specific expressions' trees at the same time.
While the leafs of the generic tree are not reached yet, we just compare both trees as both considered operators should be equal. If the comparison fails, there is simply no mapping possible and the step cannot be applied (A+B cannot be mapped to 4x).
Once the generic tree leaves are reached, the corresponding specific sub-expression is mapped to the generic symbol.





mapping :$\{A = 4x, B = 3\}$

Note that a symbol may appear several times in the generic expression.

Step expression :$AB + AC \to A(B + C)$

$3x + 4y$:No mapping

$3x + 3y\{A = 3, B = x, C = y\}$

Dealing with several possible mappings

Most of the time, there is not only one single possible mapping an thus not only one single way to apply a reasoning step to an expression.

Mapping $A + B = C \to A = C - B$ **to** $x + y + 1 = 3$ **:**

Mapping	Resulting expression
$\{A = x, B = y + 1, C = 3\}$	$x = 2 - y$
$\{A = x + y, B = 1, C = 3\}$	$x + y = 2$
$\{A = x + 1, B = y, C = 3\}$	$x + 1 = 3 - y$
$\{A = y, B = x + 1, C = 3\}$	$y = 2 - x$
$\{A = y + 1, B = x, C = 3\}$	$y + 1 = 3 - x$
$\{A = 1, B = x + y, C = 3\}$	$1 = 3 - x - y$

We thus need an algorithm to compute all the ways of mapping the generic expression's arguments to the specific ones. In other words, we need an algorithm to compute all the ways to distribute m elements in n boxes with $m \geq n$.
This algorithm is as follow:

Find all the combinations of between 1 and m-(n-1) elements to put in the first box (there needs to remain at least n-1 elements to be distributed in the n-1 remaining boxes). This can easily be done with the *combinations()* function of the python *itertools* library.
Then for each combination, associate it with each one of the possibilities computed by the algorithm applied to the remaining elements and boxes.

Equation and non-equation reasoning steps

As said earlier, equations cannot really be represented by a single expression but rather by a pair of expressions, each corresponding to the right or left hand side of the equation. There are thus 2 ways of applying an equation reasoning step to an equation exercise.

Mapping $A + B = C$ to $x + 1 = x - 6$:

Mapping
$\{A + B = x + 1, C = x - 6\}$
$\{A + B = x - 6, C = x + 1\}$

Naturally, an equation step cannot be mapped to a non-equation exercise (ex: fraction simplification exercise).

On the other hand, non-equation steps may potentially be applied to the exercise expression (expressions if it is an equation exercise) but also to each sub-expression of the exercise.

Mapping $AB + AC$ to $2(x + y) + 2(3x + 3y)$:

Mapping
$\{A = 2, B = (x + y), C = (3x + 3y)\}$
$\{A = 3, B = x, C = y\}$

We thus need to check for ways of applying the reasoning step starting at any node of the specific expression tree.

Generalized and specific reasoning step application

Although this reasoning step application scheme is quite elegant in its generality, some reasoning steps unfortunately need a more specific processing.

Indeed steps like $AB + AC \to A(B + C)$ may encounter some problems namely if dealing with numbers. As an example, it would not be possible to factorize $2x + 6$ into $2(x + 3)$ as 6 and 2×3 are not the same in the sense of expressions.
There is also the fact that with Sympy, $2(x + 3)$ would be automatically simplified back to $2x + 6$.

For such reasons, it is sometimes necessary to define more specific schemes to apply certain reasoning steps. These could include namely integers decomposition into products of prime factors ($36 = 2 \times 2 \times 3 \times 3$) or specifying to Sympy not to simplify the expressions, etc. We could as well use the opportunity to optimize the mappings generation or other parts of the step application.

The current version of the feedback generator thus defines specific schemes for steps:

$$AB + AC \to A(B + AC)$$
$$\frac{AB}{AC} \to \frac{AB}{C}$$
$$\frac{AB}{AC} \to \frac{B}{AC}$$

5.1.6 Evaluation

Unfortunately, there is not really any way to evaluate the quality of the feedback generator else than putting it into practice. That is, get student's feedback on the quality of the received feedback, on how often the application is able to deliver a feedback when mistakes are made, or measure the use of each feedback to know if it is pertinent.

The only thing that we are able to measure without practice is the time required to generate an exercise, including the generation of its feedbacks. For that purpose, we can analyze the data on figures 5.4 and 5.5.

Graphs explanation

Each figure contains for respectively first degree equation and fraction simplification exercises a graph showing the mean generation time for three levels of exercise complexity. For each of those levels, it provides as well a boxplot detailing the distribution of the sampled data that led to those mean values.
Each mean graph associates error bars to its values.

To produce those data, we generated for each complexity level of each exercise type 5 batches of 20 exercises (thus 100 exercises in total). The mean is the mean value of the generation times of all the exercises and the error is computed as $\frac{\sigma(\mu_1,...,\mu_n)}{\sqrt{n}}$ with μ_i the mean value over batch b_i, $\sigma(\mu_1, ..., \mu_n)$ the standard deviation over μ_i and $n = 5$. The sample size is only of 100 exercises instead of 1000 for exercise generation evaluation. This is because exercise generation including feedback generation takes much more time than without feedback generation.

The complexity level definition for each exercise type is quite arbitrary. We define simple, mid-complex and complex exercises. These correspond respectively to what is estimated to be the most simple form of exercise that could be ordered, a more complex form but that is still likely to be ordered, and very complex forms to test the generator in more extreme cases and be able to identify potential limitations to it.

First degree equation exercises

Parameter	Simple exercise	Mid-complex exercise	Complex exercise
nb_param_range	[0, 0]	[0, 0]	[0, 0]
fraction_token_proba	0.25	0.75	0.75
complex_factor_proba	0.5	0.75	0.75
add_unknown_terms_range	[0, 0]	[1, 1]	[1, 2]
add_unknown_factors_range	[0, 0]	[0, 0]	[1, 1]

Fraction simplification exercises

Parameter	Simple exercise	Mid-complex exercise	Complex exercise
nb_param_range	[0, 0]	[1, 1]	[2, 2]
answer_num_complexity_range	[0, 0]	[0, 1]	[1, 2]
answer_denom_complexity_range	[0, 0]	[0, 1]	[1, 2]
nb_factor_range	[2, 2]	[2, 2]	[2, 2]
factor_complexity_range	[0, 0]	[1, 1]	[2, 2]

Graphs analysis

We can see on both graphs 5.4a and 5.5a that simple and mid-complex exercises generation times are in average below the time limit of 1 minute that was imposed for exercise generation. Let's however further analyze the results.

We can see on graphs 5.4b and 5.5b that generation time for simple exercises always have been far below the threshold and we can conclude that the generator's performance for such complexity is perfectly acceptable.
On the other hand, we can observe on graphs 5.4c and 5.5c that there are some mid-complex exercises whose generation time exceeded the limit. We can see however that these cases are a clearly a minority and do not exceed the limit by too far anyway. As teachers will most probably order bunches of several tens of exercises, there is no risk for the generation to take an unexpectedly long time.

Unfortunately, as can be seen on both figures, complex exercise generation times clearly exceed the time limit. In fact, generation time increases so fast with the exercise complexity that complex exercises as defined for both exercise types are not even that much complex. Even more complex exercises would just have taken too much time to generate.

This is due to the fact that generation time does unfortunately not depend only on the number of wrong answers that are considered. It also depends on the time to apply reasoning steps on the exercises which may increase very fast with the exercise's complexity.
As an example let's consider the reasoning step $AC + AB \rightarrow A(C + AB)$. Applying it to a sum expression requires to map AC and AB to the terms of that sum.

Let's consider now a fraction simplification exercise with :

Parameter	Value
nb_param_range	[2, 2]
answer_num_complexity_range	[1, 1]
answer_denom_complexity_range	[1, 1]
nb_factor_range	[3, 3]
factor_complexity_range	[1, 1]

that is, with 2 parameters, numerator and denominator complexities of 1 and 3 factors with complexities of 1.
In the worst case, we could then have each factor and numerator/denominator being a sum of 2 terms. Multiplying a sum of m terms with a sum of n terms yielding a sum of $m \times n$ terms, we could thus end up with $2^4 = 16$ terms both at numerator and denominator.
Mapping $AC + AB$ to such a sum means considering all the ways to distribute 16 elements in two boxes which amounts to more than 65 000 combinations.
Considering such numbers, we can easily understand the cause for such generation times.

We could consider several partial solutions:

- Further limit the maximum number of wrong answers to be considered.

- Some reasoning steps might be less pertinent for some exercise types than for others (ex: factorization for first degree equations). We could then restrict the set of reasoning steps to be applied in function of the exercise type.

- Not exploring the entirety of the combination possibilities when applying a reasoning step. However, that would mean finding and implement a way to restrict this exploration. This could be difficult, especially if selecting the combinations in term of pertinence instead of randomly proves to be necessary.

- Restrict the complexity of the exercises that can be ordered.

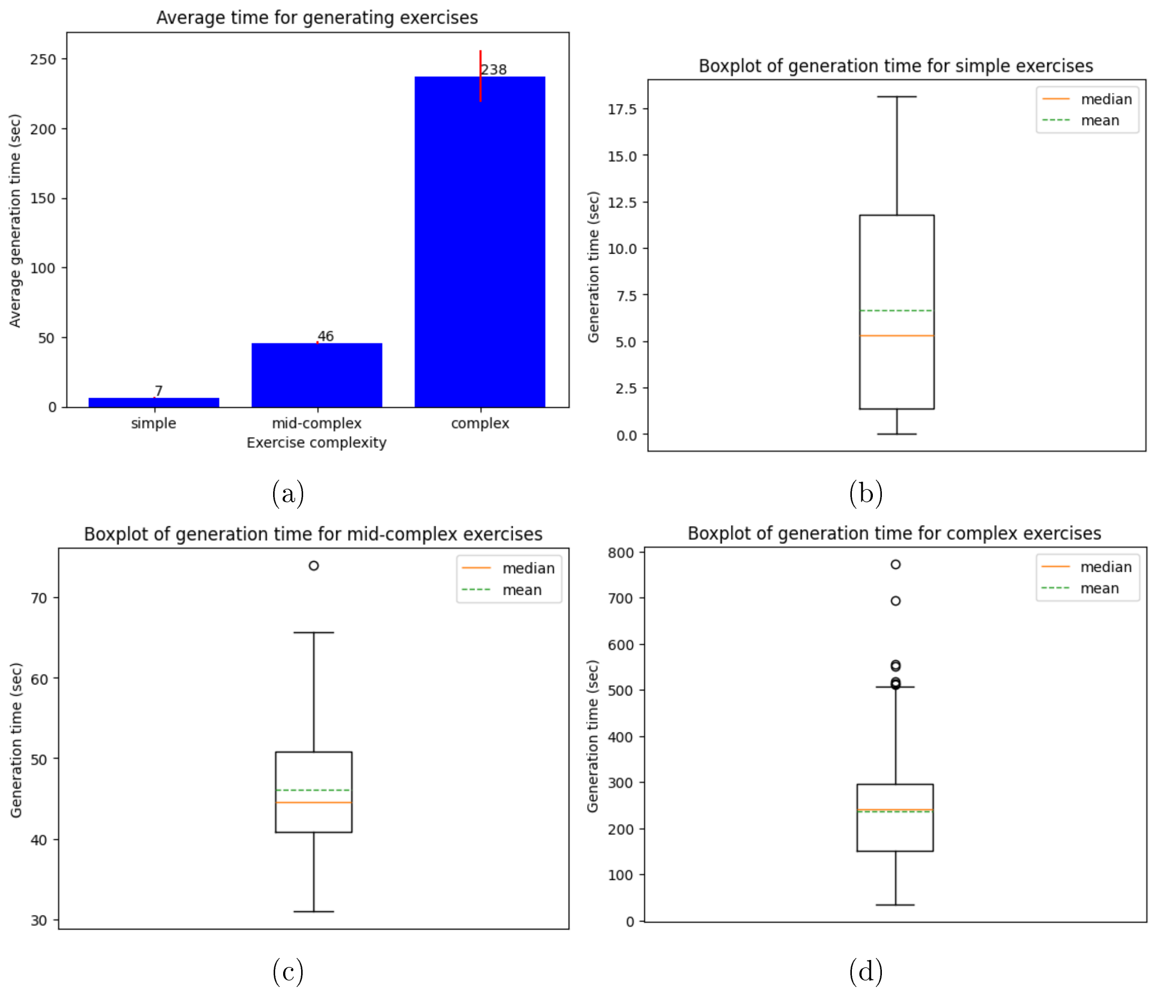


Figure 5.4: First degree equation exercise generation time for different exercise complexities

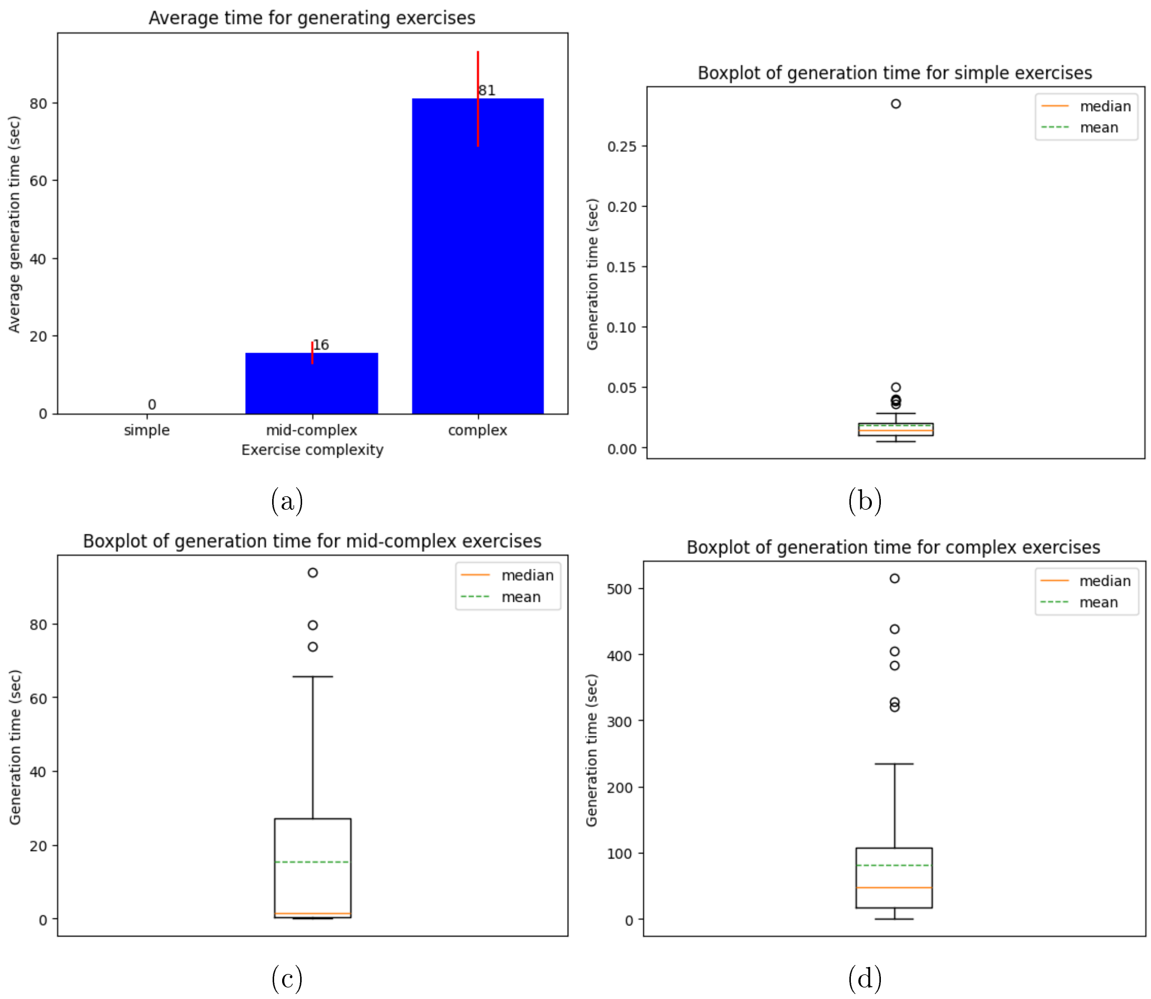


Figure 5.5: Fraction simplification exercise generation time for different exercise complexities

5.2 Synchronous feedback generation

In the case where the asynchronous feedback generator has been unable to predict a student's mistake, we want a synchronous feedback generator to be able to analyze the student's specific reasoning and take a last chance of delivering consistent feedback.

The algorithm is quite simple:
Given the student's reasoning step by step, check for each step that it leads to the correct answer (Would an exercise whose expression is equal to the considered step have the same solution than the initial exercise?).
As soon as a step leads to a different answer, we can conclude that the transition between this step and the previous one is erroneous.
Check all possible expressions resulting from all the ways to perform all the defined reasoning steps on the previous step (using algorithms defined in section 5.1.5). If one of these expressions has a corresponding answer equal to the one corresponding to the considered step, then the reasoning step that resulted in this expression is considered to be the cause of the error. The corresponding feedback is thus sent back along with the reasoning step's generic expression and the student's reasoning steps between which the mistake was introduced.

We don't check if other mistakes were introduced somewhere else in the student's reasoning as this is synchronous feedback generation. Indeed we don't want the server to preform too much computation and we want students to quickly receive their feedback.

5.3 Specific correctors

We have defined that the nodes of a reasoning tree are leaves if they correspond to the expression of the answer to the exercise.
We also defined a reasoning tree as bounded in depth.

The definitions of answer expression and the depth bound vary with the exercise type and we thus need to define specific feedback generators that manage those notions for each exercise type.

5.3.1 First degree equation corrector

In the case of first degree equations, an answer expression is defined as the unknown equaled to a simplified unknown independent expression. Simplification can be performed and checked by the Sympy function "*simplify()*".

$$\textbf{Answer :} x = 3$$

$$\textbf{Answer :} x = a + b$$

The depth bound is defined as the sum of the complexities of both side expressions of the equation.

5.3.2 Fraction simplification corrector

An answer expression for a fraction simplification exercise is simply a maximally simplified expression.

$$\textbf{Answer :} \frac{a+b}{3}$$

The depth bound is defined as equal to 2 times the number of factors that multiplied both the numerator and denominator of the exercise's answer during exercise generation.

5.4 Extensibility

We have seen that the part of a feedback generator that is specific to the exercise type is quite restrained. It should thus be relatively easy to modify it as to be able to support other types such as factorization exercises for example.
However there are still exercise types that would require major additions. As an example, second degree equation resolution involves going through three stages:

$$ax^2 + bx + c = 0$$
$$\rho = b^2 - 4ac$$

then set

$$x_{1,2} = \frac{-b \pm \sqrt{\rho}}{2a}$$

These steps do not directly follow one another. We cannot move from the homogeneous equation to the ρ calculation by a reasoning step as currently defined.

We have also seen that the reasoning step application scheme is as general as possible but yet is not compatible with every possible step.
One who wants to add new reasoning steps thus may be able to do it very easily but may as well have to define a specific application scheme. In any case, one should have a good understanding of the algorithms used to apply a step in order to be able to define new steps in an optimal way.

Chapter 6

Guide

This section is a guide explaining the file structure of the code and how to use it and extend/edit it.

6.1 File organization

6.1.1 Main files

The files that are essential to the project are:

- **generator.py :** Contains the exercise generators implementations.
- **corrector.py :** Contains the feedback generators implementations.
- **exercise.py :** Defines the *Exercise* class used to define exercises and associate their attributes.
- **reasoningTree.py :** Defines the *Tree* and *Node* classes used to build the reasoning tree during asynchronous feedback generation.
- **step.py :** Defines the *Step* class used to define reasoning steps and associate their attributes.
- **steps.xlsx :** An excel file in which reasoning steps are manually defined.
- **generatorConfig.json :** A json file defining the generator's parameters.

6.1.2 Secondary files

There are other files for testing, etc :

- **testGenerators.py :** Contains the code for the exercise generators evaluation.
- **testCorrectors.py :** Contains the code for the feedback generators evaluation.
- **testCorrectorsData.json :** Contains the data computed by *testCorrectors.py* (in case we want to rebuild the graphs without having to regenerate hundreds of exercises).
- **graphs :** A folder containing the graphs produced by test files.
- **example.py :** A small demo of the exercise and feedback generation.
- **firstDegEquExample.json** and **fracSimpExample.json :** Each contain a precomputed exercise (so that the user can try different answer on same exercise without having to regenerate it).

6.2 Running the program

6.2.1 Dependencies

Running the code requires to have the *pandas*, *numpy*, *sympy* and *matplotlib* (for graph generation) libraries installed. A makefile is provided to install these and can be executed by running the command "make install".

6.2.2 Generation demo

An *example.py* file is provided as a demo for exercise and feedback generation. The program can be launched with the command
./example.py <ex_type> [-generateNew] [-seed <seed>]
where <ex_type> can take 'fraction_simplification' or 'first_degree_equation' as a value and -generateNew and -seed <seed> are optional parameters. -generateNew indicates to generate a new exercise instead of loading the precomputed one (which will then be overwritten by the new exercise). -seed <seed> indicates the seed for the random generator if the user wants to specify it.

6.2.3 Generation testing

The program for testing the exercise generation can be launched with the command
./testGenerators.py <ex_type>
where <ex_type> can take 'fraction_simplification' or 'first_degree_equation' as a value.
The resulting graphs are located in respectively graphs/First_degree_equation and graphs/Fraction_simplification.

The program for testing the feedback generation can be launched with the command
./testGenerators.py [-graphOnly]
where -graphOnly is an optional parameter that specifies to only recompute the graphs based on the precomputed data instead of regenerating all the exercises.
The resulting graphs are located in respectively graphs/First_degree_equation and graphs/Corrector.

6.2.4 Integration to My Success Squared

In the context of this project being integrated into the My Success Squared project, the code is meant to be used in the following way :

First create an instance of the right corrector with
FracSimpCorrector(max_considered_answers, max_considered_errors) or
FirstDegEquCorrector(max_considered_answers, max_considered_errors) where
max_considered_answers and max_considered_errors are the bounds respectively to the number of considered wrong answer and the number of mistakes to be inserted in a reasoning. Their default values are respectively 500 and 3.

Then create an instance of the right generator with
FracSimpGenerator(config_file, corrector, config, seed) or
FirstDegEquGenerator(config_file, corrector, config, seed) where corrector is the corrector instance created in previous step, seed (optional) is the seed for the random generator, config_file (optional) is the opened json file containing the generator's configuration and config (optional) is the python dictionary containing the generator's configuration (either config_file or config should thus be provided).
This yields a generator capable of providing exercises with their feedback and corresponding to the provided configuration.

An exercise can be ordered along with its feedback with the generator's function *generateEx(seed)* where seed (optional) is the seed for the random generator. This function returns a json containing all the necessary data :

- **params :** The set of parameters appearing in the exercise.
- **is_equation :** Whether the exercise is an equation.
- **ex_expr :** The set of the exercise's expression(s) (2 if the exercise is an equation).
- **unknowns :** The set of unknowns for this exercise (for equations).
- **factors :** The set of factors that by which the answer was multiplied (if fraction simplification).
- **answers :** The set of answers to the exercise (second degree equation would have 2).
- **answers_feedbacks :** A dictionary associating feedbacks to the corresponding wrong answer.

Synchronous feedback can be ordered with the corrector's function *getFeedbacksFromReasoning(ex_json, reasoning)* where ex_json is the exercise's json (the one returned by *generateEx(seed)*) and reasoning is the set of steps in the student's reasoning.

A feedback is a set containing one element for each mistake in the reasoning in order of appearance.
Each one of those elements are then constituted of three elements:

- **generic error :** The generic expression of the mistake that has been made.
- **specific error :** The expressions of the steps of reasoning between which the mistake has been introduced (student reasoning if synchronous feedback).
- **error message :** The feedback message associated to the mistake.

For synchronous feedback, only the first mistake is considered.

6.3 Edition and Extension

Exercise generation parameters can be edited in generatorConfig.json and the corrector's parameters are to be specified at initialization.

Reasoning steps may be edited in steps.xlsx where each reasoning step is defined with the following components:

- **Name :** The step's identifier.
- **Weight :** The step's weight reflecting it's probability of occurrence (the higher, the less probable).
- **Is error :** Whether it is an erroneous step.
- **Is equation :** Whether the step applies to an equation or to a single expression.
- **Message :** The feedback message associated to the step.
- **Pattern :** The generic pattern associated to the step.

New steps may then be easily added. However, as discussed in section 5.1.5, depending on the step, more consequent editing on the corrector may be required.

No component (corrector, step or exercise classes) have been designed for one exercise type in particular. Things have been implemented in the most general way possible to facilitate addition of new types of exercises.

Chapter 7

Conclusion

This project was aimed at finding a solution to the problem of automatically generating exercises of different types and different complexities on one hand, and to the problem of providing pertinent and helpful feedback in case of mistakes on the other hand.

Although all the exercise types have not been covered in this project, tools that may be useful for any exercise type, a model allowing for tuning the exercises' complexities and complex but general algorithms for generating feedback for a given exercise have been provided.
We have ensured that the diversity of the generated exercise is satisfying and exercises' feedbacks are generated in a reasonable time as long as the exercises are not too complex.
A weakness however remains in the feedback generator as there is a trade-off between exercise complexity and time efficiency and thus a trade-off between exercise complexity and feedback quality (complex exercise means less feedback considered).

The work that has been accomplished in this project thus provides a good basis to which extensions may easily be brought to define new exercise or error types without having to modify the whole code or design the whole exercise/feedback generation process.

The code can easily be inserted into the rest of the My Success squared project as it works only with two simple api points.

Bibliography

[1] José João Almeida, Isabel Araújo, Irene Brito, Nuno Carvalho, Gaspar J Machado, Rui MS Pereira, and Georgi Smirnov. Math exercise generation and smart assessment. In *2013 8th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–6. IEEE, 2013.

[2] José João Almeida, Isabel Araújo, Irene Brito, Nuno Carvalho, Gaspar J. Machado, Rui M. S. Pereira, and Georgi Smirnov. Exercise generation with the system passarola. In *Proceedings of the 2013 International Conference on Advanced ICT and Education*, pages 305–309. Atlantis Press, 2013/08.

[3] Valentin Baum. Implementing an automatic pointers exercises generator in cafÉ 2.0. Unpublished master's thesis, Université de Liège, 2023. Available at https://matheo.uliege.be/handle/2268.2/18258.

[4] Paul Salvador Inventado, Peter Scupelli, Cristina Heffernan, and Neil Heffernan. Feedback design patterns for math online learning systems. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs*, pages 1–15, 2017.

[5] Ana Paula Tomás and José Paulo Leal. A clp-based tool for computer aided generation and solving of maths exercises. In *International Symposium on Practical Aspects of Declarative Languages*, pages 223–240. Springer, 2002.

[6] Ana Paula Tomás and José Paulo Leal. Automatic generation and delivery of multiple-choice math quizzes. In *International Conference on Principles and Practice of Constraint Programming*, pages 848–863. Springer, 2013.

[7] Ana Paula Tomás, José Carlos Santos, José Paulo Leal, Marcos Domingues, Miguel Filgueiras, Nelma Moreira, and Nuno Pereira. Agilmat–a web application for math education.