Natural Deduction Proofs for Educational Feedback

Daniel Macau[1], Ricardo Gonçalves[1], and João Costa Seco[1]

NOVA School of Science and Technology, Caparica, Portugal

Abstract. Online tools, where students can practice and have automatic feedback, have shown to be useful both for MOOCS or as complementary material for traditional classes. Nevertheless, contrary to the myriad of tools that exist for learning programming languages, in the area of logic, a fundamental subject in any Computer Science programme, there is still a lack of such online tools, and in particular for the challenging exercise of Natural Deduction (ND) proofs. The few existing tools usually do not provide effective feedback, which is in fact challenging since tree-like ND proofs are non-linear and some steps are not immediate, as it is the case of proofs by contradiction. In this paper, we present an algorithm for pedagogical purposes that can generate human-readable ND proofs for a given problem. The algorithm supports both Propositional and First-Order Logic, and is based on hypergraphs. This allows obtaining the shortest and most direct proof for a given problem, but it can also adapt to a user's current unfinished proof, allowing for greater flexibility in the proof construction. The algorithm can be integrated with educational platforms to guide users through the proofs by providing advanced feedback, at the same as it can help teachers grading ND exercises.

Keywords: Natural Deduction · Propositional Logic · First Order Logic · Automation · Algorithm · Feedback · Grading

1 Introduction

Learning logic is a fundamental component of today's curriculum, as it plays a key role in several important areas, including programming languages, databases, artificial intelligence, and algorithms [4]. Logic courses cover a wide range of topics, and one that stands out is ND, given its importance in helping students develop reasoning skills [1], which are valuable in real-world contexts where structured thinking and argumentation are required [11].

Natural deduction exercises are generally considered among the most challenging for students due to the complex reasoning involved. Since mastering them requires extensive exposure, many students struggle to become familiar with the rules and the formal way of thinking. These exercises involve numerous logical rules, and it is not always clear which one to apply. Additionally, students often need to keep track of multiple steps and assumptions simultaneously, which can be confusing.

Despite the importance of logic education, there remains a lack of online tools that support this type of exercise. The few existing tools typically offer very limited feedback, focusing mainly on syntactic and semantic errors while failing to provide deeper guidance that could help students overcome conceptual difficulties [9]. A major challenge students face when solving ND problems is becoming stuck either because they reach an impasse or feel they are overcomplicating the exercise.

This highlights the need for more advanced and effective feedback systems to address these gaps. One approach to producing this type of feedback is through algorithms capable of generating complete proofs. This allows the system to derive hints from the generated proof, ensuring that the guidance leads to a correct solution rather than a dead end. However, most existing algorithms were not developed for this purpose: while they can find a solution or detect contradictions, they often lack flexibility in terms of rule application, depend on the specific ND system used, and cannot dynamically adapt to a user's reasoning process. Capturing the user's reasoning enables the feedback to be personalized and aligned with the student's chosen path.

In this article, we present an algorithm developed during a thesis project with a pedagogical focus. It was originally created to support an application that helps students build and verify ND proofs. However, the application itself is not addressed in this article. The algorithm is capable of automatically generating multiple Gentzen-style proofs for the same problem in a human-readable way, both in PL and FOL. A distinguishing feature of our algorithm is its ability to adapt to a user's solution, providing step-by-step guidance that aligns with the student's reasoning process. This is made possible through the use of directed hypergraphs that store information about which rules can be applied at each step, capturing multiple valid proof paths. By leveraging well-known graph search and traversal algorithms, the system not only finds correct solutions but also identifies the shortest proofs, enabling feedback on how a solution could be improved. Additionally, the algorithm can be used to assess exercises by determining how far a student's resolution is from a valid or optimal solution, thereby offering a measurable assessment of proofs.

2 Background

Proof systems are fundamental in logic education because they provide a structured framework for understanding and constructing logical arguments, ensuring rigor and validity in mathematical reasoning. Among the many proof systems, one usually adopted in the logic courses is the Natural Deduction (ND) system, which was defined to more closely align with mathematical and everyday reasoning under assumptions [7]. It emerged in 1934 with Gentzen and Jaśkowski's work, gaining widespread acceptance by the 1960s [8]. In a ND system we can construct proofs showing that a formula φ is a logical consequence of a set of formulas Γ, written $\Gamma \vdash \varphi$.

O documento Gouveia e Dionisio nao deve ser citado por ser apenas um draft. Já agora, podemos usar este comando para deixar comentários ao longo do texto. Depois é fácil desativar todos de uma só vez.

Even within ND, there are many ways to represent proofs. The two main styles are the Gentzen-style, which organizes the proof in a tree-shaped structure, also known as deduction tree, and the Fitch style, which uses a linear structure with deeper indentation levels to represent assumptions or intermediate steps in the proof. Our article focuses on the first one.

Gentzen style trees represent proofs, and each nodes of the tree is a formula. The most basic tree is compose of just a formula, and more complex trees are obtained from these by successively applying inference rules. Rules are represented using horizontal lines, where its hypotheses appear above the line and the rule's conclusion appears below. Since some of these inference rules have hypothetical assumptions, Gentzen-style ND uses natural number marks on the leafs of a tree as a mechanism to reference such hypothesis. An hypothesis (tree leaf) is considered discharged or closed if its mark is referenced by a rule applied below in the tree, meaning that this hypothesis is an assumption of such rule. If it is not close, an hypothesis is said to be open. The rule's name and the corresponding marks for the hypotheses are placed on the right-hand side of the fraction representing the application of the rule. Given a complex tree, the formula at its root is called the conclusion of the proof, and formulas at the leaves are called hypotheses.

Regarding the inference rules, we will consider the usual classical logic rules. For each classical connective or quantifier Δ, we have two rules: introduction rule (Δ_I), which constructs more complex formulas from simpler ones, and elimination rules (Δ_E), which extracts information from complex formulas. Additionally, a special rule, known as absurdity ($\perp$), allows deriving any conclusion from an explicit contradiction $\perp$. Some rules can only be applied under specific circumstances, called side conditions. For simplicity, we will not detail these here, but we take them into consideration in the algorithm. Figure 1 lists the complete set of classical rules we consider. Greek letters represent generic formulas, symbols of the form $\mathcal{D}$ represent subtrees within branches, m and n denote marks, and $[(\varphi)^m$ over $\mathcal{D}$ represents the fact that φ can be used as an assumption in $\mathcal{D}$.

RG: Acho que a notação está vaga porque usámos a mesma notação "[ ]" para coisas diferentes (o que nunca é boa ideia).

The notation $[\varphi]^x_t$ indicates the substitution of the variable x with term t in formula φ.

Figure 2 shows an example of a ND proof which proves $\{\neg(\varphi \vee \psi)\} \vdash \neg\psi$.

Definition 1 (Well-Formed Tree Proof). *A tree proof is well-formed if and only if it is finite and applies the inference rules correctly.*

Definition 2 (Consequence Proven by a Proof). *Given a tree proof and a problem $\Gamma \vdash \phi$, we say that the tree solves the problem if and only if it is well-formed and the following conditions both hold:*

1. *The root of the tree is ϕ.*

Introduction Rules

$$\dfrac{\overset{\mathcal{D}_1}{\varphi} \qquad \overset{\mathcal{D}_2}{\psi}}{\varphi\wedge\psi}(\wedge_I) \qquad \dfrac{\overset{\mathcal{D}}{\varphi}}{\varphi\vee\psi}(\vee_{I_r}) \qquad \dfrac{\overset{\mathcal{D}}{\psi}}{\varphi\vee\psi}(\vee_{I_l}) \qquad \dfrac{\overset{[\varphi]^m}{\overset{\mathcal{D}}{\psi}}}{\varphi\to\psi}(\to_I, m)$$

$$\dfrac{\overset{[\varphi]^m}{\overset{\mathcal{D}}{\bot}}}{\neg\varphi}(\neg_I, m) \qquad \dfrac{\overset{\mathcal{D}}{[\varphi]^x_t}}{\forall x\,\varphi}(\forall_I) \qquad \dfrac{\overset{\mathcal{D}}{[\varphi]^x_t}}{\exists x\,\varphi}(\exists_I)$$

Elimination Rules

$$\dfrac{\overset{\mathcal{D}}{\varphi\wedge\psi}}{\varphi}(\wedge_{E_r}) \qquad \dfrac{\overset{\mathcal{D}}{\varphi\wedge\psi}}{\psi}(\wedge_{E_l}) \qquad \dfrac{\overset{\mathcal{D}_1}{\varphi_1\vee\varphi_2} \qquad \overset{[\varphi_1]^m}{\overset{\mathcal{D}_2}{\psi}} \qquad \overset{[\varphi_2]^n}{\overset{\mathcal{D}_3}{\psi}}}{\psi}(\vee_E, m, n)$$

$$\dfrac{\overset{\mathcal{D}_1}{\varphi} \qquad \overset{\mathcal{D}_2}{\varphi\to\psi}}{\psi}(\to_E) \qquad \dfrac{\overset{\mathcal{D}_1}{\varphi} \qquad \overset{\mathcal{D}_2}{\neg\varphi}}{\bot}(\neg_E) \qquad \dfrac{\overset{\mathcal{D}}{\forall_x\,\varphi}}{[\varphi]^x_t}(\forall_E) \qquad \dfrac{\overset{\mathcal{D}_1}{\exists_x\,\varphi} \qquad \overset{([\varphi]^x_y)^m}{\overset{\mathcal{D}_2}{\psi}}}{\psi}(\exists_E, m)$$

Absurdity Rule

$$\dfrac{\overset{[\neg\varphi]^m}{\overset{\mathcal{D}}{\bot}}}{\varphi}(\bot, m)$$

Fig. 1. List of rules for both PL and FOL

$$\dfrac{\dfrac{\neg(\varphi\vee\psi)^1 \qquad \dfrac{\psi^2}{(\varphi\vee\psi)}\;(\vee_{I_l})}{\bot}\;(\neg_E)}{\neg\psi}\;(\neg_I, 2)$$

Fig. 2. Tree proving $\{\neg(\varphi\vee\psi)\}\vdash\neg\psi$.

2. Every open hypothesis in the tree is contained in Γ.

3 Guidance Components

Developing an algorithm to provide advanced feedback on ND exercises begins with clearly defining its key features. Our idea was to design an effective feedback system capable of delivering relevant information to assist students in solving proofs, making the learning process more teaching-like and adapted to each student's resolution. With this focus, we identified four fundamental aspects of a well-designed feedback system:

- **Providing guidance on rule applications:** Some rule applications in ND are not obvious, making it difficult for students to progress. For example, sometimes, no matter how hard students try, the desired proof cannot be obtained. This may be because an indirect proof is needed: a sentence φ is proved by assuming $\neg\varphi$ and showing that it leads to a contradiction [ND-pack]. The system should be able to identify such situations and suggest the appropriate rule applications.
- **Breaking proofs into smaller sub-proofs:** To simplify reasoning, the feedback should allow students to focus on smaller proofs. By dividing proofs into smaller steps, it will reduce the cognitive load and encourage incremental learning.
- **Indicating the distance to a solution:** Showing how many steps (rule applications) are needed to complete the proof helps students maintain focus and gain a clear sense of progress.
- **Improvements in the proof:** Providing feedback about irrelevant steps taken or possible shortcuts that the student could apply to make the proof clearer. This can also be used to show different ways to tackle the same problem.

These components offer several advantages in the learning process. By providing structured and clear information, the system becomes more robust, helping students overcome challenges throughout the proof.

4 Algorithm

The algorithm is structured in three sequential steps, which we will discuss in more detail in the following subsections.

4.1 Transition Graph

The first step is to create the Transition Graph (TG). This graph stores the formulas that might be part of the final proof, as well as the rules that can be applied to each formula. In short, the TG sets up the rules of the "game".

To generate the graph, we need to specify the main goal of the exercise $\Gamma \vdash \varphi$, what the exercise wants us to prove, and the target goal $\Sigma \vdash \theta$, which is the part of the proof that needs to be completed. The main goal is used to generate all the natural proof paths, which are proofs built using only formulas derived from decomposing the main goal. The target goal, on the other hand, can sometimes be used to generate non-natural proof paths. By this, we mean proofs that include more complex formulas than those directly derived from the main goal. By considering both goals, the system is able to generate more personalized and user-guided proofs, as it also takes into account the deviations made by the user, which is one of the core elements of our algorithm.

Before we describe the procedure, let us introduce some definitions:

Definition 3 (Labeled Directed Hypergraph with Labeled Heads). *A* Labeled Directed Hypergraph with Labeled Heads *is a pair*

$$H = (V, E),$$

where:

- V *is a finite set of* nodes*, and*
- $E \subseteq V \times \mathcal{P}(V \times (V \cup \{\varepsilon\})) \times L$ *is a finite set of* labeled hyperedges*, where each hyperedge consists of:*
  - *a* tail*, which is a single input node from V,*
  - *a* labeled head*, which is a set of pairs $(v, w) \in V \times (V \cup \{\varepsilon\})$, where v is a head node and w is either a node or the empty symbol ε, and*
  - *a label $\ell \in L$.*

Definition 4 (Transition Graph). *A* Transition Graph (TG) *is a pair*

$$T_G = (F, T_E),$$

where:

- F *is a finite set of* formulas*, and*
- $T_E \subseteq F \times \mathcal{P}(F \times (F \cup \{\varepsilon\})) \times R$ *is a finite set of* labeled hyperedges *called* Transition Edge (TE)*, where each edge consists of:*
  - *a tail formula $f \in F$,*
  - *a set of pairs $(f_1, f_2) \in F \times (F \cup \{\varepsilon\})$, where f_1 is a* hypothesis *and f_2 is a* closed hypothesis*, and*
  - *a rule $r \in R$.*

A TE is the application of a rule to a formula. Figure 3 shows an example of the rule being applied $\vee_E$ and the corresponding edge.

We will have $T_E = T_E \cup \{(c, \{(a \vee b, \varepsilon), (c, a), (c, b)\}, \vee_E)\}$. Our edges always point from the conclusion to its hypotheses. If we were working with FOL proofs, we would also need to consider side conditions as part of T. The reason for using this type of graph is that it allows us to map rule applications directly into a data structure. In the final step, we will explain why it is necessary to store edges

Rule application

$$\frac{a \vee b \quad c \quad c}{c}\,(\vee_E,1,2)$$

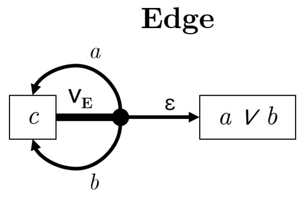

Fig. 3. Example of a transition edge.

Algorithm 1: Transition Graph Construction

```
   Input: Main goal $\Gamma \vdash \varphi$, Target goal $\Sigma \vdash \theta$
   Output: Transition Graph $T_G = (F, T_E)$
1  $F \leftarrow \Gamma \cup \Sigma \cup \{\varphi, \theta\}$ ;                    // Initialize formulas
2  $T_E \leftarrow \emptyset$ ;                                          // Initialize edges
   // Compute formulas
3  foreach $f \in F$ do
4      if $f$ was not already added as a negation then
5          $F \leftarrow F \cup \{\neg f\}$ ;               // Add negation for indirect rules
6      Decompose $f$ into parts $S$;
7      $F \leftarrow F \cup S$;
   // Compute transitions
8  foreach $f \in F$ do
9      if $f$ was not added as a negation then
10         $T_E = T_E \cup \{(f, \{(\bot, \neg f)\}, \bot)\}$;
11     if $f = \neg\alpha$ for some $\alpha$ then
12         $T_E = T_E \cup \{(\neg\alpha, \{(\bot, \alpha)\}, \neg_I)\}$;
13         $T_E = T_E \cup \{(\bot, \{(\alpha, \varepsilon), (\neg\alpha, \varepsilon)\}, \neg_E)\}$;
14     if $f = \alpha \vee \beta$ for some $\alpha, \beta$ then
15         $T_E = T_E \cup \{(f, \{(\alpha, \varepsilon)\}, \vee_{I_R})\}$;
16         $T_E = T_E \cup \{(f, \{(\beta, \varepsilon)\}, \vee_{I_L})\}$;
17         foreach $f' \in F$ do
18             $T_E = T_E \cup \{(f', \{(f, \varepsilon), (f', \alpha), (f', \beta)\}), \vee_E\}$;
19     if $f = \alpha \wedge \beta$ for some $\alpha, \beta$ then
20         $T_E = T_E \cup \{(\alpha \wedge \beta, \{(\alpha, \varepsilon), (\neg\beta, \varepsilon)\}, \wedge_I)\}$;
21         $T_E = T_E \cup \{(\alpha, \{(\alpha \wedge \beta, \varepsilon)\}, \wedge_{E_R})\}$;
22         $T_E = T_E \cup \{(\beta, \{(\alpha \wedge \beta, \varepsilon)\}, \wedge_{E_L})\}$;
23     if $f = \alpha \rightarrow \beta$ for some $\alpha, \beta$ then
24         $T_E = T_E \cup \{(\alpha \rightarrow \beta, \{(\beta, \alpha)\}, \rightarrow_I)\}$;
25         $T_E = T_E \cup \{(\beta, \{(\alpha, \varepsilon), (\alpha \rightarrow \beta, \varepsilon)\}, \rightarrow_E)\}$;
```

in this way. At this point, there is no need to keep track of the marks, since they can be easily added in the final steps when reconstructing the full proof.

With all the necessary definitions in place, we now present the procedure to generate the TG, as shown in Algorithm 1.

The computing of formulas and transitions can be done in one step to avoid unnecessary loops, but for simplicity, we kept them separate. The decomposing step is an important part to know in advance which formulas we will have in the graph and also to match with formulas that require that information, since they can be applied to every formula, such as the Elimination of Disjunction rule. The decomposition is done by splitting the formula at the outermost logical operators (if it is not an atomic formula). For example, $a \to \neg b$ can be split into a and $\neg b$, but a cannot be decomposed since it is atomic. However, we can decompose $\neg b$ into just b.

To illustrate the construction process in detail, we now present a full example in Figure 4, based on the specific main goal and target goal $\vdash a \to a$ (so we want a full proof for the problem).

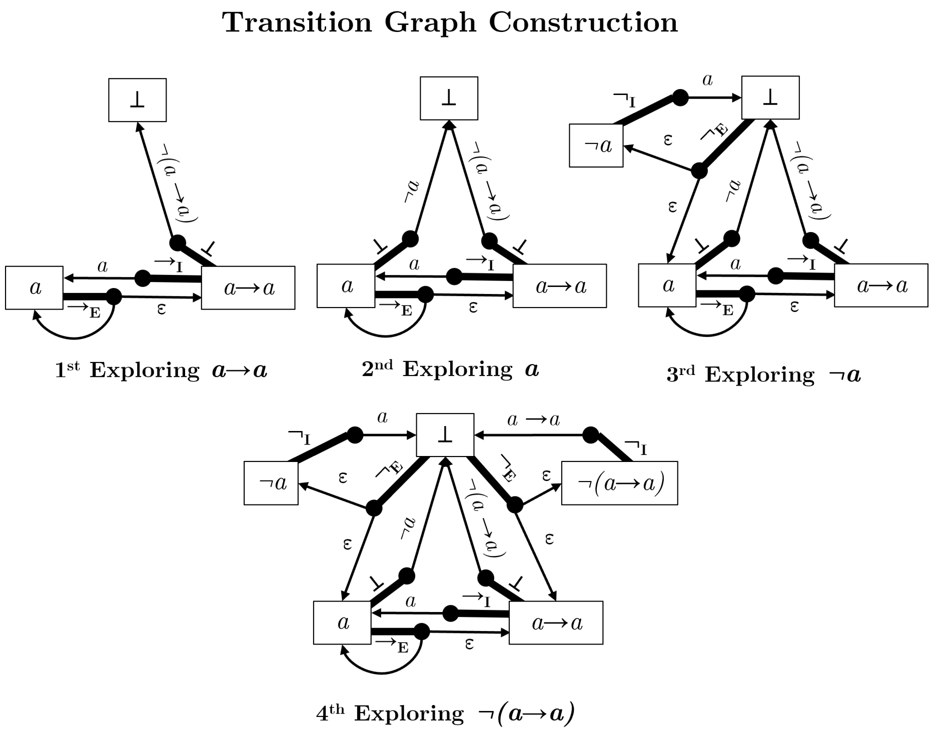

Fig. 4. Final TG generated from main goal and target goal $\vdash a \to a$

4.2 Proof Graph

The second step is to build the Proof Graph (PG). The structure of this graph is very similar to the TG, but it stores different objects. It stores the possible sub-goals derived from the target goal. In this graph, the nodes are goals, and the edges are adaptations of transition edges (TE) that now store goals. The purpose of this graph is to decompose the target goal into smaller goals that are easier to prove, until we reach goals that can be directly proved. In the end, after generating all sub-goals, the graph may contain multiple proof paths. Some of these paths may not lead to a solution, while others may succeed in proving the target goal.

To generate the PG, we use the TG, previously generated, and the target goal. Note that the target goal can also be the initial goal, in case we want to generate a full proof for the problem. Before we describe the procedure, we define some key terms:

Definition 5 (Proof Graph). *A* Proof Graph (PG) *is a pair*

$$P_G = (G, P_E),$$

where:

- *FG is a finite set of* goals*, and*
- $P_E \subseteq G \times \mathcal{P}(G \times (F \cup \{\varepsilon\})) \times R$ *is a finite set of* labeled hyperedges *called* Proof Edge (PE)*, where each edge consists of:*
  - *a tail goal* $g \in G$*,*
  - *a set of pairs* $(g_1, f_1) \in G \times (F \cup \{\varepsilon\})$*, where* g_1 *is a goal and* f_1 *is a* closed hypothesis*, and*
  - *a rule* $r \in R$*.*

Definition 6 (Proved Goal). *A goal* $\Delta \vdash \delta$ *is* proved *if either:*

- $\delta \in \Delta$*, or*
- *there exists a Proof Edge* $(g, T, r) \in P_E$ *in the Proof Graph* $P_G = (G, P_E)$*, such that* $g = \Delta \vdash \delta$*, and for every pair* $(g_1, f_1) \in T$*, the goal* g_1 *is proved.*

The definition of proved goal is extremely important because it is used as a stopping condition to avoid the algorithm looping through unnecessary goals and to guarantee that the proof is valid. This is only possible due to the type of graph chosen, as it allows us to capture the relation between the hypotheses and the conclusion of each rule application. Figure 5 shows an example of a PE and how these relations can be captured. In the figure, we want to prove $\{a \vee b\} \vdash a$. By applying the Elimination of Disjunction rule, we notice that one of the hypotheses cannot be closed using only this rule, even if the other two hypotheses are closed. So, what we actually prove with this rule is $\{a \vee b, a\} \vdash a$, which is different from our goal. To accurately track which goals are proved and ensure the proof only contains valid proved goals, this information must be stored in a hypergraph structure. This is why hypergraphs are necessary.

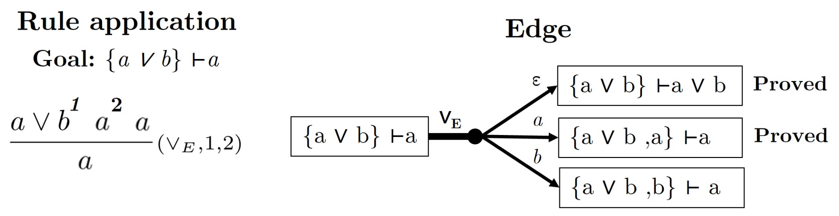

Fig. 5. Prove $\{a \vee b\} \vdash a$ using the Elimination of Disjunction rule

Algorithm 2: Proof Graph Construction

```
    Input: Transition Graph $T_G = (F, T_E)$, Target goal $t_g$
    Output: Proof Graph $P_G = (G, P_E)$
1   $G \leftarrow \{t_g\}$ ;                                // Initialize set of goals
2   $P_E \leftarrow \emptyset$ ;                            // Initialize set of proof edges
    // Compute sub-goals
3   foreach $g = \Sigma \vdash \theta \in G$ do
4       if $g$ is proved then
5           continue ;                                  // Skip proved goal
6       if stopping condition is reached then
7           break ;                                     // Avoid excessive expansion
        // Get transition edges for formula $\theta$
8       $TE_\theta \leftarrow \{(f, H, r) \in T_E \mid f = \theta\}$;
9       foreach $(f, H, r) \in TE_\theta$ do
10          $T \leftarrow \emptyset$ ;                      // Store transitions to each hypothesis
11          foreach $(f_1, f_2) \in H$ do
                // Create sub-goal by extending the current premises with
                   the closed hypothesis
12              $g_{\text{new}} \leftarrow (\Sigma \cup \{f_2\}) \vdash f_1$;
13              $T \leftarrow T \cup \{(g_{\text{new}}, f_2)\}$;
14              $G \leftarrow G \cup \{g_{\text{new}}\}$ ;          // Add sub-goal
15          $P_E \leftarrow P_E \cup \{(g, T, r)\}$ ;               // Add proof edge
```

With all the necessary definitions in place, we now present the procedure to generate the PG, as shown in Algorithm 2.

This part of the algorithm can generate very large graphs, potentially exponential in size, with millions of distinct goals depending on the complexity of the problem. In most cases, we do not want to explore the entire goal space, as many goals are extremely complex and do not provide useful feedback. Therefore, stopping conditions are required. These may include: limiting the total number of goals explored, setting a maximum number of hypotheses allowed per goal, or enforcing a timeout.

Figure 6 shows the PG generated using the TG from Figure 4 and target goal $\vdash a \to a$, with a limit of 9 goals explored. Nodes with solid borders represent proved goals, while nodes with dashed borders represent unproved goals. Since our target goal is proved, we know that at least one solution was found.

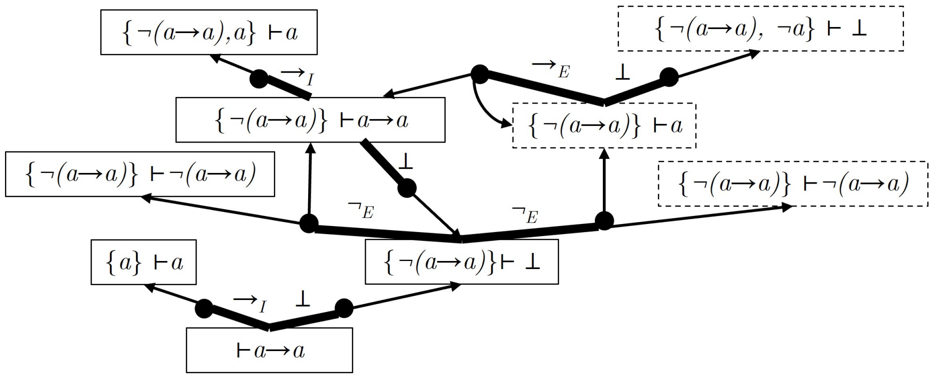
$\{\neg(a\to a), a\} \vdash a$
$\{\neg(a\to a), \neg a\} \vdash \bot$
$\to_I$
$\to_E$
$\bot$
$\{\neg(a\to a)\} \vdash a\to a$
$\{\neg(a\to a)\} \vdash a$
$\{\neg(a\to a)\} \vdash \neg(a\to a)$
$\bot$
$\{\neg(a\to a)\} \vdash \neg(a\to a)$
$\neg_E$
$\neg_E$
$\{a\} \vdash a$
$\{\neg(a\to a)\} \vdash \bot$
$\to_I$
$\bot$
$\vdash a\to a$

Fig. 6. Example of Proof Graph using the TG from Figure 4 and target goal $\vdash a \to a$

4.3 Trim Graph

The final stage of our algorithm is to trim the SG and keep only the valid solutions to the problem. In other words, the final trimmed graph will contain only states that lead to a complete and valid proof. To achieve this, we remove unclosed states and extra edges from the SG.

We define two different strategies to trim the graph. Both use a standard graph traversal technique, breadth-first search, to identify which states and edges should be kept. The difference lies in what each strategy prioritizes: the **Height Trim Strategy** focuses on finding proofs with the smallest height (fewest layers of rule applications), while the **Size Trim Strategy** aims to find proofs with the fewest total steps (smallest number of rule applications).

Height Trim Strategy Procedure: This strategy loops through all closed states and tracks the height needed to reach each one. For every descendant of a closed state, the algorithm continues this tracking process. Because it uses breadth-first traversal, the first time a node is reached, it's guaranteed to be through the shortest possible path (in terms of height). This makes the strategy efficient.

Size Trim Strategy Procedure: This strategy also starts from closed states. It updates the size (number of steps) required to reach each ancestor. For each state, it keeps only the edge that leads to a smaller proof. For every descendant of a closed state, the algorithm continues this tracking process. This process ensures that each state keeps at most one optimal incoming edge, leading to the shortest proof found within the SG's search space. Although slower than the height trim strategy, it finds more concise proofs in terms of rule applications.

Figure 7 shows an example of a trimmed SG using the **Size Trim Strategy**, on the SG shown in Figure 6. The states that could not be closed were removed, and the edges coming from higher-level solutions were also trimmed.

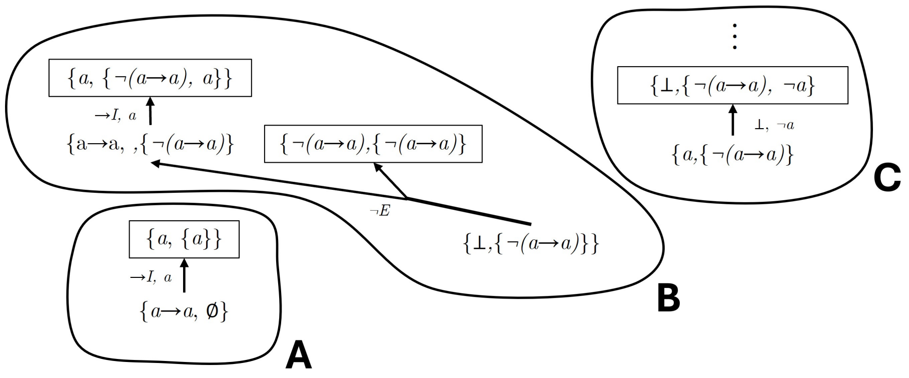

Fig. 7. Trimmed State Graph showing valid proof paths.

To check whether a valid solution was found for the full proof, we simply verify if the initial state appears in the trimmed graph. In the example above, the solution corresponds to tree **A**, as we want to solve the problem $\vdash a \rightarrow a$. That solution is always the smallest in height/size, respectively. With this trimmed graph, we can now generate feedback by querying which states are still missing in the student's proof. Figures 8 and 9 illustrate examples of how feedback can be generated from the final graph.

In this first example, the student does not know how to proceed after applying the Absurdity rule. By running the algorithm and querying the final graph with the state that is still unsolved, we get a possible solution. This case represents tree **B** in Figure 7. Knowing the remaining part of the proof, we can generate feedback. For example, we can tell the student to apply the Elimination of the

Negation rule using $a \to a$ and $\neg(a \to a)$ (**Providing guidance on rule applications**). In this specific case, we cannot give hints about sub-proofs to solve the problem, as the solution is already small. But in some cases where the solution is bigger, we can do that (**Breaking proofs into smaller sub-proofs**). We can also specify how far the student is from the final proof. In this case, we can say that they are two rules away from completing the proof (**Indicating the distance to a solution**). Finally, we can also suggest some improvements in the resolution. In this case, the student shifts their solution by applying the Absurdity rule, making it longer. That information can also be extracted from the graph by comparing the smallest proof (the one with the initial state) with the student's final proof (**Improvements in the proof**).

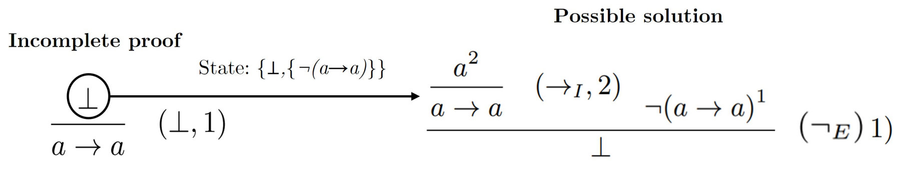

Fig. 8. Extracting a solution to produce feedback using an existing state

In this second example, a solution cannot be found, as the state assigned to the unresolved part of the proof does not belong to the final graph. In this case, we can inform the student that the path they are taking may be too complex, and we can suggest going back X rule applications until the algorithm finds the correct path again to guide the student. We cannot affirm that there is no solution, because we may not have explored the whole space of possible solutions. In this example, if the student removes the Elimination of Negation rule (one step back), we return to the situation previously presented.

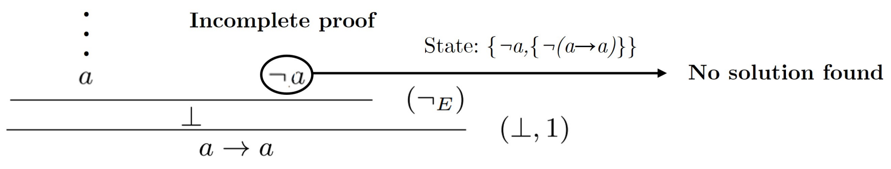

Fig. 9. Extracting a solution to produce feedback using a non-existing state.

These methodologies can also be used to assess exercises. For example, by computing how far the student's resolution is from a possible solution if the problem remains unsolvable, or how far it is shifted from the best solution. In some cases, based on the size of the explored solution space, we can say that the student overcomplicated the resolution.

5 Limitations

The algorithm was developed for pedagogical purposes, so efficiency in proof generation was not our main focus. It can generate solutions for most exercises used in teaching environments but is more limited when searching for solutions in FOL proofs, as the solution space is infinite. Our algorithm is sound: if it finds a solution, it is definitely correct. This is guaranteed by the TG, which only generates valid transitions for each formula, and by the SG, which ensures that all leaves in the proof are correctly closed.

Regarding completeness, our algorithm is not complete because it can only find solutions up to a certain depth. Some proofs generate infinite graphs, so a solution may not be found. In most cases, this is not a problem, as we aim to find direct proofs. If a student's proof deviates too much from the solution, it is not useful to provide feedback on that solution because the student is overcomplicating the problem. For example, if a teacher sees that a student is still working on a problem that can be solved in 10 steps, but the student's current resolution already has 50 steps, even if a solution exists following the student's approach, is it helpful to give feedback on it? Will the student really learn from that?

6 Related Work

Several algorithms have been developed to verify the validity of logical formulas. One common approach is to use resolution to find contradictions. This method begins by converting formulas into Conjunctive Normal Form, which is easier for automated systems to process. After this transformation, the algorithm applies resolution rules to search for contradictions. If a contradiction is found, the formula is not valid. If no contradiction appears after checking all possible combinations, the formula is considered valid. However, this method often does not produce proofs that are easy for humans to understand. To address this, David Robinson [10] proposed an algorithm that uses a version of resolution compatible with natural deduction. His method works by encoding formulas into special clauses and then using resolution steps to reconstruct a natural deduction proof. The result is a proof that is easier for people to read and understand. Another approach is the algorithm developed by Xuehan Maka Hu [6]. This algorithm works by recursively applying introduction rules to break down the conclusion, and then using elimination rules to handle the assumptions. It generates complete and human-readable natural deduction proofs in Gentzen style for propositional logic. Although it produces understandable proofs, the algorithm is not very flexible. Its strict rule application can sometimes lead to complicated solutions for problems that could be solved more simply.

Bolotov's [3] algorithm is another example of a proof-searching method designed for natural deduction in propositional logic. It follows a goal-directed strategy, combining forward and backward rule applications. It updates the goals dynamically and stops when a proof is found or when no further progress is possible, returning a counter-example in the latter case. One limitation of this algorithm is that it does not include a mechanism to remove irrelevant or unnecessary branches during proof search. These extra branches may contain valid but unneeded steps, which can make the proofs longer and harder to follow. LOGAX [5] is an interactive tutoring tool that constructs Hilbert-style axiomatic proofs in propositional logic. It is designed to provide students with step-by-step hints and feedback. LOGAX uses a version of Bolotov's algorithm to generate directed acyclic multigraphs, which allow the system to store and explore multiple proof paths for the same problem. It can adapt the proof it generates to match the student's reasoning, offering both forward and backward hints depending on the student's progress. Despite its strengths, LOGAX has some limitations. It is only designed for propositional logic and can only generate linear Hilbert-style proofs. Since it is based on Bolotov's algorithm, it also inherits some of its weaknesses, including the possibility of generating longer proofs with irrelevant or unnecessary steps.

The algorithm proposed by Ahmed, Gulwani, and Karkar [2] uses a very similar approach to our algorithm but is applied to a different set of inference rules and only works for PL. It is based on a hypergraph called the Universal Proof Graph (UPG), where nodes represent possible formulas (they are represented as bitvectors that map each formula to a truth table, reducing the number of formulas in the graph for better performance), and the edges represent possible rule applications that can be applied under each node. From that graph, we can extract abstract proofs (called "abstract" because expressions are still mappings to truth tables and not actual formulas), which are later converted into natural deduction proofs. The algorithm can also be used to generate problems and specify their difficulty, which can be useful for teachers. A big problem the algorithm faces is that it can only solve PL problems, since we cannot map FOL expressions to truth tables. Another issue is related to proof size, as the algorithm cannot guarantee that the solution found is the smallest one within the space explored.

7 Conclusion

Conclusion

References

1. Umair Ahmed, Sumit Gulwani, and Amey Karkare. Automatically generating problems and solutions for natural deduction. pages 1968–1975, 08 2013.
2. Umair Ahmed, Sumit Gulwani, and Amey Karkare. Automatically generating problems and solutions for natural deduction. pages 1968–1975, 08 2013.

3. Alexander Bolotov, Vyacheslav Bocharov, Alexander Gorchakov, and Vasilyi Shangin. Automated first order natural deduction. In Bhanu Prasad, editor, *Proceedings of the 2nd Indian International Conference on Artificial Intelligence, Pune, India, December 20-22, 2005*, pages 1292–1311. IICAI, 2005.
4. Phokion Kolaitis, Daniel Leivant, and Moshe Vardi. Panel: logic in the computer science curriculum. volume 30, pages 376–377, 01 1998.
5. Josje Lodder, Bastiaan Heeren, Johan Jeuring, and Wendy Neijenhuis. Generation and use of hints and feedback in a hilbert-style axiomatic proof tutor. *Int. J. Artif. Intell. Educ.*, 31(1):99–133, 2021.
6. Xuehan Maka Hu. Automatic generation of human readable proofs.
7. Paolo Mancosu, Sergio Galvan, and Richard Zach. 65natural deduction. In *An Introduction to Proof Theory: Normalization, Cut-Elimination, and Consistency Proofs*. Oxford University Press, 08 2021.
8. Francis Jeffry Pelletier. A brief history of natural deduction. *History and Philosophy of Logic*, 20(1):1–31, 1999.
9. Ján Perháč, Samuel Novotný, Sergej Chodarev, Joachim Tilsted Kristensen, Lars Tveito, Oleks Shturmov, and Michael Kirkedal Thomsen. Onlineprover: Experience with a visualisation tool for teaching formal proofs. *Electronic Proceedings in Theoretical Computer Science*, 419:55–74, May 2025.
10. David Robinson. Using resolution to generate natural proofs.
11. Jan von Plato. *Natural deduction*, page 31–63. Cambridge University Press, 2014.