# 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 fundamental in many fields, including programming, databases, AI, and algorithms [4]. Among logic topics, ND stands out for its role in developing students' reasoning skills [1], crucial for structured thinking and argumentation [10]. ND exercises are challenging due to their complex reasoning and numerous rules, which students often find hard to apply correctly. Keeping track of multiple steps and assumptions can be confusing, leading to difficulties in mastering these exercises.

Despite this importance, few online tools effectively support ND learning. Existing tools generally offer limited feedback, focusing on syntax or semantics but failing to guide students through conceptual challenges [9]. Students often get stuck when unable to proceed or when overcomplicating solutions. This situation reveals the need for advanced feedback systems that generate complete proofs to guide students effectively. However, most existing algorithms lack flexibility, depend on specific ND systems, and cannot adapt dynamically to a user's reasoning. Personalized feedback aligned with the student's approach remains a challenge.

In this article, we present an algorithm developed with a pedagogical focus to support building and verifying ND proofs. The algorithm generates multiple Gentzen-style proofs for problems in Propositional Logic (PL) and First-Order Logic (FOL). Using directed hypergraphs, it captures multiple valid proof paths and adapts to the user's reasoning, providing step-by-step guidance. Using graph search methods, the system finds correct and shortest proofs, allowing it to give feedback on how good the solution is and track student progress.

## 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 $\varphi$ is a logical consequence of a set of formulas $\Gamma$, written $\Gamma \vdash \varphi$. 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 $\Delta$, we have two rules: introduction rule ($\Delta_I$), which constructs more complex formulas from simpler ones, and elimination rules ($\Delta_E$), which extracts information from complex formulas. Additionally, a special rule, known as absurdity ($\bot$), allows deriving any conclusion from an explicit contradiction $\bot$. 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 $\varphi$ can be used as an assumption in $\mathcal{D}$. With $[\varphi]^x_t$ we represent the result of substituting free occurrences of $x$ with $t$ in $\varphi$.

**Introduction Rules**

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

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

**Elimination Rules**

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

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

**Absurdity Rule**

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

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

A ND proof is said to be well-formed if and only if it is finite and the inference rules are applied correctly. Moreover, we say that a ND proof proves the consequence $\Gamma \vdash \phi$ if and only if it is well-formed, the root of the tree is $\phi$, and every open hypothesis in the tree is contained in $\Gamma$. In Fig. 2 we have an example of a well-formed ND proof of $\{\neg(\varphi \vee \psi)\} \vdash \neg\psi$.

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

Fig. 2: ND tree proving $\{\neg(\varphi \vee \psi)\} \vdash \neg\psi$.

## 3 Requirements

Before we introduce the proposed algorithm, we must clarify its main goal. We want our algorithm to be able to support advanced feedback for students learning and practicing ND proofs. Such effective feedback system should be able to deliver relevant information to assist students at any stage of their exercise resolution. Focusing on this main goal, we identified four fundamental aspects a well-designed feedback system should satisfy:

- **Providing guidance on rule applications:** Some rule applications in ND are not obvious, making it difficult for students to progress. A paradigmatic example is the case of proofs by contradiction, which is a distinctive feature of classical logic. In some cases no direct proof exists, and the result can only be proved by contradiction: $\varphi$ is proved by assuming $\neg\varphi$ and showing that this leads to a contradiction. The feedback system should be able to identify such situations and suggest the appropriate rule applications.
- **Breaking proofs into smaller sub-proofs:** Proofs in ND are incrementally built from smaller proofs. Dividing proofs into smaller steps reduces the cognitive load and simplifies reasoning. Therefore, the feedback system should encourage students to start with smaller proofs and incrementally build the main proof.
- **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 could make the proof clearer. It should also allow visualizing different ways to tackle the same problem.

Designing an algorithm that provides the basis for a feedback mechanism satisfying the above requirements is quite challenging. We aim to provide structured and clear information, as a tool for helping students to overcome the usual challenges of producing ND proofs.

## 4 Algorithm

In this section we present our proposal of an algorithm to allow advanced feedback for ND exercises. Our algorithm is structured in three sequential steps, which we will discuss in more detail in the following subsections.

### 4.1 Transition Graph

RG: Estes dois parágrafos inicias precisavam de alguma revisão (em termos de apresentação das ideias), mas não consegui fazer. Posso voltar a isto depois.

The first step is to create the so-called 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 personalised 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. For the type of information we want to store, we will make use of a special type of graph.

**Definition 1 (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, over a finite set of labels $L$, where each hyperedge $(t, \{(h_i, \ell_i) : i \in I\}, \ell)$ consists of:*
  - *a tail $t$, representing a single input node from $V$,*
  - *a set of labeled heads, which is a set of pairs $(h_i, \ell_i) \in V \times (V \cup \{\varepsilon\})$, where $h_i$ indicates one of the output nodes and $\ell_i$ is its label, which is either a node or the empty symbol $\varepsilon$, and*
  - *the global label $\ell$ of the hyperedge, which is an element of $L$.*

The Transition Graph (TG) we will construct is a hypergraph as defined above, where vertices are formulas and each hyperedge represents the application of an inference rule, which, as we have seen, may have mode than one premisse.

To give some intuition on the formal construction of TG, we now present an example, in Figure 3, that shows how the applications of rule $\vee_E$ can be represented using hypergraphs.

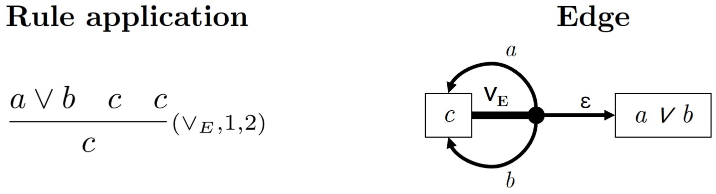


Fig. 3: Example of a transition edge.

Formally, the above hyperedge is represented as $(c, \{(a \vee b, \varepsilon), (c, a), (c, b)\}, \vee_E)$. Edges always go from the unique conclusion of the rule to its hypotheses. The labels of the heads of the hyperedge represent the possible additional hypothesis that can be used in that branch. In the particular case of the above rule $\vee_E$, each side of the disjunction can be used as an additional hypothesis in the second and third branch of the rule, representing the reasoning by cases encoded by the rule. If we were working with FOL proofs, we would also need to consider side conditions as part of TG. Also, at this point of the construction, there is no need to keep track of the marks. We will see later that we can easily deal with these in the final step of the construction.

We now present, in Algorithm 1, the concrete procedure to generate the TG. As we said, we have as input the consequence we aim to prove $\Gamma \vdash \varphi$ and a consequence $\Sigma \vdash \theta$ representing a partial proof of an user. The computation of the relevant formulas and transitions between these can be done in just one loop, but, for the sake of simplicity of presentation, we kept these separated. For the formulas we just consider the set of all subformulas of the formulas contained in the consequence we want to prove and in the partial consequence, together

**Algorithm 1:** Transition Graph Construction

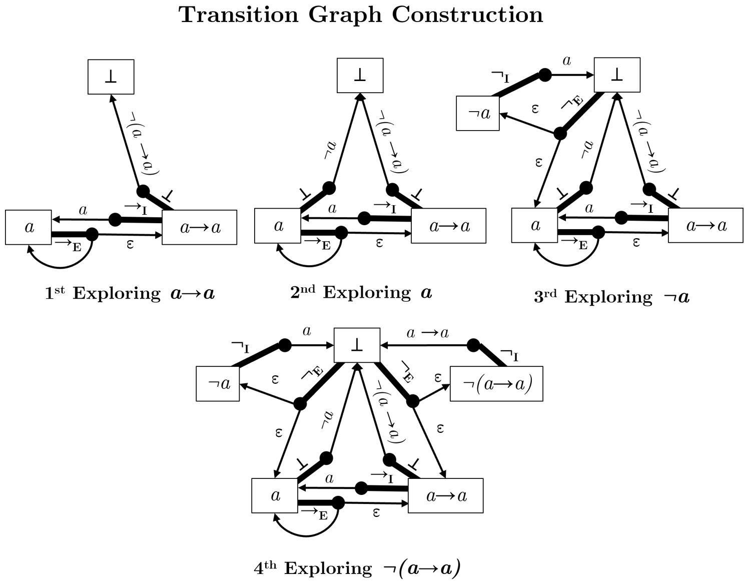
```
Input: Main goal $\Gamma \vdash \varphi$, Target goal $\Sigma \vdash \theta$
Output: Transition Graph $T_G = (F, T_E)$
$F \leftarrow \Gamma \cup \Sigma \cup \{\varphi, \theta\}$ ;                    // Initialize formulas
$T_E \leftarrow \emptyset$ ;                                              // Initialize edges
// Compute formulas
foreach $f \in F$ do
    if $f$ was not already added as a negation then
        $F \leftarrow F \cup \{\neg f\}$ ;                    // Add negation for indirect rules
    Decompose $f$ into parts $S$;
    $F \leftarrow F \cup S$;
// Compute transitions
foreach $f \in F$ do
    if $f$ was not added as a negation then
        $T_E = T_E \cup \{(f, \{(\bot, \neg f)\}, \bot)\}$;
    if $f = \neg\alpha$ for some $\alpha$ then
        $T_E = T_E \cup \{(\neg\alpha, \{(\bot, \alpha)\}, \neg_I)\}$;
        $T_E = T_E \cup \{(\bot, \{(\alpha, \varepsilon), (\neg\alpha, \varepsilon)\}, \neg_E)\}$;
    if $f = \alpha \vee \beta$ for some $\alpha, \beta$ then
        $T_E = T_E \cup \{(f, \{(\alpha, \varepsilon)\}, \vee_{I_R})\}$;
        $T_E = T_E \cup \{(f, \{(\beta, \varepsilon)\}, \vee_{I_L})\}$;
        foreach $f' \in F$ do
            $T_E = T_E \cup \{(f', \{(f, \varepsilon), (f', \alpha), (f', \beta)\}), \vee_E\}$;
    if $f = \alpha \wedge \beta$ for some $\alpha, \beta$ then
        $T_E = T_E \cup \{(\alpha \wedge \beta, \{(\alpha, \varepsilon), (\neg\beta, \varepsilon)\}, \wedge_I)\}$;
        $T_E = T_E \cup \{(\alpha, \{(\alpha \wedge \beta, \varepsilon)\}, \wedge_{E_R})\}$;
        $T_E = T_E \cup \{(\beta, \{(\alpha \wedge \beta, \varepsilon)\}, \wedge_{E_L})\}$;
    if $f = \alpha \rightarrow \beta$ for some $\alpha, \beta$ then
        $T_E = T_E \cup \{(\alpha \rightarrow \beta, \{(\beta, \alpha)\}, \rightarrow_I)\}$;
        $T_E = T_E \cup \{(\beta, \{(\alpha, \varepsilon), (\alpha \rightarrow \beta, \varepsilon)\}, \rightarrow_E)\}$;
```

with the negation of these formulas. This way, all formulas that could appear in ND proof of the main consequence are vertices of TG. The hyperedges of TG correspond to the possible rule applications between these considered formulas.

To illustrate the construction process, we now present a full example in Figure 4. The input in this case is the main consequence ?? and partial consequence $\vdash a \to a$??.

RG: Não percebo qual é o input neste caso: qual a main consequence e a partial consequence?

DM: Neste caso são o mesmo, por outras palavras, estamos a computar uma solução completa para o problema e não necessariamente a gerar feedback para um passo incompleto de uma prova, por isso main e partial $= \vdash a \to a$. A minha ideia era ter feito um exemplo em que, de facto, se completasse uma prova mas o numero de nos e arestas cresce demasiado e nao consigo representar o grafo mesmo sendo um pequeno desvio ex: $\{\neg(a \to a)\} \vdash a \to \bot$ já ficamos com 6 nos e 12 edges


**Transition Graph Construction**

$1^{st}$ **Exploring** $\boldsymbol{a \to a}$

$2^{nd}$ **Exploring** $\boldsymbol{a}$

$3^{rd}$ **Exploring** $\boldsymbol{\neg a}$

$4^{th}$ **Exploring** $\boldsymbol{\neg(a \to a)}$

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. In short, the PG aims to find the maximum number of different "game" combinations.

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 2 (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 3 (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.

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

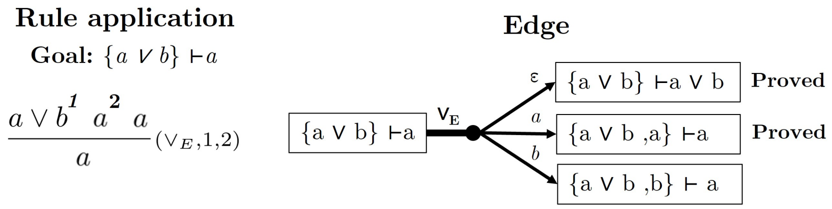


Fig. 5: Prove $\{a \vee b\} \vdash a$ using the Elimination of Disjunction rule

**Algorithm 2:** Proof Graph Construction

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

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.

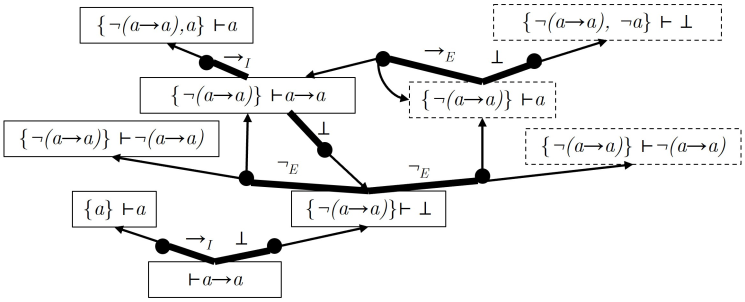


Fig. 6: Example of Proof Graph using the TG from Figure 4 and target goal $\vdash a \to a$

### 4.3 Proof Graph Trimming

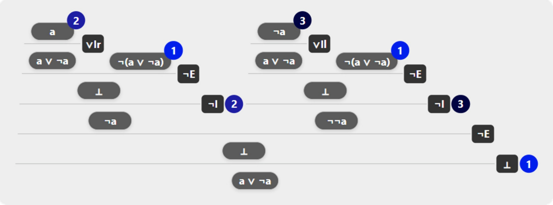
The final stage of our algorithm consists of trimming the PG to keep only the valid solutions to the problem. The resulting trimmed graph includes only proved goals and the edges that lead to the shortest proofs.

There are two different ways to define what constitutes a short proof: one based on height (Height Trim Strategy), and the other based on the number of formulas involved (Size Trim Strategy). Both strategies rely on a standard graph traversal technique, namely *breadth-first search*, to determine which goals and edges should be preserved.

The trimming process begins by iterating through all goals and discarding those that cannot be proved. Then, one of the following trimming strategies is applied:

**Height Trim Strategy (HTS):** This algorithm traverses the PG in reverse order: it starts from the leaves and visits nodes using breadth-first search. Since this traversal guarantees that each node is first reached through the shortest possible path in terms of height, the algorithm retains only the first PE that reaches each goal. All other incoming edges to the same goal are discarded, even if other solutions exist with the same height.

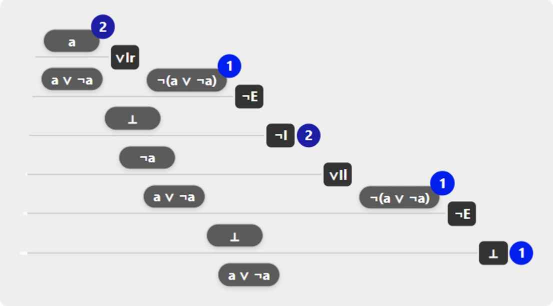
**Size Trim Strategy (STS):** This strategy is similar to HTS, but it tracks the size of each proof, defined as the number of formulas involved. Instead of retaining the first edge to reach a goal, it must explore all incoming edges to identify the one that yields the smallest proof. Consequently, a goal may be



**Height: 6 Size: 12**

Fig. 7: Example of a full proof generated by the algorithm for $\vdash a \vee \neg a$, using HTS.

visited multiple times, making this strategy more computationally expensive. However, it often results in more concise proofs in terms of rule applications.



**Height: 7 Size: 9**

Fig. 8: Example of a full proof generated by the algorithm for $\vdash a \vee \neg a$, using STS.

Another key feature that distinguishes us from other algorithms is that we do not store just a single solution, but rather a set of possible solutions. This is important to avoid recomputing the entire algorithm when generating feedback for the same problem. However, it comes at a cost in terms of space, as it stores thousands of goals. The algorithm can be queried to generate a new feedback step if all formulas in the new target goal are contained within the set of formulas in the TG used to generate the final trimmed PG. Figure 9 shows the TG from Figure 6 after being trimmed using STS. The algorithm found a solution **A** for the target problem and also found solutions for each of the subgoals presented in the graph.

### 4.4 Feedback Generation

With the final graph, we can now generate feedback by querying which goals remain unproved in the student's proof. Figures 10 and 11 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 querying the graph with the goal that is still unproved, we get the solution **B** in Figure 9. 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**

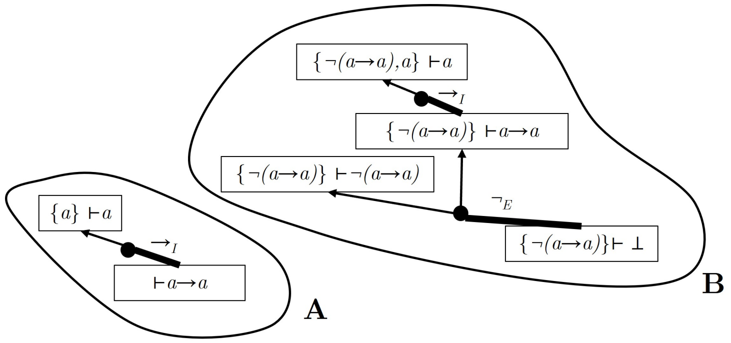
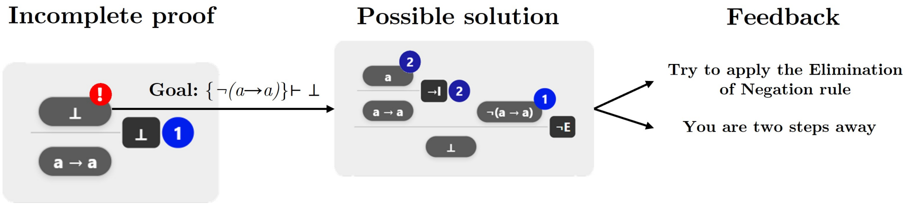


Fig. 9: Trimmed TG using STS.

**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 (**Improvements in the proof**).



Try to apply the Elimination of Negation rule

You are two steps away

Fig. 10: Extracting a solution to produce feedback from a proved goal

In this second example, a solution cannot be found, as the goal 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. For example, the final graph was only constructed considering the first 9 nodes. In this example, if the student removes the Elimination of Negation rule (one step back), we return to the situation previously presented.

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.

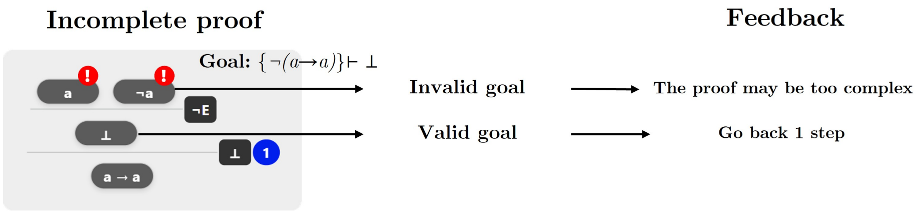


Fig. 11: Extracting a solution to produce feedback from an unproved goal.

Our algorithm is already implemented and has been tested. It is currently being integrated into an online tool focused on ND proofs. Some of the figures shown were actually taken from this tool, which allows students to practice solving this type of problem.

## 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 grows faster. 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 PG, which ensures that all goals in the proof are proved. 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? Our algorithm stores multiple solutions for the same goal, which incurs memory costs since thousands of nodes may be stored in the final graph. However, this trade-off enables fast feedback generation, as the computational work has already been done.

## 6 Related Work

Several algorithms have been developed to verify the validity of logical formulas. One common approach is the resolution algorithm, widely used in automated theorem proving. It is effective but often produces proofs that are difficult for humans to follow. Xuehan Maka Hu's [6] algorithm generates complete, human-readable Gentzen-style ND proofs for PL by recursively applying introduction and elimination rules. While producing clear proofs, its rigid rule application can lead to complex solutions, and it was not designed to produce feedback.

Bolotov's [3] algorithm is a goal-directed proof search for ND in PL, combining forward and backward reasoning. It lacks a way to prune irrelevant branches, which can lead to longer, harder-to-follow proofs. LOGAX [5] is an interactive tutoring tool for linear Hilbert-style proofs in PL and FOL, based on Bolotov's method. It adapts proofs to student reasoning through graphs but shares Bolotov's limitations, including sometimes producing unnecessarily long proofs. Ahmed, Gulwani, and Karkar's [2] algorithm, works for PL using a Universal Proof Graph (UPG) with bitvector-encoded formulas to improve efficiency. It extracts abstract proofs later converted to ND and can generate problems with specified difficulty. However, it only handles PL problems and cannot guarantee minimal proof size.

Despite these advances, existing methods either focus exclusively on PL or produce proofs that lack flexibility and adaptability to students' reasoning processes. Additionally, most approaches generate a single proof, which limits the possibility of giving personalized feedback and tracking the student's progress effectively. Because of these limitations, there was a need to create an algorithm that could generate multiple valid ND proofs for both PL and FOL, while providing efficient and adaptive feedback that follows the student's reasoning.

## 7 Conclusion

This work introduced an algorithm designed to support the learning and teaching of ND, with the aim of generating high-quality educational feedback. Through the construction of graphs based on labeled directed hypergraphs, the system is able to automatically generate complete, human-readable ND proofs and explore multiple proof paths for a given problem.

The main contributions of this work are:

- A method for generating multiple valid ND proofs for both PL and FOL.
- An efficient feedback mechanism that aligns with the student's reasoning, rather than enforcing a single rigid solution path.
- A way to quantify progress, offering metrics such as distance to a valid proof and detection of redundant or more complex steps.

Furthermore, the algorithm has the potential to generate exercises with specified difficulty levels, and exploring this capability is planned as part of future work.

Unlike previous methods, our algorithm not only returns a single best solution but also provides step-by-step feedback that aligns with the student's problem-solving process, enhancing its educational value. Additionally, it stores multiple solutions in advance, making real-time feedback efficient and scalable.

The algorithm is fully implemented and is being integrated into an online learning platform. We expect it to have immediate impact in classroom and self-study settings, helping students better understand ND and teachers to evaluate student solutions more effectively.

## 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. Jan von Plato. *Natural deduction*, page 31–63. Cambridge University Press, 2014.