DEPARTMENT OF
COMPUTER SCIENCE

DANIEL GONÇALVES FUSETA ROSA MACAU

BSc in Computer Science

INTERACTIVE TOOL FOR PRACTICING AND EVALUATING LOGIC EXERCISES

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
Draft: *January 10, 2025*

DEPARTMENT OF
COMPUTER SCIENCE

INTERACTIVE TOOL FOR PRACTICING AND EVALUATING LOGIC EXERCISES

DANIEL GONÇALVES FUSETA ROSA MACAU

BSc in Computer Science

Adviser: Ricardo Gonçalves
Assistant Professor, NOVA University Lisbon

Co-adviser: João Costa Secol
Associate Professor, NOVA University Lisbon

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
Draft: *January 10, 2025*

Abstract

Abstract en

Keywords: One keyword, Another keyword, Yet another keyword, One keyword more, The last keyword

Resumo

Resumo pt

Palavras-chave: Primeira palavra-chave, Outra palavra-chave, Mais uma palavra-chave, A última palavra-chave

Contents

List of Figures

Acronyms

DAM	Directed Acyclic Multigraph *(pp. v, 13, 14)*
FOL	First-Order Logic *(pp. 11, 16, 17)*

1 Introduction

Citing something online [3, 2].

1.1 Motivation

1.2 Problem formulation

1.3 Research

1.4 Document Structure

2

Background

2.1 Propositional Logic

Logic in general, is defined as the study of the principles of the reasoning. Propositional logic is a branch of logic that focus on the study of propositions and their relationships. The goal of logic in computer science is to create languages that help us represent situations we deal with as computer scientists. These languages allow us to think about and analyze these situations in a structured way. By using logic, we can build clear and valid arguments about these situations, ensuring they make sense and can be tested, defended or even carried out by a machine. [PAPER LogicInCS]

Proposition is a declarative statement that can either be true (denoted as T or 1) or false (denoted as F or 0), but not both. Propositions are the basic building blocks of propositional logic.

Examples of Propositions:

- "It is raining." (can be true or false)
- "The planet is round." (is true)
- "2 + 3 = 4" (is false)

Non-Examples of Propositions:

- "How was the meal?" (this is a question, not a declarative sentence)
- "Stop watching TV!" (this is an order, it can't be evaluated with true or false)

2.1.1 Syntax

To define a formal language, one must choose the alphabet of the language and establish the set of words that make up the language. These words are usually called formulas when the formal language is associated with a logic, as is the case here. The alphabet of the language is a set of symbols, and each formula is a finite sequence of symbols from the alphabet. Symbols are used to represent propositions and the relationships between them. By convention, propositions are represented by a single lower case letters (example p,q,r). The tables below list all the logical constants, propositional variables, and logical connectives in propositional logic.

Symbol	**Name**	**Example**
$\top$	Top	$\top$: "True"
$\bot$	Bottom	$\bot$: "False"
p,q,r	Propositions	p: "It is raining."

Table 2.1: Logical Constants and Propositional Variables

Symbol	**Name**	**Arity**	**Example**
$\neg$	Not	1	$\neg p$: "It is not raining."
$\wedge$	And	2	$p \wedge q$: "It is raining and it is cold."
$\vee$	Or	2	$p \vee q$: "It is raining or it is cold."
$\rightarrow$	Implication	2	$p \rightarrow q$: "If it is raining, then it is cold."
$\leftrightarrow$	Equivalence	2	$p \leftrightarrow q$: "It is raining if and only if it is cold."

Table 2.2: Logical Connectives

A well-formed formule (WFF) in Propositional Logic is defined recursively according to the following set of rules, which specify the conditions under which a formula is considered well-formed, and these rules build upon each other to allow for the construction of more complex logical expressions.

$$\begin{cases} \alpha \text{ is WFF} & \text{,if } \alpha \text{ is a proposition,} \\ \neg\alpha \text{ is a WFF} & \text{,if } \alpha \text{ is a WFF,} \\ (\alpha) \text{ is a WFF} & \text{,if } \alpha \text{ is a WFF,} \\ \alpha \wedge \beta \text{ is a WFF} & \text{,if } \alpha \text{ and } \beta \text{ are WFFs,} \\ \alpha \vee \beta \text{ is a WFF} & \text{,if } \alpha \text{ and } \beta \text{ are WFFs,} \\ \alpha \rightarrow \beta \text{ is a WFF} & \text{,if } \alpha \text{ and } \beta \text{ are WFFs,} \\ \alpha \leftrightarrow \beta \text{ is a WFF} & \text{,if } \alpha \text{ and } \beta \text{ are WFFs} \end{cases}$$

Examples of WFF:

- $\top$: "True" (Represents a tautology).
- $(p \wedge q) \rightarrow r$: "If it is raining and it is cold, then it is snowing."
- $(p \rightarrow q) \wedge (q \rightarrow r)$: "If it is raining, then it is cold, and if it is cold, then it is snowing."

Non-Examples of WFF:

- $p\vee$: Missing a second proposition after the disjunction.
- $\neg \wedge p$: Incorrect placement of negation with a binary operator ($\wedge$) without a second operand.
- $\neg(\wedge p)$: Missing a first proposition before the conjunction.

2.1.2 Equivalences

A key task in propositional logic is to determine if two formulas are logically equivalente (denoted as $\equiv$), that happens when both formulas have always the same truth values. There are several methods to check whether two logical formulas are equivalent:

- **Truth tables:** A truth table examines all possible combinations of truth values for the propositions involved. To construct a truth table, we need to list all propositions, enumerate all possible truth value combinations for those propositions, and evaluate the truth values of the formulas for each combination. If the results match in all rows, both formulas are equivalent. This combination can easily scale up depending on the number of propositions involved. For example, consider that you have n propositions, the total number of rows will be 2^n. This is not a feasible method for evaluating large expressions due to the exponential growth in the number of rows. However, for smaller expressions, truth tables are effective. The table below demonstrates $p \rightarrow q \equiv \neg p \vee q$.

p	q	$p \rightarrow q$	$\neg p \vee q$
T	T	T	T
T	F	F	F
F	T	T	T
F	F	T	T

- **Algebraic Manipulation:** This approach relies on a set of rules, known as logical equivalences, that can be used to manipulate expressions, similar to how it works in mathematics. By applying these equivalences, we can transform one logical expression into another that is logically equivalent, meaning both expressions have the same truth value in every scenario.

  - **Identity Law:** $p \wedge \top \equiv p$ and $p \vee \bot \equiv p$
  - **Domination Law:** $p \wedge \bot \equiv \bot$ and $p \vee \top \equiv \top$
  - **Idempotent Law:** $p \wedge p \equiv p$
  - **Negation Law:** $p \wedge \neg p \equiv \bot$
  - **Double Negation Law:** $\neg(\neg p) \equiv p$
  - **Commutative Law:** $p \wedge q \equiv q \wedge p$
  - **Associative Law:** $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$
  - **Distributive Law:** $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$ and $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$
  - **Absorption Law:** $p \wedge (p \vee q) \equiv p$
  - **Implication Law:** $p \rightarrow q \equiv \neg p \vee q$
  - **DeMorgan's Law:** $\neg(p \wedge q) \equiv (\neg p) \vee (\neg q)$ and $\neg(p \vee q) \equiv (\neg p) \wedge (\neg q)$

2.1.3 Natural Deduction

Another key task in propositional logic, is to determine whether a propositional expression is valid. An expression is considered valid if, for all possible interpretations, its truth value is always true. Regardless of the values assigned to the propositions, the final truth value must be always true. To determine if a sentence if valid, one can use a semantic or a syntatic approach. The semantic approach focuses on the meaning of the formulas in different interpretations, in contrast, the syntatic approach focuses on the manipulation of symbols using a deductive system. [PAPER LCGOUVEIADIONISIO] Natural deduction is a type of deduction system that uses rules of inference. By applying proof rules to the premises (φ_1), we hope to get some more formulas, and by applying more proof rules to those, to eventually reach the conclusion (ψ). [PAPER LOGICINCS] This can be denoted by:

$$\begin{array}{ll} \varphi_1, \varphi_2, \ldots, \varphi_n \vdash \psi & \text{With premises} \\ \vdash \psi & \text{Without premises} \end{array}$$

There are many styles to represent these proofs. For example, the Fitch-style uses a linear structure, with deeper indentation levels to represent assumptions or intermiate steps in the proof, while Tree-style organizes the proof in a tree-shaped structure. This thesis will focus on the Tree-style representation. The following schema illustrates an example of a proof for $\vdash \neg(\varphi \vee \psi) \rightarrow \neg\varphi$ using the Tree-style representation.

$$\dfrac{\dfrac{\dfrac{\neg(\varphi \vee \psi)^1 \qquad \dfrac{\psi^2}{(\varphi \vee \psi)}\,(\vee I_l)}{\bot}\,(\neg E)}{\neg\psi}\,(\neg I, 2)}{\neg(\varphi \vee \psi) \rightarrow \neg\varphi}\,(\rightarrow I, 1)$$

These tree-shaped structures represent proofs and are composed of sequences of inference rules, identified on the right-hand side of the fractions. The trees are constructed from nodes, which are formulas. The formulas at the leaves are called hypotheses and are associated with marks (numbers). The formula at the root is the conclusion of the proof. We call closed hypothesis if its mark is used in a rule of the tree otherwise it is said to be open.

There are many ways to build these proofs: you can go bottom-up by starting from the conclusion, top-down by starting from the presmisses or open clauses, or you can do both at the same time.

Considering a bottom-up solution the first step is to apply the Implication Introduction rule ($\rightarrow I, 1$) given the fact that the conclusion has an implication as an outter operation. By doing so we add the left part of the implication as a hypothesis. This premisse can be useful to help us solving the deduction. Then we applied Negation Introduction rule ($\neg I, 2$), and we add ψ to our list of premisses. After that we applied the Negation Elimination rule ($\neg E$), on the left hand we've added the hypothesis derived from the instroduction of implication ($\neg(\varphi \vee \psi)$), and on the right hand we've added the same expression but without negation ($\varphi \vee \psi$). Finally we applied the Disjunction Introduction left ($\vee I_l$) using the premisse derived from the introduction of negation.

Each rule has it own characteristic and can only be applied under certain circumstances. Some add new hypotheses that must be closed and others not. We say that the proof is finished when all the hyphotehses that must be closed are closed and the root of the tree contains the desired conclusion. Here is the complete list of rules in propositional logic.

[FAZ SENTIDO COLOCAR TODAS?? TENHO QUE EXPLICAR UMA A UMA?]

$$\frac{\begin{matrix}\mathcal{D}_1 & \mathcal{D}_2\\ \varphi & \psi\end{matrix}}{\varphi \wedge \psi}\ (\wedge I)$$

Conjunction Introduction

$$\frac{\begin{matrix}\mathcal{D}\\ \varphi \wedge \psi\end{matrix}}{\varphi}\ (\wedge E_r)$$

Conjunction Elimination, right

$$\frac{\begin{matrix}\mathcal{D}\\ \varphi \wedge \psi\end{matrix}}{\psi}\ (\wedge E_k)$$

Conjunction Elimination, left

$$\frac{\begin{matrix}\mathcal{D}\\ \varphi\end{matrix}}{\varphi \vee \psi}\ (\wedge I_r)$$

Disjunction Introduction, right

$$\frac{\begin{matrix}\mathcal{D}\\ \psi\end{matrix}}{\varphi \vee \psi}\ (\wedge I_l)$$

Disjunction Introduction, left

$$\frac{\begin{matrix} & [\varphi_1]^m & [\varphi_2]^n\\ \mathcal{D}_1 & \mathcal{D}_2 & \mathcal{D}_3\\ \varphi \vee \psi & \psi & \psi\end{matrix}}{\psi}\ (\vee E, m, n)$$

Disjunction Elimination

$$\frac{\begin{matrix}\mathcal{D}_1 & \mathcal{D}_2\\ \varphi & \varphi \rightarrow \psi\end{matrix}}{\psi}\ (\rightarrow E)$$

Implication Elimination

$$\frac{\begin{matrix}[\varphi]^m\\ \mathcal{D}\\ \varphi\end{matrix}}{\varphi \rightarrow \psi}\ (\rightarrow I, m)$$

Implication Introduction

$$\frac{\begin{matrix}\mathcal{D}_1 & \mathcal{D}_2\\ \varphi & \neg\varphi\end{matrix}}{\bot}\ (\neg E)$$

Negation Elimination

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

Negation Introduction

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

Contradiction

2.2 First-Order Logic

Another branch of logic is First-Order Logic (FOL), also known as predicate logic. Unlike propositional logic, which focuses solely on simple declarative statements, first-order logic extends this by introducing quantifiers and predicates, variables, constants and functions. This additional components allows us to express more complex declarative sentences, capturing relationships between objects and their properties in a specified context. [LOGICINCS]

Examples of First-Order:

- "There exists a black cat that likes baths."
- "John is a friend of Mary."
- "If a variable is a integer and positive, then it is greater than zero."

2.2.1 Syntax

First-Order Logic uses the same syntax as propositional logical but adds more features to turn it more expressive. It introdcues quantifiers that allow us to generalize or specify expressions, making it possible to express universal truths or existencial statements. Quantifiers also enable the introduction of variables within a certain domain. Predicates that are used to express properties or relationships, allowing FOL to capture facts about objects and their interactions. They are always denoted with a capital letter, return a truth value and can have different arities. Similar to predicates, FOL includes functions that represent mappings or computations. Functions are represented with lowercase letters and their return value is a specific value in the domain.

Symbol	**Name**	**Example**
x, y, z	Variables	x: "An individual object"
$black$	Constants	$black$: "Value that is fixed"
$Cat(x)$	Predicates	$Cat(x)$: "True if x is a cat"
$color(x)$	Functions	$color(x)$ = black: "The color of x is black"

Table 2.3: Examples of variables, constants, predicates, and functions in FOL

Symbol	**Name**	**Example**
$\forall$	Universal	$\forall x\,(Cat(x) \to Mammal(x))$: "All cats are mammals"
$\exists$	Existential	$\exists x\,(Cat(x) \wedge color(x) =$ black$\wedge LikesBaths(x))$: "There exists a black cat that likes baths"

Table 2.4: Quantifiers in FOL

We can extend the definition of a well-formed formula (WFF) from Proposition Logic to represent a well-formed formula in First-Order Logic. To do this, we first need to introduce a new concept called a term. A term is an expression that represents a specific value in the domain. The schema below shows a simplified version of how to define a WFF in FOL, considering terms, comparisons and mathematical operation between terms. [FIRST COURCE IN LOGIC]

$$\begin{cases} c \text{ is a term} & \text{, if } c \text{ is a constant,} \\ x \text{ is a term} & \text{, if } x \text{ is a variable,} \\ t_1 \{+,-,\times,/\}\ t_2 \text{ is a term} & \text{, if } t_1 \text{ and } t_2 \text{ are terms.} \\ f(t_1,t_2,\dots,t_n) \text{ is a term} & \text{, if } f \text{ is a function and } t_1,t_2,\dots,t_n \text{ are terms.} \end{cases}$$

$$\begin{cases} P(t_1,t_2,\dots,t_n) \text{ is a WFF} & \text{, if } P \text{ is a predicate and } t_1,t_2,\dots,t_n \text{ are terms,} \\ \neg\alpha \text{ is a WFF} & \text{, if } \alpha \text{ is a WFF,} \\ (\alpha) \text{ is a WFF} & \text{, if } \alpha \text{ is a WFF,} \\ \forall x\ \alpha \text{ is a WFF} & \text{, if } \alpha \text{ is a WFF and } x \text{ is a variable,} \\ \exists x\ \alpha \text{ is a WFF} & \text{, if } \alpha \text{ is a WFF and } x \text{ is a variable,} \\ t_1 \{=,\neq,<,\leq,>,\geq\}\ t_2 \text{ is a WFF} & \text{, if } t_1 \text{ and } t_2 \text{ are terms,} \\ \alpha \{\wedge,\vee,\rightarrow,\leftrightarrow\}\ \beta \text{ is a WFF} & \text{, if } \alpha \text{ and } \beta \text{ are WFFs,} \end{cases}$$

2.2.2 Equivalences

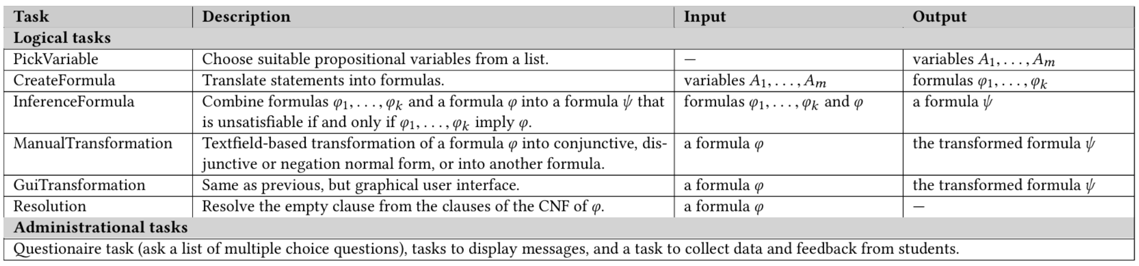
2.2.3 Natural Deduction

3 Related Work

In this section... TODO: Falta falar sobre MOOCs/MOOEPs e Feedback

3.1 Iltis Web-Based System for Teaching Logic

Iltis is an interactive online tool that assists students in learning logic from its foundation. [3, 2] The goal of this tool is to provide a system that supports a wide variety of content (propositional logic, modal logic, and first-order logic), along with a valuable feedback system that helps the learner better understand their mistakes. The developers of this web application divided it into multiple sections. Each section consists of a series of tasks, or exercises, that intensify in difficulty as the learner progresses through them. For each kind of task, this application provides a custom feedback generator. Feedback generators are pre-implemented pieces of code that dynamically provide various forms of feedback in tasks. This feedback can vary depending on the mistakes made by the learner. Some tasks have different levels of feedback that may differ based on the learner's proficiency. Low feedback levels provide a vaguer hint, and the high ones a more precise and explicit hint. The image below provides a list of the currently available types of exercises in Iltis.



Task	**Description**	**Input**	**Output**
Logical tasks			
PickVariable	Choose suitable propositional variables from a list.	–	variables $A_1, \ldots, A_m$
CreateFormula	Translate statements into formulas.	variables $A_1, \ldots, A_m$	formulas $\varphi_1, \ldots, \varphi_k$
InferenceFormula	Combine formulas $\varphi_1, \ldots, \varphi_k$ and a formula φ into a formula ψ that is unsatisfiable if and only if $\varphi_1, \ldots, \varphi_k$ imply φ.	formulas $\varphi_1, \ldots, \varphi_k$ and φ	a formula ψ
ManualTransformation	Textfield-based transformation of a formula φ into conjunctive, disjunctive or negation normal form, or into another formula.	a formula φ	the transformed formula ψ
GuiTransformation	Same as previous, but graphical user interface.	a formula φ	the transformed formula ψ
Resolution	Resolve the empty clause from the clauses of the CNF of φ.	a formula φ	–
Administrational tasks			
Questionaire task (ask a list of multiple choice questions), tasks to display messages, and a task to collect data and feedback from students.			

Figure 3.1: List of tasks available in Iltis [2].

From the teacher's perspective, this framework provides a way to create more tasks. Teachers can achieve this by creating an XML file where they specify a set of tasks and a list of feedback generators to be presented to the learner.

3.1.1 Feedback

Feedback generators comprise the Iltis feedback system. Teachers are allowed to associate more than one feedback generator with the task, creating different levels of feedback. Some exercises rely on feedback generators constructed using reversion rules, providing better and more accurate feedback. Reversion rules were built based on a previous study, where researchers collected some of the most frequent mistakes made by learners and built a list of reversion rules. A common example of a reversion rule in the "Propositional Formulas" exercises is to switch the order of the antecedent and consequent in implications. Whenever a learner switches two parameters, the feedback generator tries to apply reversion rules to find the correct solution. If successful, this indicates that the solution is close to the correct one, making it possible to provide more precise feedback based on the applied rule(s). Otherwise, it suggests that the solution is far from the correct one.

3.1.2 Conclusion

There are some positive aspects to consider from this system when developing our own tool, such as the intuitive way (it presents a low learning curve, and it is fundamental for these kinds of tools) that the exercises are presented to the learner, the advanced feedback system, and the simple access to the tool. It also provides a vast set of exercise types and a modular way to create them. On the other hand, teachers need to specify tasks in XML, and this requires some extra knowledge. Some types of exercises are still missing in this tool, like the deduction tree proof. Since it was developed by a German university and is not open-source, it cannot be expanded.

3.2 Logic4Fun

Logic4Fun is an online tool with a wide range of logical problems and puzzles focused on logical modeling and formalization ([1]).

This tool has been under development since 2001 by an Australian university. It was projected to help students practice and develop skills in formalizing logical problems, as this is a challenging topic to teach, and learners often struggle with it.

Logic4Fun uses many-sorted first-order logic (MSFO) [1] language to express the problems. It has a solver that takes as input a set of formulas and searches for finite models of this set. This tool presents a web page with different levels of problems: Beginner, Intermediate, Advanced, Expert, and Logician, with increasing complexity. It starts with trivial exercises to help students better understand how to use the site (declare vocabulary, set constraints, and read the solver output) and progresses to more complex and challenging exercises that require a strong background in logic. One of the key advantages of

[1]Many-sorted first-order logic is an extension of First-Order Logic (FOL). In FOL, all variables come from the same domain, limiting flexibility when modeling exercises with multiple distinct domains. MSFO extends this by allowing the assignment of types (or sorts) to variables and predicates, making the language more expressive.

using this tool is the ability to receive immediate and accurate feedback, in contrast with traditional teaching methods. This helps keep learners motivated and encourages them to invest more time and effort into solving problems. This site also allows users to enroll in a course by using the credentials provided by the teacher.

3.2.1 Feedback

Logic4Fun tracks two kinds of errors: syntactic and semantic. Syntactic errors are mistakes in the structure or arrangement of words that violate the grammar rules of the language. These errors can be captured by the parser or type checker. When a user attempts to submit an exercise with syntactic errors, a message is presented with some suggestions. When the type checker detects an error, it provides more information, especially about the expected and given types. Semantic errors are mistakes in logic or meaning in a programming language that occur in program execution. Since there are no predefined solutions, these errors are harder to classify and to deal with. Given these difficulties, Logic4fun created a diagnostic tool to provide more informative responses to the users when a solution cannot be found. The diagnostic tool uses two approaches to provide information: approximate models and unsatisfiable cores.

- In the approximate models approach, the solver starts by marking some unsatisfied constraints as "soft" and then attempts to satisfy as many as possible. Then a user can adjust constraints and rerun the solver, iterating to find the optimal approximations.

- In the unsatisfiable cores approach, the solver can try to identify groups of unsatisfied constraints that are causing the problem. Each group must contain at least one contradiction, giving useful clues for troubleshooting the problem.

3.2.2 Conclusion

Logic4Fun has several positive aspects, such as allowing exercises to be saved, enabling learners to pause their work and resume it later, progressively increasing the difficulty of exercises, helping learners integrate with the tool, and incorporating a diagnostic tool to address the lack of feedback. It has a class system where professors can invite their students to enroll. However, it only provides a restricted range of exercises based on first-order logic. It has some limitations with the solver's performance (the number of models presented to the user is restricted), and it is still facing issues with feedback. Sometimes, the reported errors are overly detailed or unclear, which can become frustrating for the learners.

3.3 LOGAX

LOGAX is a tool designed to help students in constructing axiomatic logic proofs[2]. in Hilbert style. [paper GENERATION andUseOfHintsandFeedback] This tool is capable of providing feedback and hints at different levels of the proofs. It can provide the solutions for the steps to follow, as well as the complete solution to the problem.

Students are not strictly required to solve the exercises in one way. This tool allows proofs to be proven in both directions (bottom-up, top-down), as described in 2.1.3. In deduction, there can be multiple possible solutions to the same problem. LOGAX can adjust the solution space to better assist learners based on their path. This algorithm for adjusting the solution is based on the user's steps. If the user takes a step in the current solution space, LOGAX can give feedback and hints directly. However, if the step diverges from the solution space, it is necessary to recalculate it, and then the system uses the new solution space to compute feedback and hints. This dynamic behavior enables the system to consistently provide feedback and hints to the user, preventing them from becoming lost in the exercise.

LOGAX has a system capable of automatically generating multiple possible solutions for a given proof. The algorithm is an adaptation of the one created by Bolotov ([Bolotov's Algorithm]). Bolotov's algorithm constructs a directed acyclic multigraph (DAM) from a list of premises and a conclusion. This data structure can capture all possible steps throughout the proof that can lead to the conclusion. In this DAM, vertices represent statements and edges connect dependent statements (rule application). An example of a generated DAM in LOGAX is shown in imageX, where three assumptions are given (p, $p \rightarrow q$ and $q \rightarrow r$) and the goal is to prove $q \rightarrow r \vdash (p \rightarrow q) \rightarrow (p \rightarrow r)$. There are solid red edges that show how to use Modus Ponens (which is the same thing as the Implication Elimination rule) and dashed blue edges that show how to use the Deduction Theorem (which is the same thing as the Implication Introduction rule). The statements $(q \rightarrow r) \rightarrow (p \rightarrow (q \rightarrow r))$ and $p \rightarrow (q \rightarrow r) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r))$ are axioms. This generated DAM captures three different solutions for the same proof: one that uses axioms a and b, one that uses the Deduction Theorem and axiom a and one that uses no axioms and applies the Deduction Theorem twice.

Storing this information in a DAM makes it easier to provide feedback about the steps required to complete the proof at any given level. The user can apply the rules in any order they choose. The system can adapt to the user's solution if it diverges, simply by following the sequence of rules chosen by the user. This allows the system to easily provide information about next steps (top-down proof) or previous steps (bottom-up proof) using the edges. To provide a complete solution to the user, it is necessary to extract and trim the solutions from the DAM.

The design of this tool focuses on interfaces that allow students to concentrate on the

[2] Axiomatic proofs are a kind of proof in formal deduction where each step of the proof is supported by axioms or inference rules previously established.

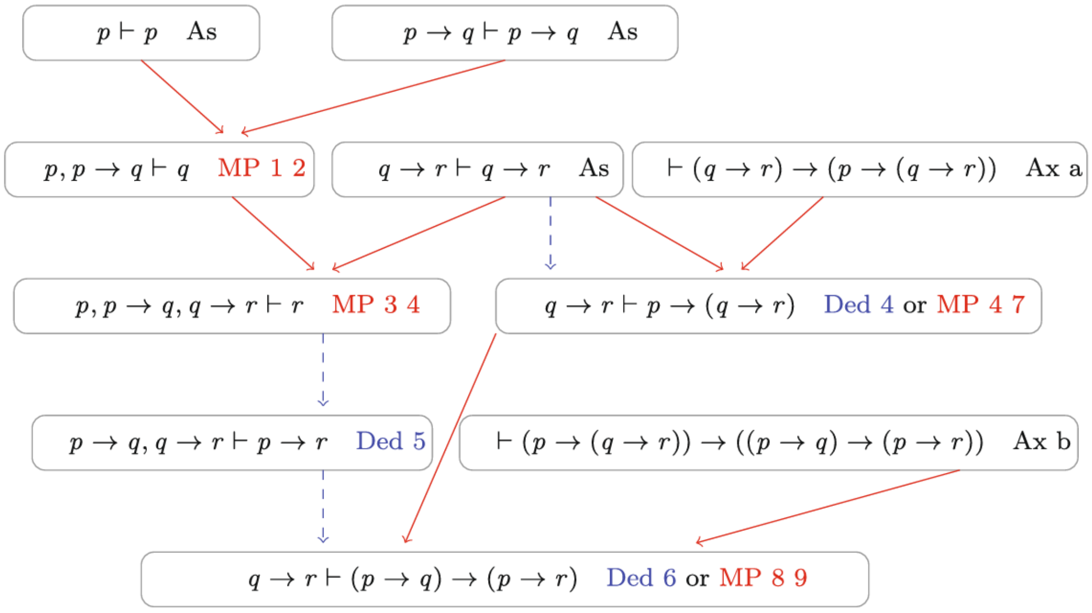

$p \vdash p$ As
$p \to q \vdash p \to q$ As
$p, p \to q \vdash q$ MP 1 2
$q \to r \vdash q \to r$ As
$\vdash (q \to r) \to (p \to (q \to r))$ Ax a
$p, p \to q, q \to r \vdash r$ MP 3 4
$q \to r \vdash p \to (q \to r)$ Ded 4 or MP 4 7
$p \to q, q \to r \vdash p \to r$ Ded 5
$\vdash (p \to (q \to r)) \to ((p \to q) \to (p \to r))$ Ax b
$q \to r \vdash (p \to q) \to (p \to r)$ Ded 6 or MP 8 9

Figure 3.2: Example of a DAM generated by LOGAX.

goal of solving proofs rather than waste time figuring out syntactic errors. Shortening the number of steps and distractions required to reach the goal leads to better learning outcomes.

3.3.1 Hints & Feedback

Before developing the system, the developers of LOGAX explored two different approaches to constructing guidance and providing feedback for a proof. The first approach entails the teacher providing a hidden solution or deriving it from a set of student solutions. However, this method has some drawbacks: it can only recognize solutions that are nearly identical to the stored proofs, it is limited to a fixed set of exercises, and every time a teacher wants to add a new exercise, they must provide a hidden solution. The second approach relies on Bolotov's algorithm, which addresses all the issues of the previous method.

After that, they began by developing a basic hint system that includes hints to: provide directions for the next steps (forward or backward), indicate the next rule to be applied, and offer an explicit step-by-step procedure for performing the next step. Then, after some studies, they realized that students exposed to informative feedback along with information about a subgoal are more likely to succeed in correcting mistakes than students exposed to only informative feedback. By providing feedback that includes information about subgoals, the system can help students understand why a certain step is useful. Consequently, they expanded the system to track a list of subgoals, specifically sub-proofs, during the proof algorithm's execution. This way, it can provide hints about subgoals at any level of the proof.

They also did some studies on students' common mistakes for this kind of proof, and they came up with the following types of mistakes:

- Oversights: Correspond to syntactic errors, for example, when a user forgets to close a parenthesis in a sentence or when logical symbols are not placed in the correct position.
- Conceptual errors: These occur when a student fundamentally misunderstands a concept or its application. For example, this error can occur when choosing the statements to apply a certain rule.
- Creative rule adaptations: These occur when students try to invent their own rules. For example, from this $\neg p \rightarrow (q \rightarrow \neg r)$ and q, we can conclude $\neg p \rightarrow \neg r$ using Modus Ponens (Implication Elimination rule). This usually happens when the student does not know how to proceed.

Finally, they expanded the system to also track those mistakes. This way, the system can point out a mistake and, if possible, mention exactly which formula, subformula, or set of formulas does not match the chosen rule.

3.3.2 Conclusion

For the deduction tree exercises (REF deduc tree exercises), LOGAX appears to be a perfect fit. It covers a wide range of aspects to consider when developing a system like this. Starting with the algorithm that finds all the possible solutions for a given proof, the fact that it ignores the order of the steps, and the ability to adjust the guidance based on the user's solution. It also includes different approaches for giving feedback and hints, as well as the idea of giving subgoals as hints to help the student understand the proof. Additionally, there are some important aspects regarding the design of the interfaces.

Unfortunately, this is an old project, and the tool is no longer available. This tool has some minor drawbacks that we can consider when developing our solution. One of them is that the system doesn't provide fading strategies to reduce the amount of feedback. We might want to control the amount of feedback sent to the student based on their level of expertise. Another problem this tool faces is that the user can't erase lines of the proof. As a consequence, the final proof can be more extensive than expected. Overall, it seems to meet all the requirements for a good tutoring tool, and for sure, this tool will be used as a reference for the one that we are going to develop.

3.4 MineFOL

MineFOL is a game for learning First Order Logic [paper do minefol]. This game is very similar to a well known game called Minesweeper. Minesweeper is a game that features a grid containing empty cells and cells with hidden mines. The object is to locate all the hidden mines using hints provided as the player explores the grid. These hinst indicate the number of mines adjacent to a given cell, helping the player deduce their locations. MineFOL is an adaptation of that game where, instead of giving hints with the number of the nearest mines it gives messages with FOL expressions to help locating them for example: $\neg\exists x\,\text{mine}(1, x)$. These expressions are hidden in safe squares, and the goal is to find the maximum number of expressions to have enough information to locate all mines in less steps possible. The player can only travel trought safe squares, a square is considered safe if it is possible to prove that using the collected messages. If the player steps out of the proven zone the game ends. The game always starts at the top left corner of the grid($cell(1, 1)$) with a initial expression and players have information, about how many mines are there and the maximum number of moves to solve the problem. That information can be combined with the collected expressions to derive more usefull information. An example of a game is shown in image X, where the goal is to find one mine in 35 movements.

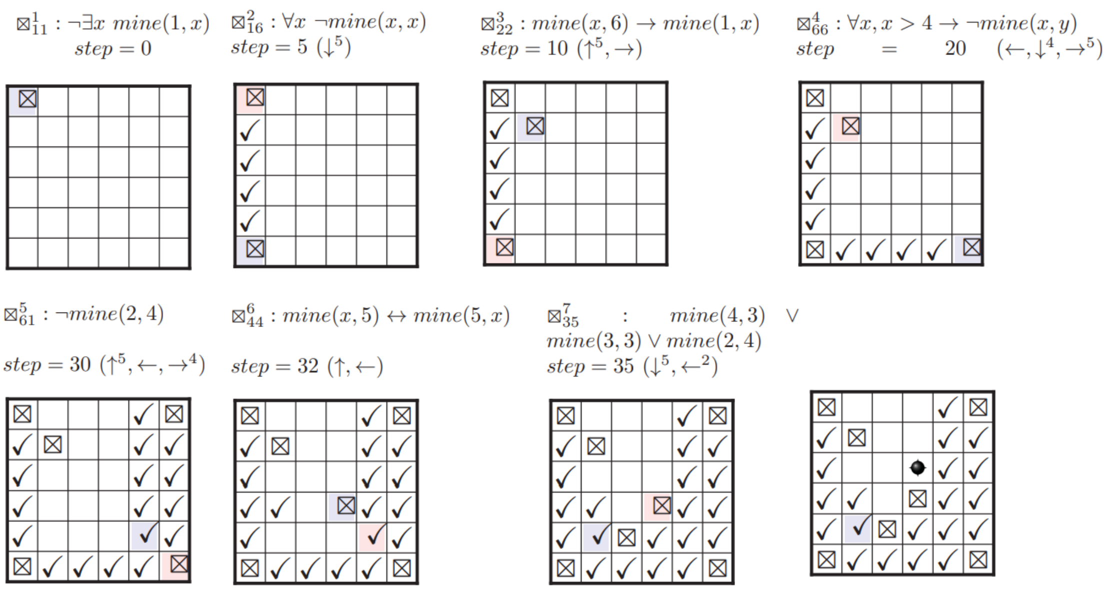

$\boxtimes^1_{11} : \neg\exists x\ mine(1, x)$
$step = 0$
$\boxtimes^2_{16} : \forall x\ \neg mine(x, x)$
$step = 5\ (\downarrow^5)$
$\boxtimes^3_{22} : mine(x, 6) \rightarrow mine(1, x)$
$step = 10\ (\uparrow^5, \rightarrow)$
$\boxtimes^4_{66} : \forall x, x > 4 \rightarrow \neg mine(x, y)$
$step = 20\ (\leftarrow, \downarrow^4, \rightarrow^5)$
$\boxtimes^5_{61} : \neg mine(2, 4)$
$step = 30\ (\uparrow^5, \leftarrow, \rightarrow^4)$
$\boxtimes^6_{44} : mine(x, 5) \leftrightarrow mine(5, x)$
$step = 32\ (\uparrow, \leftarrow)$
$\boxtimes^7_{35} : mine(4, 3) \vee mine(3, 3) \vee mine(2, 4)$
$step = 35\ (\downarrow^5, \leftarrow^2)$

Figure 3.3: MineFOL example with one mine and 35 movements. Blue squares represent the player's current position, and the red ones represent the previous position. Cells with: ⊠ contain a message and ✓ represent safe squares.

MineFOL has three different modes:

- Play it yourself!: In this mode the user tries to locate all mines based on a set of decovered FOL expressions. This can help the student practice reasoning in FOL, since the student has to translate the expressions to natural language to unserstand the message in order to solve the game.
- Challenge a software agent!: In this mode is the computer responsible to solve the game. The student doesn't need to worry about anything. If the game is valid, the agent will find a solution for that game and it will present it to the student.

- Create your own game!: In this mode is the user responbile to setup the game. Students can define their own FOL expressions and respective cells, as well as, the size of the rid and the number of mines. The complexity of the game can vary based on the grid size, ther number of mines, the size and complexity of the FOL expressions, and the number of steps required to complete the game. This mode is good for students to practice formalization in FOL. To check if the game is valid (can be solved), the student can challenge the software agent.

3.4.1 Feedback

This game includes a feedback system that provides feedback every time the player loses. For example, if the student steps on a mine cell, the system explains why the mine is there by displaying a resolution-based proof.

3.4.2 Conclusion

MineFOL introduces a new concept that none of the preivouly tools described has. The concept of gamification, this consists in applying game mechanincs and concepts on non-gaming environments. Studies shows that if the learning platform is gamified, it does not only drastically increase the user enrolment but also increase user engagement throughtout the course (ref GAMIFICATION OF MOOCS FOR INCREASING USER engagement). Using MineFOL, students delevolp skills in reasoning an formalization while playing the game. It also allows competitivity and creativity between students by creating and challenging other students to complete their own games. The game also has different levels of complexity, allowing students with less experience to have a chance to learn, while more advanced students can challenge themselves with harder levels to further improve their skills.

4 Proposed Work

4.1 Technologies

4.1.1 Architecture

4.2 Proposed Exercises

The exercises I am proposing are divided into two branches of Logic: Propositional Logic (2.1) and First-Order Logic (2.2). Generally, courses that teach logic separate the subject into these two branches, so the idea behind this decision is to cover exercises for both parts of the subject. This way, students will have material to study and to be evaluated on both parts of the course. For each branch, I plan to implement two types of exercises that are similar in both branches. One will involve converting natural language into a formula in propositional/first-order logic, and the other will focus on making deductions in a tree-shaped format. Deduction tree proofs are the kind of exercises where students struggle the most, so i also took that into consideration selecting the exercises. The proposed exercises serve as a guide for planning the implementation. If there is enough time, there may be an opportunity to implement more exercises. The focus of the exercises will not only be on the correctness of the provided solution, but also on offering feedback to help the student better understand what they are doing. Finding the right way to give feedback to the student is challenging because we don't want to provide too much information, but at the same time, we don't want the student to feel lost. [REF para o topico de feedback] The following sections will discuss the different types of exercises, how they work, how they can be presented to the final users, and various ways to provide the appropriate level of feedback.

4.2.1 Propositional Logic

4.2.1.1 Transforming a sentence from natural language to propositional logic

This type of exercise essentially involves translating a declarative sentence in natural language into propositional logic. For example, consider the following propositions:

$$p : \text{It is raining}, \quad q : \text{It is cold.}$$

Now, imagine the exercise asks you to write "It is raining and it is cold" in propositional logic, using the propositions defined above. A correct answer would be: $p \land q$. Another example could be "If it is cold, it is raining," which can have multiple correct answers, such as: $q \rightarrow p$ or $\neg q \lor p$. While developing this exercise we need to create a parser to check weather the expression is well formed or not (2.1.1). We also have to consider that there may be more than one possible anwser for the same question so it would be necessary to create a system to check equivalences between expressions like the ones described in 2.1.2.

Design: The Graphic User Interface (GUI) for this exercises will include a list of propositions written in natural language, with each sentence associated with a corresponding letter (literals). The user will then be presented with multiple exercises based on the same set of propositions, each one with increasing complexity. Below each exercise, an input box will be available for the user to enter their anwser. To assist the user in typing sentences using propositional symbols, a custom keyboard will appear when the input box is clinked. This layout is inspired by the one used in Iltis(3.1). It is user-friendly, does not require extensive knowledge to understand how it works, and has also been tested by numerous students.

Feedback: The feedback of this exercise can be divided in two types: syntatic feedback (the given sentence is not well formed) and semantic feedback (the given sentence is not equivalent to the correct). Syntatic feedback can be detected at the parser level. We can provide precise informations about what's wrong, such as indicating the column where the error occured or specifying that a symbol is missing. Alternatively, we can take a more general approach by giving a subexpression and stating that something is incorrect without specifying teh exact issue. Finally, we can simply state that the the expression contains syntatic errors, leaving it up to the user to determine what might be wrong. Semantic feedback can be verified after checking for syntatic errors. This can also be devided into multiple levels of feedback. We can provide a translation of the input sentence into natural language and leave the user to interpret the difference between what the question is asking and the answer they provided. Another option involves using reversion rules (3.1.1) where, in some cases, we can provide the exact mistake made by the user and generate more accurate feedback based on that. Alternatively, we can display part of a truth table where the values don't match, stating that the literals under these values should have a specific truth value.

4.2.1.2 Deduction trees in propositional logic

Proofs in tree-shaped representation are another type of exercises in logic, the goal of these exercises is to derive the conclusion logically from the premisses using a set of rules. For example, consider the following sequent:

$$\vdash \neg(\varphi \vee \psi) \rightarrow \neg\varphi$$

A possible solution could be the one presented in 2.1.3. As in the transforming natural language into proposicional logic exercises, in this one there are also many possible solutions for the same exercise. For example, the order of the rules can be different, so as the number of rules applyied and this can create different tree-shapes for the same problem. We also need to create a parser to check the rules provided we also need to use the typechecker for propositional logic to check weather the expressions introduced in the rules/premisses are correct or not.

Design: There design of this exercise will consist in building block proof, where each block represents a rule or an expression that then can be merged and unmerged into a rule, this way creating the final tree-shaped proof. By having draggable blocks we can start from any point of the tree, we are not fixed to solve it always the same way. This can help students to progress in their proofs, because sometimes some steps are not obvisous if, for example, we follow a bottom up solution we have to predict some further steps. For someone with experiance it will be fine but this project will be designed for students that are starting to learning how to create proofs. As in the previous exercise, in this one the user also have input boxes to write logical expressions. Each inputbox will be followed by a keyboard with propositional symbols to help the student.

Feedback: We have multiple ways to provide feedback for this kind of exercises. Syntatic feedback, will check if the inputed expressions are well formed as well as the applied rules. Semantic feedback, that will check if the proof really prove what the exercise wanted to be prove. We can add highlight places where you can merge expressions when dragging the blocks. The system can pre-fill some input boxes after selecting a rule, the system can also restrict the set of rules, listing only the possible ones based on the last rule applied. The system can guide the user based on a hidden solution, by giving hints for the folowing steps. In some cases (during evaluations), we want to leave the user complete free to do whatever he wants, for example to create their own rules, to apply rules that doesnt make sence etc.. But in onther cases we want to help the student to understand what he is doing, and we can use some of the strategies prevously mention to, based on the level of experiance of the student, generate a good feedback.

4.2.2 First-Order Logic

4.2.2.1 Transforming a sentence from natural language to propositional logic

Exercise description **Design:** **Feedback:**

4.2.2.2 Deduction trees in propositional logic

Exercise description **Design: Feedback:**

4.3 Work Plan

Bibliography

[1] https://users.cecs.anu.edu.au/~jks/papers/L4F.pdf. [Accessed 13-11-2024] (cit. on p. 11).

[2] *arxiv.org*. https://arxiv.org/pdf/1804.03579. [Accessed 13-11-2024] (cit. on pp. 1, 10).

[3] *Iltis: Learning Logic in the Web — arxiv.org*. https://arxiv.org/abs/2105.05763. [Accessed 13-11-2024] (cit. on pp. 1, 10).