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 [6]. 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 [13].

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 [11]. 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 deter-

mining how far a student's resolution is from a valid or optimal solution, thereby offering a measurable assessment of proofs.

2 Background

ND is a logical proof system that reflects mathematical and everyday reasoning under assumptions [9]. It emerged in 1934 with Gentzen and Jaśkowski's work, gaining widespread acceptance by the 1960s [10]. Using ND, an important goal is to verify whether a formula φ is a semantic consequence of a set of formulas Γ, written $\Gamma \vdash \varphi$. This means that whenever all formulas in Γ hold, φ must also hold because it logically follows from Γ [4, 5]. There are many ways to represent these proofs. The two main styles that exist are the Gentzen style, which organizes the proof in a tree-shaped structure, 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 uses tree-shaped structures, also known as deduction trees. These structures represent proofs and are built by starting with individual trees, which are formulas, and successively applying rules of inference to generate new and more complex trees. The formula at the root is the conclusion of the proof, and formulas at the leaves are called hypotheses and are generally associated with marks. In this article, we use numbers to denote such marks. Marks are used to identify distinct assumptions and to indicate when they are discharged (closed). A hypothesis is considered discharged if its mark is referenced by a rule. Rules are represented using fractions (a horizontal line), where hypotheses appear above the line and the rule's conclusion appears below. The rule's name and the marks for the hypotheses are normally placed on the right-hand side of the fraction. There are two main groups of rules for each logical connective: introduction (I), which constructs more complex formulas from simpler ones, and elimination (E), which extracts information from complex formulas. Additionally, a special rule, known as absurdity ($\bot$), allows deriving any conclusion from a contradiction. Each rule has its own characteristics, and some can only be applied under specific circumstances, called side conditions. For simplicity, we do not discuss these here, but they are included in our algorithm. Figure 1, list all the rules considered in our implementation. Greek letters represent generic formulas. The symbols $\mathcal{D}$ represent subtrees within branches, while m and n denote marks. The notation $[\varphi]^a_b$ indicates the substitution (or mapping) of terms from a to b, and $[\varphi]^m$ represents which assumptions can be used in each branch by the rule.

These proofs can be constructed either bottom-up, from the conclusion to the hypotheses, or top-down, from the hypotheses to the conclusion. Figure 2 shows an example of a proof, where the goal is to prove: $\{\neg(\varphi \vee \psi)\} \vdash \neg\psi$.

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 (Tree Proof Correctness). *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 both of the following conditions hold:*

1. *The root of the tree is ϕ.*
2. *Every open hypothesis in the tree is contained in Γ.*

Introduction Rules

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

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

Elimination Rules

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

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

Absurdity Rule

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

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

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

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

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

In this chapter, we explain how our algorithm works. It is divided into three sequential steps. The first step is to generate a hypergraph based on the initial goal (what the exercise asks us to prove) and the target goal (the part of the student's proof we want to complete), in order to store all possible rule applications for both goals. Then, we use this graph to build a second hypergraph that simulates as many proof constructions as possible by decomposing the target goal into sub-goals. In the last step, we trim this second graph so that it retains only valid and minimal proofs, either in terms of the number of steps required or the height needed to solve the problem. Finally, we use the resulting graph to extract and build readable proofs, which can later be used to generate feedback.

4.1 Transition Graph

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

To do this, we need both the initial goal and the target goal. In cases where we want a complete proof, we can assume that the target goal is the same as the initial goal. The initial goal is used to generate all the natural proof paths, which are proofs constructed using only formulas derived from decomposing the initial goal. The target goal, on the other hand, can sometimes be used to generate non-natural proof paths. These are proofs that rely on more complex formulas than those directly derived from the initial goal. This only happens when the part of the proof we want to complete shifts away from the initial goal. By considering both goals, the system is capable of generating more personalized and user-guided proofs, 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*
- $T_E \subseteq F \times \mathcal{P}(F \times F) \times R$ *is a finite set of* labeled hyperedges *called* Transition Edge (TE)*, where each edge consists of:*
  - *a tail formula* $f \in F$*,*
  - *a (possibly empty) set of pairs* $(f_1, f_2) \in F \times F$*, called* Transitions (T)*, where* f_1 *is the* hypothesis *and* f_2 *is the* closed hypothesis*, and*
  - *a rule label* $r \in R$*.*

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.

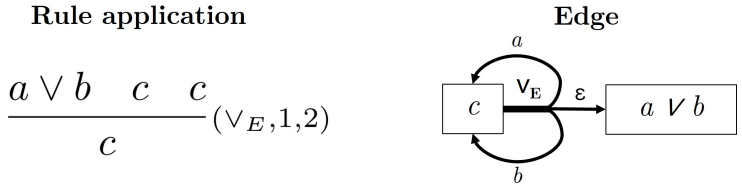

Fig. 3. Example of a transition edge.

We will have the transition: $T_1 = (\vee_E, \{(a \vee b, \varepsilon), (c, a), (c, b)\})$, and our TG will have this edge: $E_T = E_T \cup (c, \{T_1\})$. Our edges will always point from the conclusion to its hypotheses. If we are working on FOL proofs, we also need to consider the side conditions as part of the transition. The reason why we are using hypergraphs instead of regular graphs is that some rule applications can have more than one hypothesis, like in the example we showed and we want to map them in the graph.

Procedure:

1. First, we initialize our graph by defining the set of possible formulas F as the union of all formulas in the problem and the initial state. The set of rules R are the rules presented in Figure 1, and the edges E_T are initialized as an empty set.

2. We loop through all formulas in F, and for each formula f:

2.1. We add its negation $\neg f$ to F, but only if the expression was not already added as the negation of another expression (to avoid infinite loops). We include negations to support indirect proofs using the Absurdity rule.

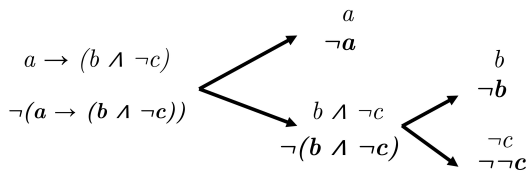

Fig. 4. Decomposition of the expression $a \rightarrow (b \wedge \neg c)$

2.2. If f is not a literal, we decompose it and add the decomposed formulas to F.

Figure 4 shows an example of the decomposition of the expression $a \rightarrow (b \wedge \neg c)$

So, at the end of this iteration, the set of formulas will be: $F = \{a \rightarrow (b \wedge \neg c), \neg(a \rightarrow (b \wedge \neg c)), a, \neg a, b \wedge \neg c, \neg(b \wedge \neg c), b, \neg b, \neg c, \neg\neg c\}$.

3. After computing the formulas that our proof can contain, we select which rules can be applied to each formula. To do that, we loop again through the formulas F, and for each formula f:

3.1. For each rule $r \in R$:

3.1.1. If the rule is Absurdity, we add a transition edge to the graph, as this rule can be applied to any formula.

3.1.2. Else if the rule is Elimination of Negation and $\neg f \in F$, we add a transition edge with that rule.

3.1.3. Else if f is a conjunction and the rule is Elimination of Conjunction, we add a transition edge with all matching formulas in F. The reason for this is that the Elimination of Conjunction rule can be applied to any formula when there is a disjunction, the same applies to the Elimination of Existential rule.

3.1.4. Else if f is not a literal and does not match the above rules, we try to match f with rule r using its outermost logical symbol. For example, if $f = a \rightarrow b$, it matches the Introduction and Elimination of Implication.

An example of the edges generated for the formula $a \rightarrow b$ is shown in Figure 5.

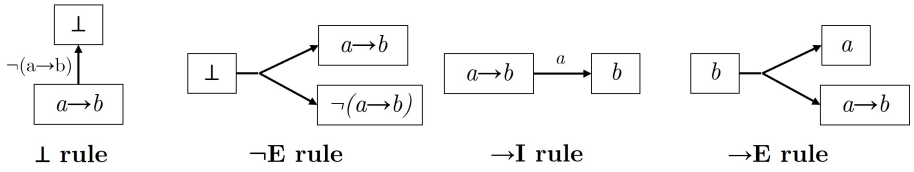

Fig. 5. Edges generated for the formula $a \rightarrow b$

As the algorithm considers the deviations taken by the formulas derived in the incomplete proof, it will be able to generate solutions that are more aligned with the student's reasoning process. Figure 6 shows the final TG for the problem $\vdash a \to a$, considering all possible formulas and transitions for this problem.

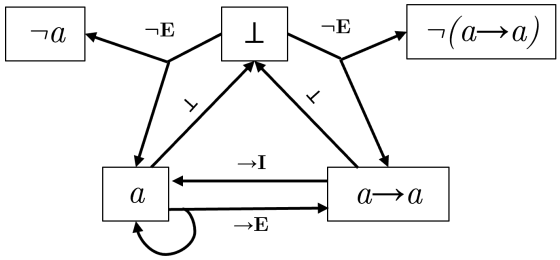

Fig. 6. Final TG generated from $\vdash a \to a$

4.2 State Graph

The second step is to build the State Graph (SG). To do this, we use the previous TG and the state we want to complete. This could be the original problem or a part of the proof the student did not finish. The SG is similar to the TG, but instead of storing transitions, it stores states. Before we describe the procedure, let us define some key terms:

Definition 5 (State Graph). *The SG is defined as a quadruple:*

$$SG = (\beta, TG, S_F, E_S),$$

where β is the initial state, TG is the Transition Graph, $S_F \subseteq \mathcal{P}(S)$ is the set of discovered states, and $E_S \subseteq F \times \mathcal{P}(SE)$ maps each formula $f \in F$ to a (possibly empty) set of state edges.

Definition 6 (State Edge). *A* state edge *is an element of the set*

$$S_E \subseteq R \times \mathcal{P}(S),$$

where R is the rule applied, and $\mathcal{P}(S)$ is the set of possible new states after applying that rule.

Definition 7 (Closed state). *A state $S = (\alpha, \Delta)$ is* closed *if $\alpha \in \Delta$.*

Procedure:

Algorithm 1: Transition Graph Construction

Input: Problem (Γ, ϕ), Incomplete State (Δ, α)
Output: Transition Graph $TG = (F, R, E_T)$

$F \leftarrow \Gamma \cup \Delta \cup \{\phi, \alpha\}$; `// Initialize formulas`
$E_T \leftarrow \emptyset$; `// Initialize edges`
$R \leftarrow$ set of inference rules ; `// Load rules`
`// Compute formulas`
foreach $f \in F$ **do**
  if *f was not already added as the 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**
  foreach $r \in R$ **do**
    if *r is the Absurdity rule* **then**
      $T_e = (r, \{(\bot, \neg f)\})$;
      $E_T \leftarrow E \cup \{(f, \{T_e\})\}$;
    else if *r is the Elimination of Negation rule and $\neg f \in F$* **then**
      $T_e = (r, \{(f, \varepsilon), (\neg f, \varepsilon)\})$;
      $E_T \leftarrow E \cup \{(\bot, \{(r, T_e)\})\}$;
    else if *r is the Introduction of Conjunction rule and f is a conjunction $(\psi \vee \phi)$* **then**
      foreach $f_{sub} \in F$ **do**
        $T_e = (r, \{(f, \varepsilon), (f_{\text{sub}}, \psi), (f_{\text{sub}}, \phi)\})$;
        $E_T \leftarrow E \cup \{(f_{\text{sub}}, \{(r, T_e)\})\}$;
    else if *f is not a literal and none of the rules above* **then**
      if *r matches f based on its main logical symbol* **then**
        Compute the transition edge and add it to the edge to the graph;

1. The algorithm starts by setting the initial state β. This can be the original problem state or the incomplete part of the student's proof. The TG is the one built before. The set of found states S_F starts with only β, and the set of edges E_S is empty.
2. Then, we go through each state $s = (\alpha, \Delta)$ in S_F:
   - **2.1.** If the state is closed, we skip it. There is no need to explore it further, because a valid mark can be assigned to it at this point. Other stopping conditions are needed to avoid very long executions, since the graph can grow very fast. For example, we can set a limit on the number of nodes explored.
   - **2.2.** Then we use the Transition Graph (TG) to find all rule applications for the formula α. This gives us a set of rule applications. For each transition:
     - **2.2.1.** We apply the rule to the current state s. If this creates a new state, we add it to S_F. The new state is always constructed based on the current state s, keeping track of the transformations made in previous states.
   - **2.3.** Finally, the algorithm combines the rule and the generated states into a state edge S_E, and adds it to the set E_S.

Figure 7 shows part of the graph for the problem $\vdash a \to a$. Nodes represent states. Solid borders are closed states and dashed borders are unclosed states. In this example, the state $\{\neg a, \neg(a \to a)\}$ cannot be closed because $\neg a \notin \{\neg(a \to a)\}$, and $\neg a$ has no outgoing edges in the TG (Figure 6).

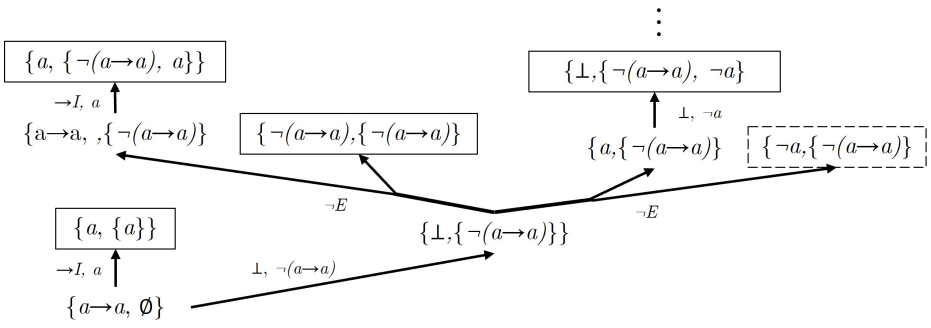

Fig. 7. Example of State Graph for $\vdash a \to a$

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

Algorithm 2: State Graph Construction

Input: Transition Graph TG, Initial State β
Output: State Graph $SG = (\beta, TG, S_F, E_S)$

$S_F \leftarrow \{\beta\}$; `// Initialize set of states`
$E_S \leftarrow \emptyset$; `// Initialize state edges`
`// Expand states`
foreach $s = (\alpha, \Delta) \in S_F$ **do**
  if *s is closed or s* **then**
    continue ; `// Skip closed state`
  if *stopping condition is reached* **then**
    break ; `// Avoid too many expansions`
  `// Apply rules from TG`
  Get list of rule applications T;
  foreach $(r, t) \in T$ **do**
    Compute new states S_r from s using transitions t;
    $S_E \leftarrow (r, S_r)$;
    $E_S \leftarrow E_S \cup \{(\alpha, \{S_E\})\}$; `// Add state edge`

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

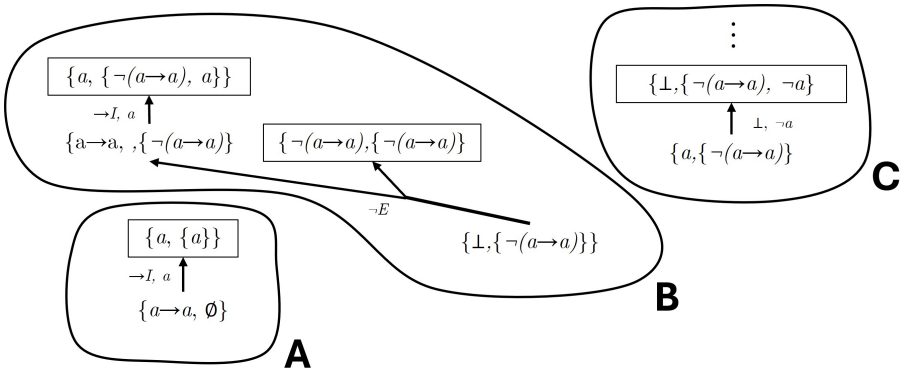

Fig. 8. Trimmed State Graph showing valid proof paths.

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 $\vdash a \to a$. 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 9 and 10 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 8. 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**).

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.

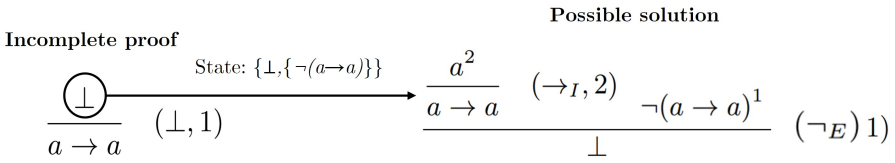

Fig. 9. Extracting a solution to produce feedback using an existing state

In this example, if the student removes the Elimination of Negation rule (one step back), we return to the situation previously presented.

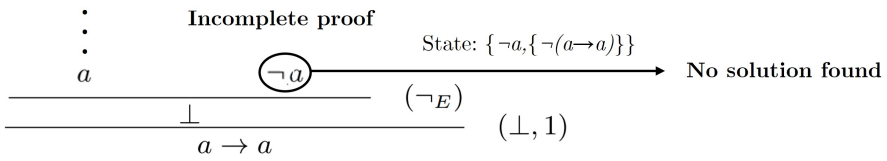

Fig. 10. Extracting a solution to produce feedback using a non-existing state.

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 over-complicating 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 [12] 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 [8]. 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 [7] 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 possi-

ble 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. Paula Gouveia and F Miguel Dionísio. Lógica computacional capítulo 1 -lógica proposicional.
5. Paula Gouveia and F Miguel Dionísio. Lógica computacional capítulo 3 -lógica de primeira ordem.
6. Phokion Kolaitis, Daniel Leivant, and Moshe Vardi. Panel: logic in the computer science curriculum. volume 30, pages 376–377, 01 1998.
7. 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.
8. Xuehan Maka Hu. Automatic generation of human readable proofs.
9. 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.
10. Francis Jeffry Pelletier. A brief history of natural deduction. *History and Philosophy of Logic*, 20(1):1–31, 1999.
11. 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.
12. David Robinson. Using resolution to generate natural proofs.
13. Jan von Plato. *Natural deduction*, page 31–63. Cambridge University Press, 2014.