Feedback Systems for Students Solving Problems Related to Polynomials of Degree Two or Lower

Luke Adrian Gubbins Bayzid
Campus Universitário de Santiago, University of Aveiro, Portugal
Thematic Line GEOMETRIX, University of Aveiro, Portugal

Ana Maria Reis D'Azevedo Breda
Campus Universitário de Santiago, University of Aveiro, Portugal
Center for Research & Development in Mathematics and Applications, University of Aveiro, Portugal

Eugénio Alexandre Miguel Rocha
Campus Universitário de Santiago, University of Aveiro, Portugal
Center for Research & Development in Mathematics and Applications, University of Aveiro, Portugal

José Manuel Dos Santos Dos Santos [1]
Centre for Research and Innovation in Education (inED),
Escola Superior de Educação – Politechnic of Porto, Portugal

Abstract

In this paper, we present the first attempts to design and implement an algorithm that effectively responds to errors in a student's resolution in problems related to polynomials of degree two or lower. The algorithm analyzes the student's input by comparing pre-established resolution patterns. The obtained results of the implementation show that the algorithm is effective at the classes of problems created within the project's scope. Future work will concern the expansion of the number of classes to other common types of problems, such as higher-degree polynomials, and its use at a large scale using open-source software with CAS capabilities.

2012 ACM Subject Classification Mathematics of computing; Computing methodologies

Keywords and phrases Automatic feedback, Algorithms, Algebraic systems

Digital Object Identifier 10.4230/OASIcs.ICPEC.2022.5

Funding This research was supported by the Center for Research and Development in Mathematics and Applications (CIDMA) through the Portuguese Foundation for Science and Technology (FCT – Fundação para a Ciência e a Tecnologia), and within the scope of the research conducted by Thematic Line GEOMETRIX, references UIDB/04106/2020 and UIDP/04106/2020; The Centre for Research and Innovation in Education (inED), through the FCT – Fundação para a Ciência e a Tecnologia, I.P., under the scope of the project UIDB/05198/2020; and Organization of Ibero-American States for Education, Science and Culture (OEI).

1 Introduction

Feedback is crucial in the development of a student's ability to validly reason in any subject of study [2, 3, 1]. Consequently, the detection of mistakes made by students while solving a problem is of extreme importance, as it permits the personalization of feedback, highlighting aspects of the problem that the student should pay more attention to. Once a student knows which issues they should focus on, studying goes from a task about haphazardly improving at a set of problems into recognizing and formulating specific ideas associated to the material at hand, facilitating the identification and communication of issues.

[1] Corresponding author.

© Luke Adrian Gubbins Bayzid, Ana Maria Reis D'Azevedo Breda, Eugénio Alexandre Miguel Rocha, and José Manuel Dos Santos Dos Santos;
licensed under Creative Commons License CC-BY 4.0

Given the number of students assigned to a class, it can be difficult or even impossible for a teacher to provide each student with the advice needed to improve at the subject at hand, and this situation becomes even more complex for students with difficulties in expressing their doubts, some of them choosing to focus only on autonomous study as a consequence.

Unlike many other fields, mathematics deals with structured forms that have definite rules, allowing anyone, even computers, to modify previous structures into others. While generating interesting structures continues to be an issue, verifying them automatically is something that computers excel at.

In this article, we propose a general method to generate feedback based on the input of an ordered list of structured equations provided by a student, presenting a particular case of using this procedure for problems involving polynomials of degree two or lower.

2 Feedback

In general, *feedback* can be defined as a piece of information related to the previous results of a particular task that reflects what ought to be improved. For example, information detailing only the subject's achieved score would be considered as a form of feedback [7].

Given that definition, one aspect to focus on would be the quality of the information provided. In particular, the information that we decided to tackle in this article concerns only the quality of the result or the method used by the student; this restriction was made to ensure the feasibility of the project within its intended scope. An example of a type of feedback that falls outside of those two categories would be recommendations regarding the mindsets that a student should consider while studying.

With regard to education, feedback seeks to increase the long-term score of a student's ability to perform the tasks associated to the subject being taught. From various experiments performed over the years using a variety of methods, the positive effects of feedback within education have been empirically validated with a high degree of statistical certainty [8]. However, different categories of feedback and subsets thereof have also proven to be more influential than others [5, 6], necessitating a more particular analysis of the chosen categories for the project.

Using our categorizations of feedback, we were interested in knowing how much feedback regarding the student's methodology would benefit them over merely providing an aptitude score. In accordance with relevant research [4], while the results aren't drastic enough to cause a sudden shift in the education system, the measurable increase in providing both types of feedback was sufficient enough for us to deem it as a worthwhile endeavor.

Though effective feedback can take many forms [2], only the lesser of them tend to get implemented into software that's meant to support the education of students. In particular, one commonly observed detail about many of those programs is that they can only discern whether or not an answer is correct; some of them might also provide a solution written out by a professor. While having a curated solution can definitely help students struggling to answer questions aptly, it might not help them with understanding the subject at hand, only to memorize a list of steps that lead to a correct answer. Though a proper understanding is not the goal of all students, having such might help them generalize what they're taught, improving performance in similar kinds of questions over a variety of circumstances.

3 Motivation and Problem

The difficulty in solving this particular kind of problem primarily stems from automating the process of identifying a mistake and appropriately associating it to its respective feedback. While the former problem can be solved by using a sufficiently powerful CAS, the fact that a

problem can be solved through a variety of means and with varying amounts of detail makes the latter problem a bit less trivial to solve. For example, the two mistaken resolutions $x + 1 = 0 \iff x = 1$ and $x + 1 = 0 \iff x + 1 - 1 = 0 + 1 \iff x = 1$ have different lengths and vary in detail but still display the misuse of signs being the source of the issue. As such, the feedback returned by the algorithm should be invariant to such changes.

Our motivation to implement a system based on axiom schemata and tree substitution came from the necessity to generate structured data using simple rules. Originally, we believed that it would be too computationally inefficient, but further tests showed that a naive implementation would suffice for the sample of rules necessary to represent a subset of the types of mistakes performed by a student in the given task. Later on, different ideas for optimization were considered but deemed unnecessary for the algorithm's first implementation; more research will be made in that regard.

Given the initial goal of managing polynomials of degree two or lower, a previous attempt used the roots of the provided polynomial to construct a canonical representation by expanding its factored form and dividing all coefficients by the value of the leading coefficient. From there, comparisons would be made between that representation and the one of the step at which the student made a mistake. However, due to its lack of generality beyond polynomials, this algorithm was dropped in favor of the one detailed in this article.

An idea that came from that previous attempt was to automatically encode every step in a canonical representation; in particular, the canonical representations would be constructed in a way that would facilitate the detection of mistaken symbols, such as inserting a minus instead of a plus. This idea was also abandoned due to the lack of a general pattern regarding what might facilitate the detection of mistakes for more complicated kinds of equations.

4 Algorithm

4.1 Overview

The algorithm described in this article consists in four steps: parsing and cleaning text, validating the equality of the solution sets of adjacent steps, generating paths that match the student's mistake, and providing adequate feedback. Following the presented order, each of the following paragraphs will detail the overview of the process.

The parsing algorithm that we implemented consists of a collection of metrics that decide whether or not to include a character of the input based on surrounding characters and if it belongs to a curated set of expected characters. After that, a structured tree is formed using the order of operations that are either implicit or explicit through the use of parentheses and common syntax.

The goal of a validation system is to ensure that each step is entailed from those that came before it. A property that facilitates this goal is that equations can be marked as equivalent if they have equal solution sets. Knowing that, our method uses a CAS to search for the first step in which a mistake was made by testing the equality of the solution sets of each ordered pair of consecutive steps. For a formalization thereof, see Algorithm 1; it should be noted that Ω is a function that returns the solution set associated to a step and that *Steps* is a tuple representing each equation written down by the student in order.

Once the first two equations in which a mistake happened have been identified, the system will take the first of them and generate all of the possible expressions from it using the supplied patterns; it will repeat that process for a finite number of steps. Finally, it will aggregate all of the paths with solution sets that are equal to the solution set of the mistaken step.

Algorithm 1 Determine First Erroneous Step.

1: **for** $i = 1, \ldots, |Steps| - 1$ **do**
2: **if** $\Omega(Steps_i) \neq \Omega(Steps_{i+1})$ **then**
3: return $(Steps_i, Steps_{i+1}, i + 1)$
4: **end if**
5: **end for**
6: **return** ()

Given that each pattern used in the previous step has an associated flag, the final aspect of the algorithm is to return the feedback associated to each of the generated paths in natural language.

Figure 1 shows an overview of the algorithm's logic.

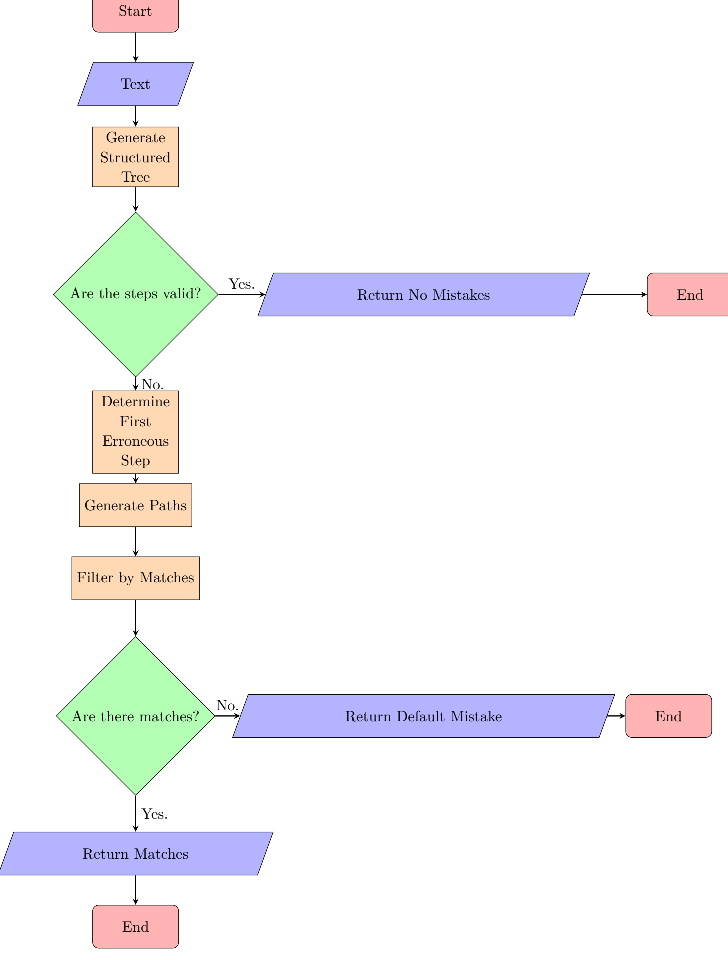

Figure 1 Flowchart of the Algorithm's Logic.

4.2 Patterns

Within our implementation, expressions are represented as trees with subtrees that are determined by the order of operations presented within the mathematical expression. In the case of polynomials, only functions of arity two or lower need to be considered, so each node in a polynomial expression need only have two or three children, where those children can be strings or other trees.

Each pattern consists of an axiom schema that reflects a particular deformation that can be applied to an expression. Our implementation represents axiom schemata as pairs of trees consisted only of constants, schematic variables, and subtrees. Constants are strings that must be exactly matched to have the pattern apply, whereas schematic variables are strings that are uniquely denoted using a dollar symbol before their declaration and can be substituted by a constant or a tree; subtrees are used to encode the order of operations within an expression. For example, $((''\$1'', ''+'', ''\$2''), (''\$2'', ''+'', ''\$1''))$ would represent the commutativity of addition, and $((''\$1'', ''*'', (''\$2'', ''+'', ''\$3'')), ((''\$1'', ''*'', ''\$2''), ''+'', (''\$1'', ''*'', ''\$3'')))$ would represent the distributive property of multiplication over addition.

4.3 Applying Patterns

Similar to many previous approaches regarding applying rules to structured data, using an ordered list of strings denoting schematic variables, constants, and other lists of the same type to represent a tree, applying a pattern merely requires verifying several conditions before performing a substitution.

A necessary property to apply patterns is to detect when they can be applied. This can be achieved by having the tree representing the pattern be a generalization of the structure that we want to deform, which primarily pertains to the order of its subtrees and schematic variables. We say that X is a generalization of Y, which we denote as $X \geq Y$, if and only if

$$|X| = |Y| \land \forall i \in [|X|], X_i \geq Y_i \lor (X_i \in V_X \land Y_i \notin E),$$

where V_X is the set of schematic variables of X and of all of the subtrees thereof; should X be a string, V_X will return a set containing only X if and only if X is a schematic variable, returning an empty set otherwise. Another aspect of importance is that if X and Y are strings, then $X \geq Y$ if and only if X is a schematic variable or syntactically equal to Y. Finally, E represents the set of strings that cannot be considered for substitutions; our implementation disregarded operators, functions, and predicates.

Let S be the set of all strings. Given that $X \geq Y$, the next step is to generate a set containing all of the differences between them. This can be done by defining

$$\Delta(X,Y) = \begin{cases} \{(X,Y)\} & X \in S \land X \neq Y \\ \bigcup_{i \in [|X|]} \Delta(X_i, Y_i) & X \notin S \land X \neq Y \\ \emptyset & X = Y \end{cases}.$$

We say that a set of differences is consistent if and only if it can be constructed into a function. In other words, $\forall x, y \in \Delta(X,Y), x_1 = y_1 \implies x_2 = y_2$, which we represent as $\phi(\Delta(X,Y))$.

Under the assumption that it's consistent, we need a function to apply the set of differences provided by Δ to a particular tree. Denoting it by α, we can define it as

$$\alpha(X, \Delta(X,Y)) = \begin{cases} \Delta(X,Y)_{i,2} & \exists i \in [|\Delta(X,Y)|], X = \Delta(X,Y)_{i,1} \\ X & \forall i \in [|\Delta(X,Y)|], X \neq \Delta(X,Y)_{i,1} \land X \in S \\ (\alpha(X_1, \Delta(X,Y)), \ldots, \alpha(X_{|X|}, \Delta(X,Y))) & \forall i \in [|\Delta(X,Y)|], X \neq \Delta(X,Y)_{i,1} \land X \notin S \end{cases}.$$

Having defined all of the above, the substitution function can, therefore, be defined as

$$\sigma(X,(Y,Y')) = \begin{cases} \alpha(Y', \Delta(Y,X)) & Y \geq X \wedge \phi(\Delta(Y,X)) \\ X & Y \ngeq X \vee \neg\phi(\Delta(Y,X)) \end{cases},$$

where (Y,Y') represents a pattern.

4.4 Generating Paths

A path consists of an ordered list of trees. Given a particular axiom schema, our implementation generates a path by starting from the top node, creating one tree where the axiom schema is not applied and one where it is applied, and applying this algorithm recursively to each of the possibilities generated by the previous step. Finally, once all of the axiom schemata have been applied, all of the generated paths are collected and taken into consideration for the next application of the same set of axiom schemata, and this process repeats itself for a number of steps chosen by the user. More precisely, letting

$$\Sigma(X,(Y,Y')) = \begin{cases} \{X, \sigma(X,(Y,Y'))\} \cup \bigcup_{i \in [|X|]} \{(\ldots, X_{i-1}, x, X_{i+1}, \ldots) \mid x \in \Sigma(X_i,(Y,Y'))\} & X \notin S \\ \{\sigma(X,(Y,Y'))\} & X \in S \end{cases},$$

we can generate all possibilities of the application of a pattern to a particular tree. From that, denoting A as the last step before the first mistake, we can start with an initial set T_0, where $T_0 = \{(A)\}$. As such, to generate the paths, we need only state that

$$T_i = \bigcup_{((x_1,\ldots,x_m),Y) \in T_{i-1} \times P} \{(x_1, \ldots, x_m, y) \mid y \in \Sigma(x_m, Y)\},$$

where P represents the set of all specified patterns for the algorithm. This process of iterating T_i happens until i reaches a desired value.

For an overview of this section in pseudocode, see Algorithm 2.

Algorithm 2 Generate Paths.

```
1:  Paths ← {(Steps_k, ())}
2:  for i = 1, ..., n do
3:      Copy ← Paths
4:      for ((x_1, p_1), ..., (x_i, p_i)) ∈ Copy do
5:          for j ∈ Patterns do
6:              Paths ← Paths ∪ {((x_1, p_1), ..., (x_i, p_i), (x, j)) | x ∈ Σ(x_i, j)}
7:          end for
8:      end for
9:  end for
10: return Paths
```

4.5 Returning Feedback

Once all paths have been generated, the algorithm picks only the ones whose final step has a solution set that's equal to the solution set of the student's mistaken step.

It is important to note that since only the solution sets need to be equal, which is not as strict as requiring syntactic equality between a generated step and the subsequent step, the algorithm is robust against not having all steps provided or a general lack of detail. While

this might come at the cost of potentially returning irrelevant feedback when the algorithm is equipped with patterns that conditionally return the same solution sets without having the same sort of feedback associated, we believe that certain heuristics or statistical models could be used to manage conflicts of that sort by ranking each piece of returned feedback. To that end, under the assumption that the probability that a student will perform many mistakes is low, we chose to implement a heuristic that sorts the list of matches by the number of steps associated to each match in ascending order, putting at the forefront the matches that are considered more common. While many other heuristics were considered to rank the matches, they were not a part of what was researched.

After the matches have been filtered, a function that maps each used pattern to its associated feedback is used. For a complete picture of the algorithm, see Algorithm 3.

Algorithm 3 Feedback Algorithm.

1: $Error \leftarrow Determine\ First\ Erroneous\ Step(Steps)$
2: **if** $Error \neq ()$ **then**
3: $Paths \leftarrow \{p \in Generate\ Paths(Error_1, Patterns) \mid \Omega(p_{|p|,1}) = \Omega(Error_2)\}$
4: **if** $|Paths| > 0$ **then**
5: **return** $\{(Feedback(p_1), \ldots, Feedback(p_n), Error_3) \mid ((x_1, p_1), \ldots, (x_n, p_n)) \in Paths\}$
6: **else**
7: **return** $\{(Error_3)\}$
8: **end if**
9: **else**
10: **return** $\emptyset$
11: **end if**

4.6 Complexity

One aspect of concern within this project regarding its generality is how the computational complexity of algorithm grows with an increase in the number of patterns and the maximum path length. In particular, this information can help make guided decisions on how the algorithm might be made more efficient in future works.

While the complexity varies significantly with the patterns and input being used, an approximation for the upper bound can trivially be found. Under the assumption that n patterns that are different from the identity pattern and result in k expressions can be applied during each step, the algorithm can generate $(n * k + 1)^m$ paths at most from a single expression, where m represents the maximum path length; the addition of unity comes as a consequence of the necessity of at least one pattern being the identity pattern. As such, an increase in the number of patterns causes a polynomial growth in complexity, whereas the algorithm's complexity scales exponentially with the maximum path length.

Taking the approximation for an upper bound of the algorithm's complexity into account, we can conclude that it is more efficient to add more patterns and reduce the maximum path length. However, while decreasing the maximum path length and increasing the number of patterns, care must be taken not to reduce the algorithm's ability to properly discern the necessary information to provide feedback.

Finally, given that each pattern can be applied independently from the others during each step, parallelization can be implemented trivially, opening avenues for hardware that supports it. While implementing it might not be prove to be too beneficial for a large maximum path length, it might be helpful for time-sensitive tasks, such as a server hosting an implementation of this algorithm responding to requests from clients.

5 Preliminary Results

5.1 Example

Before ascertaining the results, let's manually perform the algorithm. Using only $("-", "+")$ as a pattern and a maximum path length of one, we'll run through an informal example of how the algorithm ought to be performed.

Given that $x-1=0 \iff x=-1$ is the student's input, let's attempt to provide feedback. First of all, we need to find the first pair of adjacent steps whose solution sets are different; since $\{1\} \neq \{-1\}$, we've found a mistake on the second step. Secondly, each equation within the mistaken step provided must be formatted as a tree, leading $x-1=0$ and $x=-1$ to be formatted as $(("x", "-", "1"), "=", "0")$ and $("x", "=", ("-", "1"))$ respectively. Next, we need to exhaustively apply our patterns to the first of those expressions:

$\Sigma((("x", "-", "1"), "=", "0"), ("-", "+")) =$

$\{(("x", "-", "1"), "=", "0"), \sigma((("x", "-", "1"), "=", "0"), ("-", "+"))\} \cup \{(x, "=", "0") \mid x \in \Sigma(("x", "-", "1"), ("-", "+"))\} \cup \{(("x", "-", "1"), x, "0") \mid x \in \Sigma("=", ("-", "+"))\} \cup \{(("x", "-", "1"), "=", x) \mid x \in \Sigma("0", ("-", "+"))\} =$

$\{(("x", "-", "1"), "=", "0")\} \cup \{(x, "=", "0") \mid x \in \Sigma(("x", "-", "1"), ("-", "+"))\} =$

$\{(("x", "-", "1"), "=", "0")\} \cup \{(x, "=", "0") \mid x \in \{("x", "-", "1"), ("x", "+", "1")\}\} =$

$\{(("x", "-", "1"), "=", "0"), (("x", "+", "1"), "=", "0")\}.$

Finally, by checking the solution sets of our acquired paths, we can conclude that $(("x", "+", "1"), "=", "0")$ has the same solution set as $("x", "=", ("-", "1"))$; as such, since what led to that path was changing a sign, we may return feedback stating that the student erred on the second step by mistaking a sign.

5.2 Results From a Python Implementation

To perform a preliminary test of the algorithm's ability to provide meaningful feedback, twenty representative examples were chosen and tested with an implementation of this algorithm in Python using SymPy. These examples were manually sampled from what we considered to be common mistakes, such as mistaking signs or factoring incorrectly; our future works will go over results aggregated on a larger scale and under more realistic conditions.

The following is a list of the patterns used for the examples:

$("+", "-")$, $("-", "+")$,
$((("\$1", "+", "\$2"), "**", "2"), (("\$1", "**", "2"), "+", ("\$2", "**", "2")))$,
$((("\$1", "-", "\$2"), "**", "2"), (("\$1", "**", "2"), "-", ("\$2", "**", "2")))$,
$((("\$1", "**", "2"), "+", ("\$2", "**", "2")), (("\$1", "+", "\$2"), "**", "2"))$, and
$((("\$1", "**", "2"), "-", ("\$2", "**", "2")), (("\$1", "-", "\$2"), "**", "2"))$.

Before interpreting the results, a metric of success ought to be defined. Given the easy way of finding the first step in which a mistake occurred, we consider a result to be successful if and only if one of the found matches returns feedback that would've prevented the mistake. That being said, even if it can only detect which step was mistaken, the algorithm might still prove to be useful to many students.

Given that definition, despite the good results with the limited patterns used, more of them might need to be added to correctly detect certain categories of mistakes. For example, given $49 - x^2 = 0 \iff (7-x)^2 = 0$ as the input, the equipped patterns would not be sufficient; however, the addition of more patterns, such as $("\$1", (("\$1", "**", ("1", "/", "2")), "**", "2"))$, would permit the algorithm to respond aptly.

Table 1 Results From a Python Implementation of the Misuse of Signs.

<table>
<tr><th>Input</th><th>Erroneous Step</th><th>Output</th><th>Expected Output</th></tr>
<tr><td>((”x”,”−”,”1”),”=”,”0”),
(”x”,”=”,(”−”,”1”))</td><td>Second Step</td><td>Sign Error</td><td>Sign Error</td></tr>
<tr><td>((”2”,”+”,”x”),”=”,”0”),
(”x”,”=”,”2”)</td><td>Second Step</td><td>Sign Error</td><td>Sign Error</td></tr>
<tr><td>((”−”,”10”),”=”,(”5”,”*”,”x”)),
(((”5”,”*”,”x”),”−”,”10”),”=”,”0”),
((”5”,”*”,”x”),”=”,”10”),
(”x”,”=”,”2”)</td><td>Second Step</td><td>Sign Error</td><td>Sign Error</td></tr>
<tr><td>(((”7”,”*”,”x”),”−”,”8”),”=”,”0”),
((”7”,”*”,”x”),”=”,(”−”,”8”)),
(”x”,”=”,(”−”,(”8”,”/”,”7”)))</td><td>Second Step</td><td>Sign Error</td><td>Sign Error</td></tr>
<tr><td>((”x”,”−”,(”x”,”**”,”2”)),”=”,”0”),
((”x”,”**”,”2”),”=”,(”−”,”x”))</td><td>Second Step</td><td>Sign Error</td><td>Sign Error</td></tr>
<tr><td>(((”5”,”*”,(”x”,”**”,”2”)),”+”,(”−”,(”10”,”*”,”x”))),”=”,”0”),
(((”x”,”**”,”2”),”+”,(”−”,(”2”,”*”,”x”))),”=”,”0”),
((”x”,”**”,”2”),”=”,(”−”,(”2”,”*”,”x”)))</td><td>Third Step</td><td>Sign Error</td><td>Sign Error</td></tr>
<tr><td>(((”5”,”*”,”x”),”+”,”1”),”=”,”0”),
((”5”,”*”,”x”),”=”,”1”),
(”x”,”=”,(”1”,”/”,”5”))</td><td>Second Step</td><td>Sign Error</td><td>Sign Error</td></tr>
<tr><td>((((”2”,”*”,(”x”,”**”,”2”)),”−”,”x”),”+”,”1”),”=”,”0”),
((”2”,”*”,(”x”,”**”,”2”)),”=”,(”1”,”−”,”x”)),
((”x”,”=”,(”−”,”1”)),”|”,(”x”,”=”,(”1”,”/”,”2”)))</td><td>Second Step</td><td>Sign Error</td><td>Sign Error</td></tr>
<tr><td>(((”x”,”**”,”2”),”+”,”5”),”=”,”0”),
((”x”,”**”,”2”),”=”,”5”),
((”x”,”=”,(”−”,(”5”,”**”,(”1”,”/”,”2”)))),”|”,(”x”,”=”,(”5”,”**”,(”1”,”/”,”2”))))</td><td>Second Step</td><td>Sign Error</td><td>Sign Error</td></tr>
</table>

Table 2 Results From a Python Implementation of the Misuse of Factoring.

<table>
<tr><th>Input</th><th>Erroneous Step</th><th>Output</th><th>Expected Output</th></tr>
<tr><td>(((”x”,”−”,”1”),”**”,”2”),”=”,”0”),
(((”x”,”**”,”2”),”−”,(”1”,”**”,”2”)),”=”,”0”),
((”x”,”=”,(”−”,”1”)),”|”,(”x”,”=”,”1”))</td><td>Second Step</td><td>Factoring Error</td><td>Factoring Error</td></tr>
<tr><td>(((”x”,”**”,”2”),”+”,((”x”,”−”,”10”),”**”,”2”)),”=”,”0”),
(((”2”,”*”,(”x”,”**”,”2”)),”−”,”100”),”=”,”0”),
((”x”,”=”,(”−”,(”50”,”**”,(”1”,”/”,”2”)))),”|”,(”x”,”=”,(”50”,”**”,(”1”,”/”,”2”))))</td><td>Second Step</td><td>Factoring Error</td><td>Factoring Error</td></tr>
<tr><td>((”x”,”**”,”2”),”=”,(”−”,((”x”,”−”,”1”),”**”,”2”))),
((”x”,”**”,”2”),”=”,(”1”,”−”,(”x”,”**”,”2”))),
(”x”,”=”,((”1”,”/”,”2”),”**”,(”1”,”/”,”2”)))</td><td>Second Step</td><td>Factoring Error</td><td>Factoring Error</td></tr>
<tr><td>((((”x”,”+”,”1”),”**”,”2”),”+”,(”−”,(”2”,”*”,”x”))),”=”,”0”),
((((”x”,”**”,”2”),”+”,”1”),”+”,(”−”,(”2”,”*”,”x”))),”=”,”0”),
(”x”,”=”,”1”)</td><td>Second Step</td><td>Factoring Error</td><td>Factoring Error</td></tr>
<tr><td>(((”x”,”**”,”2”),”+”,(”1”,”**”,”2”)),”=”,”0”),
(((”x”,”+”,”1”),”**”,”2”),”=”,”0”),
(”x”,”=”,(”−”,”1”))</td><td>Second Step</td><td>Factoring Error</td><td>Factoring Error</td></tr>
<tr><td>(((”x”,”−”,”5”),”**”,”2”),”=”,”0”),
(((”x”,”**”,”2”),”−”,(”5”,”**”,”2”)),”=”,”0”),
((”x”,”=”,(”−”,”5”)),”|”,(”x”,”=”,”5”))</td><td>Second Step</td><td>Factoring Error</td><td>Factoring Error</td></tr>
<tr><td>((((”x”,”−”,”6”),”**”,”2”),”+”,(”2”,”*”,(”x”,”**”,”2”))),”=”,”0”),
(((”3”,”*”,(”x”,”**”,”2”)),”−”,”36”),”=”,”0”),
((”x”,”=”,(”−”,(”12”,”**”,(”1”,”/”,”2”)))),”|”,(”x”,”=”,(”12”,”**”,(”1”,”/”,”2”))))</td><td>Second Step</td><td>Factoring Error</td><td>Factoring Error</td></tr>
<tr><td>((”7”,”*”,(”x”,”**”,”2”)),”=”,(”−”,((”x”,”−”,”2”),”**”,”2”))),
((”7”,”*”,(”x”,”**”,”2”)),”=”,(”4”,”−”,(”x”,”**”,”2”))),
(”x”,”=”,((”1”,”/”,”2”),”**”,(”1”,”/”,”2”)))</td><td>Second Step</td><td>Factoring Error</td><td>Factoring Error</td></tr>
<tr><td>((((”5”,”−”,”x”),”**”,”2”),”+”,(”−”,(”24”,”*”,”x”))),”=”,”0”),
(((”25”,”−”,(”x”,”**”,”2”)),”+”,(”−”,(”24”,”*”,”x”))),”=”,”0”),
((”x”,”=”,(”−”,”25”)),”|”,(”x”,”=”,”1”))</td><td>Second Step</td><td>Factoring Error</td><td>Factoring Error</td></tr>
<tr><td>((”49”,”−”,(”x”,”**”,”2”)),”=”,”0”),
(((”7”,”**”,”2”),”−”,(”x”,”**”,”2”)),”=”,”0”),
(((”7”,”−”,”x”),”**”,”2”),”=”,”0”),
((”7”,”−”,”x”),”=”,”0”),
(”x”,”=”,”7”)</td><td>Third Step</td><td>Factoring Error</td><td>Factoring Error</td></tr>
</table>

Another issue with this approach is that it requires the solution sets of all of the expressions to be computable by the CAS. Given the task of solving it for polynomials of degree two or lower, this was not a problem, but it is possible that complications may arise from generalizations of this algorithm.

6 Future Work

Given what was discussed in the section about the algorithm's complexity, one avenue of research is finding which patterns should be used to reduce the number of steps needed to provide apt feedback. In particular, the plan is to develop a method of automatically generating useful patterns and manually label them with appropriate feedback; currently, we plan to research how machine learning could be used to do such.

Beyond that, as mentioned in our section about preliminary results, experiments will be performed on a larger scale with open-source software with CAS capabilities. The hope is to be able to find software that can serve as a front-end interface for the presented algorithm and test its capabilities on a larger scale and with a greater variety of inputs and patterns. Should the results show promise, other avenues of improvement will be researched.

References

1. Anderson Pinheiro Cavalcanti, Arthur Barbosa, Ruan Carvalho, Fred Freitas, Yi-Shan Tsai, Dragan Gašević, and Rafael Ferreira Mello. Automatic feedback in online learning environments: A systematic literature review. *Computers and Education: Artificial Intelligence*, 2:100027, 2021. doi:10.1016/j.caeai.2021.100027.
2. John Hattie and Helen Timperley. The power of feedback. *Review of Educational Research*, 77(1):81–112, 2007. doi:10.3102/003465430298487.
3. Mark Jellicoe and Alex Forsythe. The development and validation of the feedback in learning scale (fls). *Frontiers in Education*, 4, 2019. doi:10.3389/feduc.2019.00084.
4. Raymond W. Kulhavy and William A. Stock. Feedback in written instruction: The place of response certitude. *Educational Psychology Review*, 1(4):279–308, 1989. doi:10.1007/BF01320096.
5. Susanne Narciss. Feedback strategies for interactive learning tasks. *Handbook of research on educational communications and technology*, 3:125–144, 2008. URL: https://www.routledgehandbooks.com/doi/10.4324/9780203880869.ch11.
6. Susanne Narciss, Elsa Hammer, Gregor Damnik, Kerstin Kisielski, and Hermann Körndle. Promoting prospective teacher competencies for designing, implementing, evaluating, and adapting interactive formative feedback strategies. *Psychology Learning & Teaching*, 20(2):261–278, 2021. doi:10.1177/1475725720971887.
7. Ernesto Panadero and Anastasiya A. Lipnevich. A review of feedback models and typologies: Towards an integrative model of feedback elements. *Educational Research Review*, 35:100416, 2022. doi:10.1016/j.edurev.2021.100416.
8. Scott A. Schartel. Giving feedback – an integral part of education. *Best Practice & Research Clinical Anaesthesiology*, 26(1):77–87, 2012. Challenges in Anaesthesia Education. doi:10.1016/j.bpa.2012.02.003.