UNIVERSITY OF BUCHAREST

FACULTY OF PHILOSOPHY

MASTER's THESIS

Computing Philosophical Logics

Developing an Automated Proof Calculator for Propositional and Quantified, Classical and Non-Classical Logics

Student: ANDREI DOBRESCU

Supervisor: Acad. Prof. Dr. MIRCEA DUMITRU

Co-tutelary: Dr. MARIAN CĂLBOREAN

Bucharest, 2025

I am grateful to professors Mircea Dumitru, Marian Călborean,
Graham Priest, Andrei Sipoș, Gabriel Istrate and Alexandru Dragomir.

This project began in 2024 as a common seminar project for Marian Călborean's
Logic and Software elective course (Faculty of Philosophy, University of Bucharest).
I am grateful for his sustained guidance, which led to this work being finished.

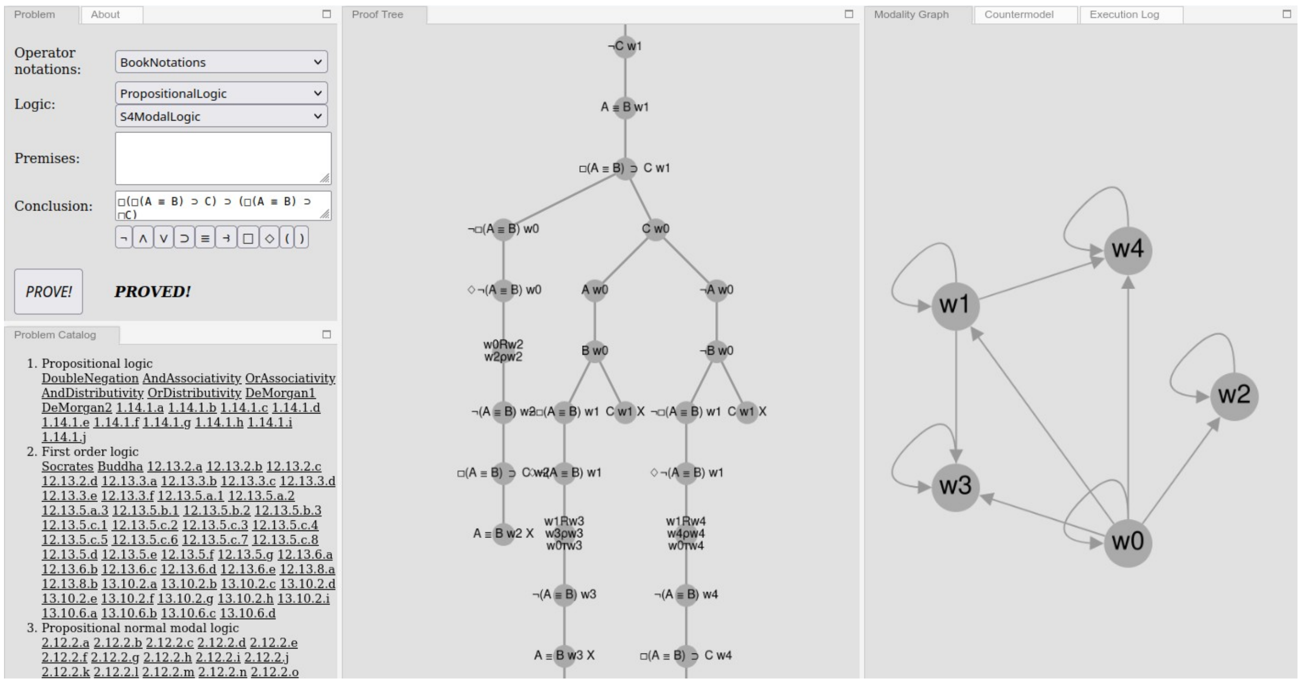
Partial results of the research and development process underlying this work were presented by myself and Marian Călborean at three academic events. These presentations should be regarded as works in progress that have contributed to the development of the current work.

Contents

Introduction

I have developed an automated proof calculator software for propositional and quantified, classical and non-classical logics. The software implements, adapts and extends the tableaux proof system presented by renowned philosopher and logician Graham Priest in his 2008 book *An Introduction to Non-Classical Logic. From If to Is (2nd edition)*[1]. Part I of this thesis describes how I have adapted and implemented book's tableaux proof algorithm as a software. Part II of this thesis presents two theoretical extensions to the book and their implementation. The final part contains two annexes, with more logical and software development details.

The software is free and open source, available online at andob.io/incl. Its source code is published at github.com/andob/INCL-automated-theorem-prover. *Figure 1* shows a screenshot of the software. It is written mostly in Rust, a modern programming language designed for building highly efficient, reliable and safe software[2; p.XXV]. In addition to that, functional programming features such as higher-order functions[2; p.257-270], algebraic data types[2; p.95-102] and pattern matching[2; p.102-107] make Rust an excellent choice for writing proof calculators (aka automated theorem provers).



Operator notations: BookNotations

Logic: PropositionalLogic, S4ModalLogic

Premises:

Conclusion: □(□(A ≡ B) ⊃ C) ⊃ (□(A ≡ B) ⊃ □C)

PROVED!

1. Propositional logic
   DoubleNegation AndAssociativity OrAssociativity AndDistributivity OrDistributivity DeMorgan1 DeMorgan2 1.14.1.a 1.14.1.b 1.14.1.c 1.14.1.d 1.14.1.e 1.14.1.f 1.14.1.g 1.14.1.h 1.14.1.i 1.14.1.j
2. First order logic
   Socrates Buddha 12.13.2.a 12.13.2.b 12.13.2.c 12.13.2.d 12.13.3.a 12.13.3.b 12.13.3.c 12.13.3.d 12.13.3.e 12.13.3.f 12.13.5.a.1 12.13.5.a.2 12.13.5.a.3 12.13.5.b.1 12.13.5.b.2 12.13.5.b.3 12.13.5.c.1 12.13.5.c.2 12.13.5.c.3 12.13.5.c.4 12.13.5.c.5 12.13.5.c.6 12.13.5.c.7 12.13.5.c.8 12.13.5.d 12.13.5.e 12.13.5.f 12.13.5.g 12.13.6.a 12.13.6.b 12.13.6.c 12.13.6.d 12.13.6.e 12.13.8.a 12.13.8.b 13.10.2.a 13.10.2.b 13.10.2.c 13.10.2.d 13.10.2.e 13.10.2.f 13.10.2.g 13.10.2.h 13.10.2.i 13.10.6.a 13.10.6.b 13.10.6.c 13.10.6.d
3. Propositional normal modal logic
   2.12.2.a 2.12.2.b 2.12.2.c 2.12.2.d 2.12.2.e 2.12.2.f 2.12.2.g 2.12.2.h 2.12.2.i 2.12.2.j 2.12.2.k 2.12.2.l 2.12.2.m 2.12.2.n 2.12.2.o

Figure 1: A screenshot of the running software project

The book offers tableaux proof systems for various logics. While developing the software, I have implemented most of them. *Figure 2* shows the implementation coverage.

	Book chapter	**Status**
	Part I: Propositional logics	
1	Classical logic	√ Classical logic fully implemented.
2	Basic modal logic	√ K modal logic fully implemented.
3	Normal modal logics	√ T, B, S4, S5 modal logics fully implemented. K tense modal logic partially implemented.
4	Non-normal modal logics	√ S0.5, N, S2, S3, S3.5 modal logics fully implemented.
5	Conditional logics	√ C fully implemented. C+ partially implemented.
6	Intuitionist logic	√ Fully implemented.
7	Many-valued logics	√ No tableaux method in this chapter.
8	First degree entailment	√ Regular FDE fully implemented. Routley star FDE variant not implemented.
9	Logics with gaps, gluts and worlds	√ K4, N4, I4, I3, W logics fully implemented.
10	Relevant logics	X Not implemented.
11	Fuzzy logics	√ No tableaux method in this chapter. However, the software implements Nicola Olivetti's[8] fuzzy logic tableaux system.
11a	Many-valued modal logics	√ Lukasiewicz logic, Kleene logic, Logic of Paradox, RMingle3 logic fully implemented.
	Part II: First-Order counterparts	
12	Classical first-order logic	√ Fully implemented.
13	Free logics	√ Implemented only with negativity constraint. Positive free logic is not implemented.
14	Constant domain modal logics	√ Fully implemented.
15	Variable domain modal logics	√ Implemented only with negativity constraint.
16	Necessary identity in modal logic	√ Fully implemented.
17	Contingent identity in modal logic	√ Fully implemented.
18	Non-normal modal logics	√ Fully implemented.
19	Conditional logics	√ C fully implemented. C+ partially implemented.
20	Intuitionist logic	√ First kind of tableaux implemented. Second kind of tableaux not implemented.
21	Many-valued logics	√ Fully implemented.
22	First degree entailment	√ Fully implemented.
23	Logics with gaps, gluts and worlds	√ Fully implemented.
24	Relevant logics	X Not implemented.
25	Fuzzy logics	√ Fully implemented.

Figure 2: Software implementation coverage

Philosophical notes

What is logic? Why are there so many philosophical logics?

In philosophy, formal logic is used to study argument validity. An argument consists of premises and a conclusion. An argument is valid iff (if and only if) the conclusion follows from the premises. An argument is valid and sound iff it is valid and all of its premises are true. Formal logic deals only with validity, as truth-seeking is the domain of science and philosophy.[3; p.3]

Logic is similar to ethics: it is not descriptive, but normative. That is, logic does not study how people usually think (that is the domain of psychology). Rather, it states how one should think correctly.[3; p.1] But what is *the correct way of thinking* is highly contentious. This is the reason there is no single logic, but a plethora of logics, developed by various philosophers, logicians and mathematicians.[1] A brief account of history is needed to clarify why.

In Ancient Greece, Aristotle developed the first formal logic, the syllogistic logic[4; p.23]. This was the *traditional logic* used by philosophers up until the 20th century. By then, philosophers and logicians Gottlob Frege and Bertrand Russell developed what is now called *classical logic*.[4; p.478] This is the contemporary formal logic, commonly used in mathematics and computer science. However, classical logic has its limitations. Later non-classical logics try to overcome these limitations.

In order to study argument validity, one has to *formalize* the argument (that is, to translate the argument from natural language into a logic's formal language). For instance, in classical first-order logic, the argument *All humans are mortal. Socrates is human. Hence, Socrates is mortal.* can be formalized as $\forall x(Hx \supset Mx), Hs \vdash Ms$ (where Hx = *x is human*, Mx = *x is mortal*, s = *Socrates*). But not all natural language statements can be formalized in classical first-order logic.

First of all, modal statements (statements involving *possibility* and *necessity*) cannot be straightforwardly formalized in classical logic. Modal non-classical logics were built by Clarence Irving Lewis, Saul Kripke and others to include modal operators *possible* and *necessary*.[4; p.548]

Secondly, classical logic is a bivalent logic. That is, any statement can be either *true* or *false*. But there are some paradoxical statements which cannot be formalized using classical logic. For instance, consider liar's paradox: *This very statement is false*. Let P: *P is false*. What is the truth value of P? For P cannot be neither *true* nor *false*, since P is self-contradictory. Maybe P is *unknown*? Many-valued non-classical logics were developed in order to overcome this limitation.

Particularly, three-valued non-classical logics add a third truth value (*unknown*) besides *true* and *false*.[1; p.120] Furthermore, four-valued non-classical logics add two truth values (*neither true nor false*, *both true and false*) besides *true* and *false*.[1; p.154] This is done by denying classical logic's law of excluded middle $(P \lor \neg P)$ and law of non-contradiction $\neg(P \land \neg P)$. Even more, there are infinite valued non-classical fuzzy logics, with truth values yielding in the [0...1] real number interval (where 0 means certainly false, 1 means certainly true).[1; p.221]

Neither classical logic nor finite many-valued logics can straightforwardly model the problem of vagueness. Consider the sorites paradox: *Imagine a heap of sand. How many grains of sand is a heap of sand? If we take out from the heap one grain at a time, when exactly does a heap become a non-heap?*. Obviously, an exact answer is impossible, since the word *heap* is a highly vague word. Maybe the truth value of *Is this a heap?* is neither true, nor false, but somewhere in between (eg. 0.25, 0.93 and so on). This can be formalized by infinite-valued non-classical fuzzy logics.[1; p.221]

What is common in all philosophical logics?

Every formal logic has the following features:

- Syntax: a set of operators and well-forming rules.
- Semantics: a set of truth values and interpretations on value and operator meaning.
- Proof systems: methods of proving formal argument validity. Examples include natural deduction, sequent calculus and tableaux method.[1; p.XVIII]
- Soundness: any statement that gets proved is true.[1; p.16]
- Completeness: all true statements are provable, there are no true unprovable statements.[3; p.110]
- Philosophical relevance (for instance, applications in philosophy of language).

What is a proof calculator (aka an automated theorem prover)?

A proof calculator (commonly known as an automated theorem prover) is a software which automatically proves mathematical statements. Particularly, a logic proof calculator proves the validity of formal arguments, by implementing a proof system. This software implements the tableaux proof system of Priest's *Introduction to Non-Classical Logic*[1].

Is there something which cannot be proved automatically?

No. As long as the logic and the proof system are complete, there are no unprovable truths. Particularly, the tableaux proof system of Priest[1] is complete. Assuming the software correctly implements it, the software is also complete. Still, it is worth noting that no software is decidable. That is, given any Turing machine (or software) T_1, no other Turing machine (or software) T_2 can determine whether T_1 will finish or hang prior to T_1's execution[3; p.100].

How fast is the software?

The tableaux method is a rather inefficient algorithm, whose best case complexity is quasi-polynomial[5]. Still, computers are pretty fast nowadays. So is the prover software.

What is the philosophical relevance of a proof calculator?

According to Priest, the tableaux method is excellent for didactic purposes: "One reason for this choice [the tableaux method] is that constructing tableau proofs, and so 'getting a feel' for what is, and what is not, valid in a logic, is very easy (indeed, it is algorithmic). Another is that the soundness and, particularly, completeness proofs for logics are very simple using tableaux. Since these areas are both ones where inexperienced students experience difficulty, tableaux have great pedagogical attractions."[1; p.XVIII].

Thus, the software can be used professionally, for either teaching or research. But besides this, the existence of such a proof calculator (and the existence of proof calculators in general) yields important questions in philosophy of mind. *Is human reasoning completely computational, algorithmic? Is the human mind just a software running on a hardware (the brain)? Or is it more than that?*[6] Although these (yet unanswered) philosophical dilemmas are very fascinating, this work will not try to answer them.

Logical notation

- In propositional logics, propositions are marked with uppercase letters. E.g.: P, Q, W
- In first-order logics, predicates are marked with uppercase letters and their arguments (objects, variables, constants) are marked with lowercase letters. Predicate arguments are enclosed by square brackets and delimited by comma. E.g.: P[x], P[y], Q[x, y], W[x, y, z]

- For unary logical operations, the following symbols are used:
  ¬ (non), ◇ (possible), □ (necessary)

- For binary logical operations, the following symbols are used:
  ∧ (and), ∨ (or), ⊃ (imply), ≡ (equivalent to), ⥽ (strict imply), ▷ (conditional)
  E.g.: $P \land (Q \lor R) \equiv (P \land Q) \lor (P \land R)$
  E.g.: $(\Box(A \supset B) \land \Diamond(A \land C)) \supset \Diamond(B \land C)$

- For quantifiers, the following symbols are used:
  ∃ (there exists), ∀ (for all), 𝔈 (there exists as object)
  E.g.: $\forall x \, \Box \, A[x] \supset \Box \, \forall x \, A[x]$

- For modality graph operations, the following symbols are used:
  ☆ (new node creation), ⇉ (adjacent nodes iteration)
  E.g.: w_i ☆ w_j marks the creation of a w_j node and a $w_i \rightarrow w_j$ vertex
  E.g.: $w_i \rightrightarrows w_j$ marks the iteration of all w_j nodes accessible from w_i by a $w_i \rightarrow w_j$ vertex.

- **Entailment is marked with the ⊢ symbol.**
- **Contradiction is marked with the ⚡ symbol.**

Part I: Developing the calculator software

I.1. Data structures

The software defines *a problem* and *a logic* as follows:

- **A problem** consists of **a logic, 0..n premises and a conclusion**
- **A logic** consists of:
  - An unique name (e.g.: PropositionalLogic, S5ModalLogic)
  - The number of available truth values (2, 3, 4 or ∞)
  - A syntax: a set of symbols available in that logic (e.g.: ¬, ∧ , ∨ , ⊃ ,…)
  - A set of rules on how to build the proof tree and how to detect contradictions.

A data structure is an organized structure of storing data in memory[7; p.64]. I use three data structures:

- **A tree**, which represents **the proof tree**
- **A directed graph**, which represents **the modality graph**
- **A queue**, which manages proof algorithm execution.

A tree is a data structure consisting of nodes disposed in an arborescent manner.[7; p.396] That is, each node must have 0 or 1 parent node(s) and 0 to n subnodes. In a tree data structure:

- The node which has no parent is called *root node*. A tree contains only one root node.
- The nodes which have no children are called *leaf nodes*.
- A list containing all adjacent nodes from the root node to a leaf node is called a *tree branch*.
- In *Figure 3*, A is the root node, E, F, C, D are leaf nodes. A→B→E, A→C are tree branches.

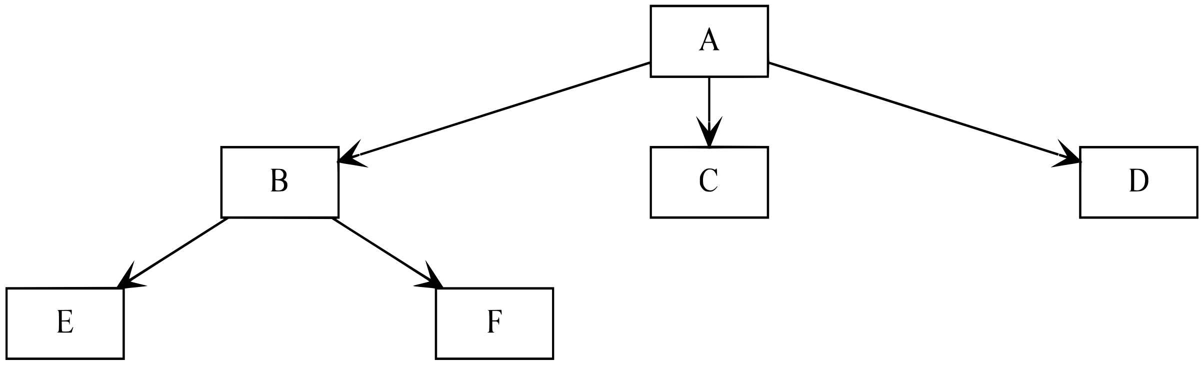


Figure 3: A visual representation of a tree data structure

A directed graph is a data structure consisting of a set of nodes and a set of vertices[7; p. 566]. A vertex is a directed relation between two nodes.

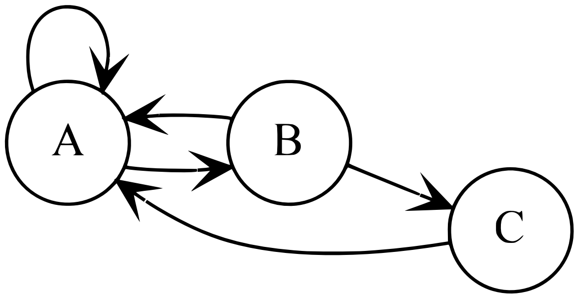


Figure 4: A visual representation of a directed graph

The computer science concept of a directed graph can be directly mapped to the philosophical concept of a Kripke frame. Graph nodes represent *possible worlds* and graph vertices represent *accessibility relations between possible worlds*. In this work, I will use the terms *modality graph node* and *possible world* interchangeably. I will also use the terms *modality graph vertex* and *accessibility relation between possible worlds* interchangeably.

A queue is a FIFO (*first in first out*) data structure[7; p.126]. Data is sequentially pushed into the rear of the queue and sequentially popped (taken out) from the front of the queue. In the software project, a queue is used to handle the processing of the formulas. The proof algorithm will sequentially pop and process formulas from the queue, until the queue becomes empty.

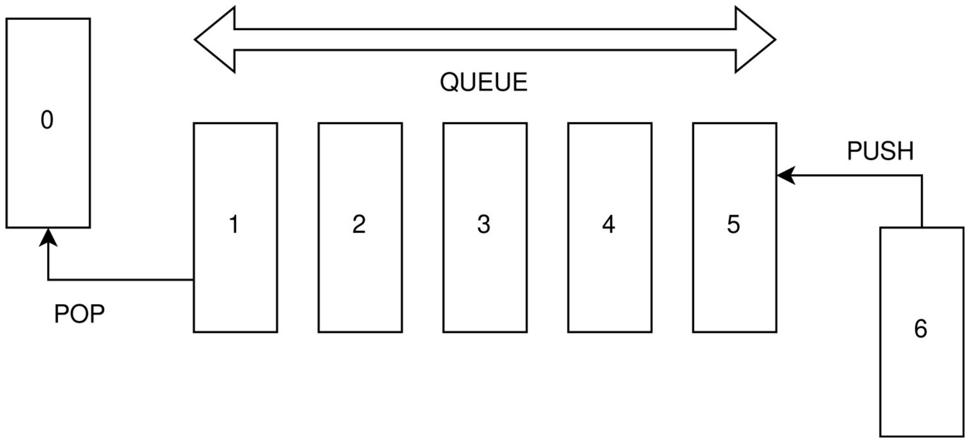


Figure 5: A visual representation of a queue

I.2. Implementing classical logic

Classical propositional logic stipulates two truth values (true and false) and a series of operations (non, and, or, imply, equivalent to). Their behavior is represented in the following truth table[1; p.5]:

P	Q	¬P	P ∧ Q	P ∨ Q	P ⊃ Q	P ≡ Q
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

Figure 6: Classical logic truth table

In classical propositional logic, the tableaux rules are[1; p.6-9]:

$$\begin{tikzcd} \neg\neg P \arrow[d] \\ P \end{tikzcd}$$

$$\begin{tikzcd} P \land Q \arrow[d] \\ P \arrow[d] \\ Q \end{tikzcd}$$

$$\begin{tikzcd} & P \lor Q \arrow[dl] \arrow[dr] & \\ P & & Q \end{tikzcd}$$

$$\begin{tikzcd} & \neg(P \land Q) \arrow[dl] \arrow[dr] & \\ \neg P & & \neg Q \end{tikzcd}$$

$$\begin{tikzcd} \neg(P \lor Q) \arrow[d] \\ \neg P \arrow[d] \\ \neg Q \end{tikzcd}$$

$$\begin{tikzcd} & P \supset Q \arrow[dl] \arrow[dr] & \\ \neg P & & Q \end{tikzcd}$$

$$\begin{tikzcd} & P \equiv Q \arrow[dl] \arrow[dr] & \\ P \arrow[d] & & \neg P \arrow[d] \\ Q & & \neg Q \end{tikzcd}$$

$$\begin{tikzcd} \neg(P \supset Q) \arrow[d] \\ P \arrow[d] \\ \neg Q \end{tikzcd}$$

$$\begin{tikzcd} & \neg(P \equiv Q) \arrow[dl] \arrow[dr] & \\ P \arrow[d] & & \neg P \arrow[d] \\ \neg Q & & Q \end{tikzcd}$$

Figure 7: Classical logic tableaux rules

The software takes **a problem** as an input and outputs **a proof tree** and **a status**:

- The problem is proved if the proof tree has contradictions on all branches. On a tree branch, contradiction is detected when the branch contains two nodes P and ¬P (where P is an atom).
- The problem is disproved if the proof tree does not have contradictions on all branches. If the problem gets disproved, a countermodel will be computed, by reading a non-contradictory branch and filtering all nodes with formulas of form P or ¬P (where P is an atom).
- The software will timeout (neither prove nor disprove the problem) if the proof tree reaches 250 nodes (if it takes *too long* to compute).

The software **implementation** of the tableaux algorithm can be described as follows:

1. Take a **problem** (a logic, 0..n premises, a conclusion) as an input
2. Build **a proof tree** with vertically disposed sequential nodes. For each premise, there will be a tree node with that premise. There will be another tree node with the non-conclusion.
3. Build **a queue** with all the premises and the non-conclusion.
4. While the queue is not empty:
   a. Pop a formula out of the queue
   b. Find the specific applicable logic rule for this formula
   c. Apply the rule and append the result to the proof tree
   d. Push the resulting formulas to the queue
   e. Check for contradictions in the proof tree
   f. Check for timeout (if reached, mark the problem with timeout)
   g. Repeat steps a-f until the queue becomes empty
5. If there is contradiction on all tree branches, mark the problem as proved
6. Or else, mark the problem as disproved
   a. Read a countermodel from a non-contradictory proof tree branch, by fetching all P nodes (where P is an atom) and all ¬P nodes (where P is an atom)[1; p.10]*
7. **Output** the resulting proof tree.

(*) For instance, consider a proof tree with only one branch. This branch contains the following nodes: { (P ∧ (Q ∧ ¬W)), (P ∧ Q), ¬W, P, Q }. This is a non-contradictory branch, a countermodel can be built from it. Given the branch contains { P, Q, ¬W }, the countermodel is { P:T, Q:T, W:F }.

For instance, this is a step-by-step tour of how the software proves ¬(P ≡ Q), P ⊢ ¬Q.

- The input is a problem with two premises (¬(P ≡ Q) and P) and conclusion ¬Q
- A proof tree is created with these nodes: ¬(P ≡ Q), P (premises), ¬¬Q (non-conclusion)
- A queue ⟨ ¬(P ≡ Q), P, ¬¬Q ⟩ is created.
- 1st iteration: since the queue is not empty:
  - → The ¬(P ≡ Q) formula is popped from the queue, the queue becomes ⟨ P, ¬¬Q ⟩
  - → The ¬≡ rule is applied. The tree is split into two branches. On the left side of the tree, two nodes are added: P and ¬Q. On the right side, other two nodes are added: ¬P and Q.
  - → All resulting formulas are pushed to the queue. It becomes ⟨ P, ¬¬Q, P, ¬Q, ¬P, Q ⟩
  - → Check for contradictions: there is a contradiction on the right branch.

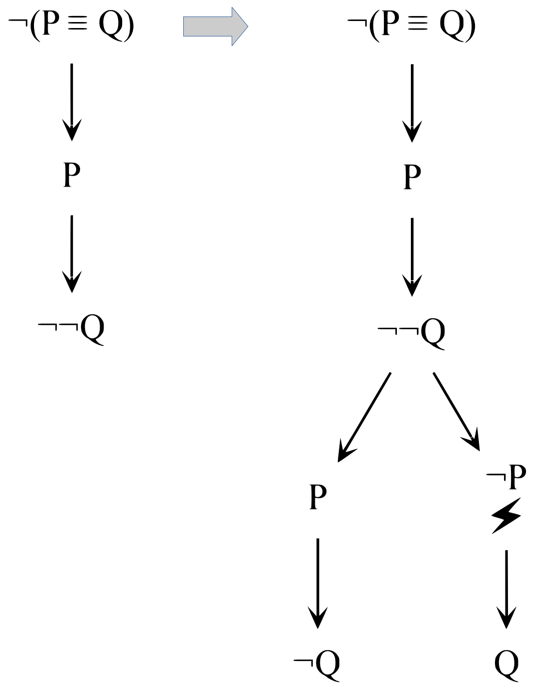


Figure 8: The proof tree before (left) and after (right) the 1st iteration

- 2nd iteration
  - → The P formula is popped from the queue. The queue becomes ⟨ ¬¬Q, P, ¬Q, ¬P, Q ⟩
  - → There is no rule to apply for P. Skip this iteration.
- 3rd iteration
  - → The ¬¬Q formula is popped from the queue. The queue becomes ⟨ P, ¬Q, ¬P, Q ⟩
  - → The double negation rule is applied. A Q node is added on each tree branch.

→ The resulting formula (Q) is pushed to the queue. It becomes ⟨ P, ¬Q, ¬P, Q, Q ⟩

→ Check for contradictions: there is a contradiction on the left branch.

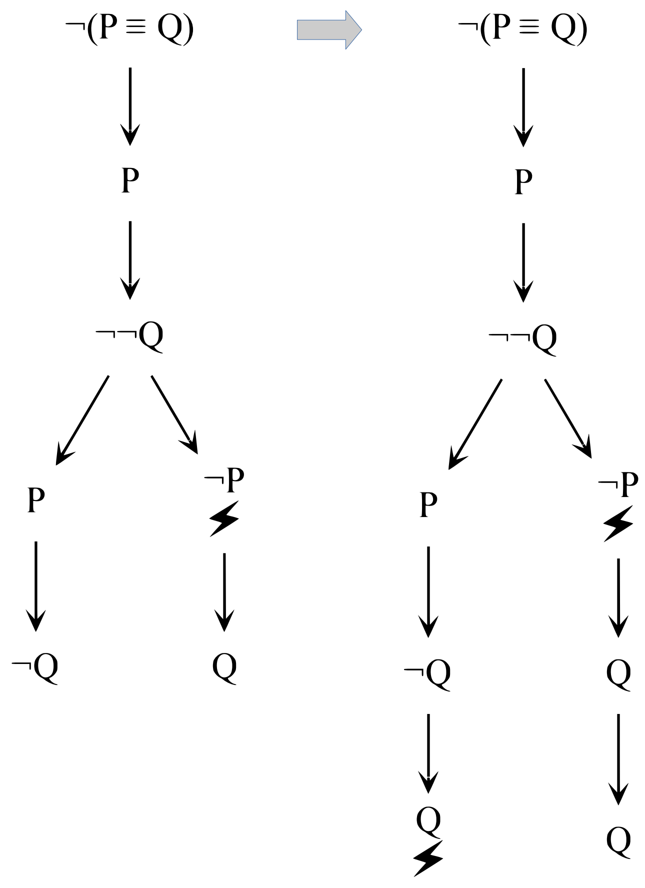


Figure 9: The proof tree before (left) and after (right) the 3rd iteration

- 4th iteration: P is popped from the queue. It becomes ⟨ ¬Q, ¬P, Q, Q ⟩. No rule to apply.
- 5th iteration: ¬Q is popped from the queue. It becomes ⟨ ¬P, Q, Q ⟩. No rule to apply.
- 6th iteration: ¬P is popped from the queue. It becomes ⟨ Q, Q ⟩. No rule to apply.
- 7th iteration: Q is popped from the queue. It becomes ⟨ Q ⟩. No rule to apply.
- 8th iteration: Q is popped from the queue. The queue becomes empty. No rule to apply.
- The queue is empty now. There is nothing left to process.
- There is contradiction on all tree branches. The problem is proved!*

(*) This is, of course, by *reductio ad absurdum*. Initially ¬(P ≡ Q), P ⊢¬¬Q was assumed. This led to contradiction on all tree branches. It can't be right, thus ¬(P ≡ Q), P ⊢¬Q is true.

I.3. Implementing propositional modal logics

Modal logics add two new operators: $\Diamond$(possible) and $\Box$ (necessary). $\Diamond P$ is true iff P is true in any accessible possible world, while $\Box P$ is true iff P is true in all accessible possible worlds. A possible world can be imagined as a state of affairs, in which some statements are true, others are false.[3; p.40]

Proof tree node's inner data will now contain the formula and the possible world. Tableaux rules[1; p.24] are depicted in *Figure 10*. $w_i ☆ w_j$ marks the creation of a new possible world w_j from w_i, while $w_i \rightrightarrows w_j$ marks the iteration of all possible worlds w_j from w_i.

$$\begin{tikzcd}
\Diamond P, w_i \arrow[d] & \Box P, w_i \arrow[d] & \neg\Diamond P, w_i \arrow[d] & \neg\Box P, w_i \arrow[d] & P, w_i \arrow[d, dash] \\
w_i ☆ w_j \arrow[d] & w_i \rightrightarrows w_j \arrow[d] & \Box\neg P, w_i & \Diamond\neg P, w_i & \neg P, w_i \; ⚡ \\
P, w_j & P, w_j & & &
\end{tikzcd}$$

Figure 10: modal logic rules

Technically, the $\Box P$, w_i rule will iterate all modality graph vertices $w_i \rightarrow w_j$. For each accessible possible world w_j, the rule will add a new P, w_j proof tree node. On the other hand, the $\Diamond P$ rule:

- Will create a new modality graph node w_j and a new modality graph vertex $w_i \rightarrow w_j$
- If the logic is reflexive*, the rule will add missing modality graph vertices. That is, for each graph node w_i, a new graph vertex $w_i \rightarrow w_i$ will be added, if missing.
- If the logic is symmetric, the rule will add missing symmetric graph vertices. That is, for each graph vertex $w_i \rightarrow w_j$, a new graph vertex $w_j \rightarrow w_i$ will be added, if missing.

(*) Different modal logics have different graph requirements. For instance, S4 modal logic requires a reflexive, transitive graph. S5 modal logic requires a reflexive, symmetric and transitive graph. K modal logic does not have any such requirement.

- If the logic is transitive, the rule will add missing transitive graph vertices. That is, for each graph vertex $w_i \rightarrow w_j$ and for each graph vertex $w_j \rightarrow w_k$, a new graph vertex $w_i \rightarrow w_k$ will be added, if missing.
- The rule will add a new P, w_j proof tree node.

For instance, the software takes the following steps while it disproves $\neg(\Diamond P \land \Box Q)$ in S4:

- Initialization: a proof tree with root node: $\neg\neg(\Diamond P \land \Box Q), w_0$ (non-conclusion in w_0) is created
- 1st iteration: $\neg\neg(\Diamond P \land \Box Q), w_0$ is expanded to $\Diamond P \land \Box Q, w_0$
- 2nd iteration: $\Diamond P \land \Box Q, w_0$ is expanded to $\Diamond P, w_0$ and $\Box Q, w_0$
- 3rd iteration: $\Diamond P, w_0$ is applied:
  - Create a new modality graph node w_1 and a modality graph vertex $w_0 \rightarrow w_1$
  - The logic requires a reflexive graph: add $w_0 \rightarrow w_0$, $w_1 \rightarrow w_1$ modality graph vertices
  - The logic requires a transitive graph: no missing vertices to add
  - Add a new proof tree node P, w_1

$$\begin{tikzcd}\neg\neg(\Diamond P \land \Box Q), w0 \arrow[r] & \Diamond P \land \Box Q, w0 \arrow[r] & \Diamond P, w0 \arrow[r] & \Box Q, w0\end{tikzcd}$$

$$\begin{tikzcd}\neg\neg(\Diamond P \land \Box Q), w0 \arrow[r] & \Diamond P \land \Box Q, w0 \arrow[r] & \Diamond P, w0 \arrow[r] & \Box Q, w0 \arrow[r] & w0 \star w1 \arrow[r] & P, w1\end{tikzcd}$$

Figure 11: the proof tree before (top) and after (bottom) the 3rd iteration (horizontally rendered)

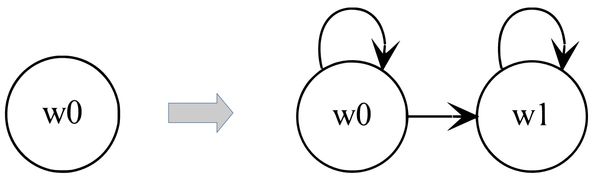


Figure 12: the modality graph before (left) and after (right) the 3rd iteration

- 4th iteration: $\Box Q, w_0$ is applied:
  - Iterate all modality graph vertices that start with w_0: $w_0 \rightarrow w_0$ and $w_0 \rightarrow w_1$. Take all accessible possible worlds: w_0 and w_1. Add two new proof tree nodes: Q, w_0 and Q, w_1.

$$\begin{tikzcd}\neg\neg(\Diamond P \land \Box Q), w0 \arrow[r] & \Diamond P \land \Box Q, w0 \arrow[r] & \Diamond P, w0 \arrow[r] & \Box Q, w0 \arrow[r] & w0 \star w1 \arrow[r] & P, w1\end{tikzcd}$$

$$\begin{tikzcd}{} \arrow[r] & w0 \rightrightarrows w1 \arrow[r] & Q, w0 \arrow[r] & Q, w1\end{tikzcd}$$

Figure 13: the proof tree after the 4th iteration (horizontally rendered and split)

I.4. Implementing propositional many-valued modal logics

Many-valued logics stipulate the existence of other truth values, besides true and false. These other values can be interpreted as *unknown*, *neither true nor false* or *both true and false*, depending on logic. For instance, Kleene logic has 3 truth values (*true*, *false* and *unknown*). Its operations act according to the following truth table[1; p.122]:

P	¬P			P					P		
T	F		P ∧ Q	F	U	T		P ∨ Q	F	U	T
U	U		F	F	F	F		F	F	U	T
F	T	Q	U	F	U	U	Q	U	U	U	T
			T	F	U	T		T	T	T	T

Figure 14: Kleene logic truth table

Proof tree node's inner data will now contain the formula, the possible world and a sign, which can be either + or −. The book's tableaux algorithm provides first degree entailment (FDE) rules, as a set of base rules for all three-valued and four-valued logics. These rules are[1; p.144]:

$$\begin{tikzcd}
\neg(P \land Q), w_i, \pm \arrow[d] & \neg(P \lor Q), w_i, \pm \arrow[d] & \neg\neg P, w_i, \pm \arrow[d] & P, w_i, + \arrow[d, dash] \\
\neg P \lor \neg Q, w_i, \pm & \neg P \land \neg Q, w_i, \pm & P, w_i, \pm & P, w_i, - \\
 & & & \lightning
\end{tikzcd}$$

$$\begin{tikzcd}
P \land Q, w_i, + \arrow[d] & & P \lor Q, w_i, + \arrow[dl] \arrow[dr] & & & P \land Q, w_i, - \arrow[dl] \arrow[dr] & & P \lor Q, w_i, - \arrow[d] \\
P, w_i, + \arrow[d] & P, w_i, + & & Q, w_i, + & P, w_i, - & & Q, w_i, - & P, w_i, - \arrow[d] \\
Q, w_i, + & & & & & & & Q, w_i, -
\end{tikzcd}$$

Figure 15: First Degree Entailment (FDE) rules

The initialization step of the algorithm is modified. The initial proof tree is generated as follows: given a problem with premises $P_1...P_n$ and conclusion C, for each premise P_i, a tree node with P_i, + will be created; then, a tree node with C, − will be created.

Also note that many-valued logics have the following contradiction criteria:

- P, w_i, + is contradictory with P, w_i, − on all many-valued logics[1; p.145]. In addition:
- P, w_i, + is contradictory with ¬P, w_i, + on some many-valued logics (K3, Ł3, I3)[1; p.148-150, p.177]
- P, w_i, − is contradictory with ¬P, w_i, − on some many-valued logics (RM3, LP)[1; p.149]

For instance, the software takes the following steps while it proves P ∧ Q ⊢ P in FDE:

- Initialization: a proof tree with root node P ∧ Q, w_0, + and subnode P, w_0, − will be created.
- 1st iteration: P ∧ Q, w_0, + is expanded into P, w_0, +, and Q, w_0, +. Contradiction is detected.

$$\begin{tikzcd}
P \land Q, w0, + \arrow[d] \arrow[r, Rightarrow] & P \land Q, w0, + \arrow[d] \\
P, w0, - & P, w0, - \arrow[d] \\
& P, w0, + \arrow[d, "\lightning"] \\
& Q, w0, +
\end{tikzcd}$$

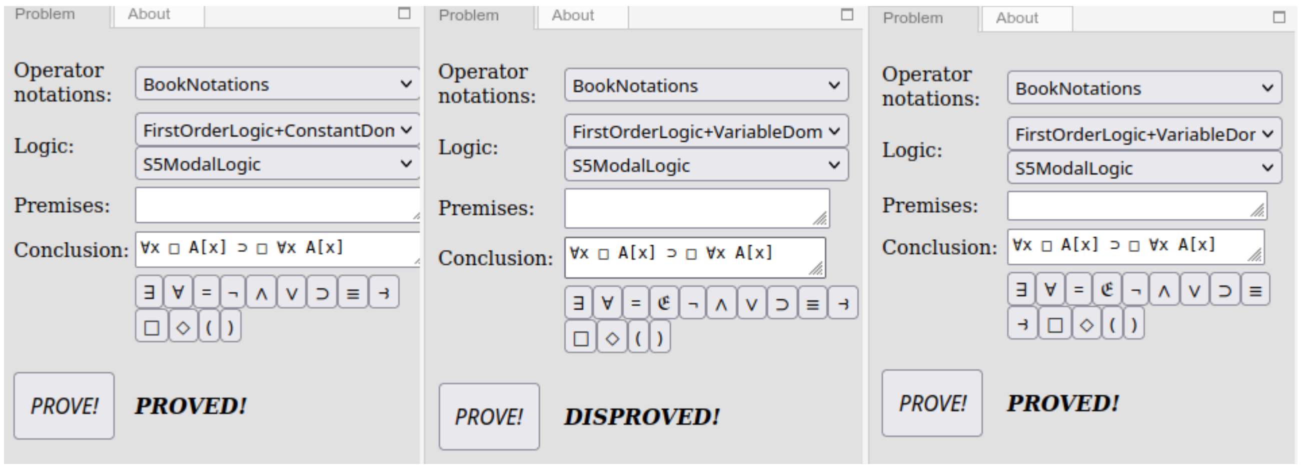
Figure 16: the proof tree before (left) and after (right) the 1st iteration

The software implements various three-valued and four-valued logics. For each logic, tableaux rules are defined by combining logic's specific rules with FDE tableaux rules and modal tableaux rules. For a complete catalog of rules, please check *Annex II*.

I.5. Implementing first-order modal logics

First-order logics (FOLs) extends propositional logics with predicates, predicate arguments, quantifiers and equality. The software implements six variants of FOLs:

- FOLs with constant domain, necessary identity
- FOLs with constant domain, contingent identity
- FOLs with variable domain, necessary identity, without domain increasing constraint
- FOLs with variable domain, contingent identity, without domain increasing constraint
- FOLs with variable domain, necessary identity, with domain increasing constraint
- FOLs with variable domain, contingent identity, with domain increasing constraint



Operator notations: BookNotations
Logic: FirstOrderLogic+ConstantDon
S5ModalLogic
Premises:
Conclusion: ∀x □ A[x] ⊃ □ ∀x A[x]
PROVED!

Operator notations: BookNotations
Logic: FirstOrderLogic+VariableDom
S5ModalLogic
Premises:
Conclusion: ∀x □ A[x] ⊃ □ ∀x A[x]
DISPROVED!

Operator notations: BookNotations
Logic: FirstOrderLogic+VariableDor
S5ModalLogic
Premises:
Conclusion: ∀x □ A[x] ⊃ □ ∀x A[x]
PROVED!

Figure 17: the software proves Barcan formula in constant domain S5 modal FOL (left), disproves it in variable domain S5 modal FOL, without domain increasing constraint (middle), and proves it in variable domain S5 modal FOL, with domain increasing constraint (right)

For technical reasons (ease of parsing), the software uses non-standard notations for predicates and their arguments. Arguments are enclosed by squared brackets and delimited by commas. E.g.: P[x] instead of Px, P[x,y] instead of Pxy and so on.

The software organizes predicate arguments into the following categories:

- Binding variables (e.g.: in ∃x P[x], x is a binding variable)
- Instantiated objects (e.g.: in P[b:x], b:x is an instantiated object with name:b, type:x)

The existence (∃) rule transforms binding variables into instantiated objects:

- Both binding variables and instantiated objects are uniquely identified by their names (while the object is created, unique names are generated considering existing names on the branch)
- Instantiated objects will carry not only their name, but also their type. The type of the instantiated object is the name of the original binding variable.
- Instantiated objects are displayed in the name:type format. For instance, in *Figure 18*, a:x is an instantiated object named a of type x spawned from the binding variable x.

$$\begin{tikzcd}
\exists x(\exists y(\exists z(P[x,y,z]))), w_i, + \arrow[d] & \text{Binding variables} \arrow[r] & \text{Instantiated objects} \\
\exists y(\exists z(P[a:x,y,z])), w_i, + \arrow[d] & x \arrow[r] & a:x \\
\exists z(P[a:x,b:y,z]), w_i, + \arrow[d] & y \arrow[r] & b:y \\
P[a:x,b:y,c:z], w_i, + & z \arrow[r] & c:z
\end{tikzcd}$$

Figure 18: a proof tree (left) and its corresponding instantiation process (right)

The existence rule varies[1; p.331] depending on the domain type. There is also a negated existence rule.

$$\begin{tikzcd}
\text{(constant domain)} & \text{(variable domain)} & \\
\exists x\ P[x], w_i, + \arrow[d] & \exists x\ P[x], w_i, + \arrow[d] & \neg\exists x\ P[x], w_i, \pm \arrow[d] \\
P[a:x], w_i, + & P[a:x], w_i, + \arrow[d] & \forall x\ \neg P[x], w_i, \pm \\
 & \mathfrak{E}a:x, w_i, + &
\end{tikzcd}$$

Figure 19: Existence rule variants and the negated existence rule

Before presenting the universal quantifier rule, an observation about equality and contradiction is needed. The algorithm supports formulas of equality between objects. Contradiction is detected when there are two nodes $P[a_1{:}x_1, a_2{:}x_2, \ldots, a_n{:}x_n]$ and $\neg P[b_1{:}y_1, b_2{:}y_2, \ldots, b_n{:}x_n]$ on the branch where $a_i{:}x_i = b_i{:}y_i$, for i from 1 to n. The algorithm detects that $x = y$ iff there is a node $x = y$ on the branch or there is a transitive chain of nodes $x = z_1, z_1 = z_2, \ldots, z_{n-1} = z_n, z_n = y$ on the branch:

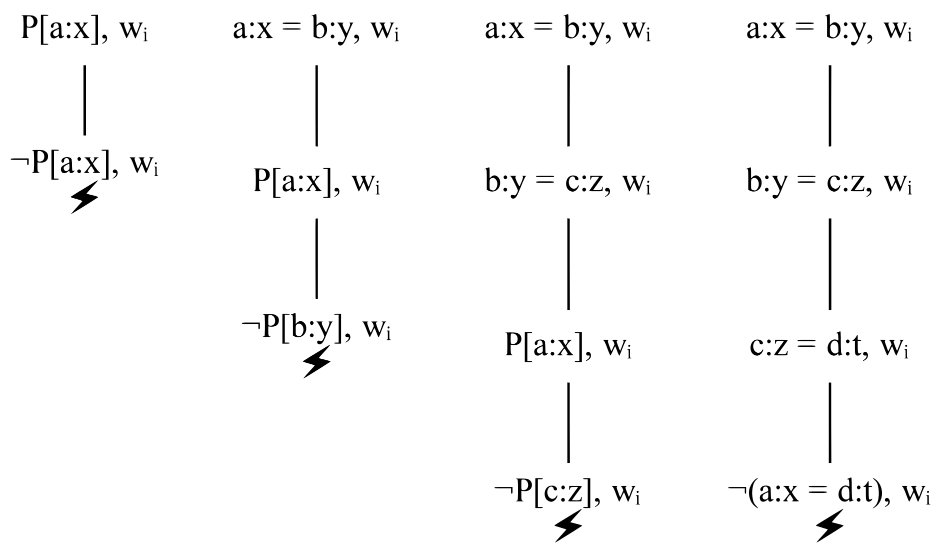


Figure 20: Examples of FOL contradiction detection

The universal quantifier ($\forall x\, P[x]$) rule will act according to the following algorithm:

- ➢ Read all instantiated objects of type x from the tree branch: $a_1{:}x, a_2{:}x, \ldots, a_n{:}x$
- ➢ For each instantiated object $a_i{:}x$
  - → If the domain type is constant domain, add a new tree node $P[a_i{:}x]$
  - → If the domain type is variable domain, add a new tree node $P[a_i{:}x]$ iff there is another node $\mathfrak{E}a_i{:}x$ already present on the tree branch
- ➢ Read all equivalent types of type x from the tree branch. For instance, if the branch contains $y_1{=}x, y_2{=}x, y_2{=}y_3, y_3{=}y_4$ nodes, equivalent types are y_1, y_2, y_3, y_4
- ➢ For each equivalent type y_i
  - → Read all instantiated objects of type y_i from the branch: $b_1{:}y_i, b_2{:}y_i, \ldots, b_n{:}y_i$
  - → For each instantiated object $b_j{:}y_i$
    - ◆ If the domain type is constant domain, add a new tree node $P[b_j{:}y_i]$
    - ◆ If the domain type is variable domain, add a new tree node $P[b_j{:}y_i]$ iff there is another node $\mathfrak{E}b_j{:}y_i$ already present on the tree branch.

The universal quantifier rule varies[1; p.331] depending on domain type. There is also a rule for non-∀:

$$\begin{tikzcd}
\begin{array}{c}\text{(constant domain)}\\ \forall x\ P[x],\ w_i,\ +\end{array} \arrow[d] & & \begin{array}{c}\text{(variable domain)}\\ \forall x\ P[x],\ w_i,\ +\end{array} \arrow[dl] \arrow[dr] & & \neg\forall x\ P[x],\ w_i,\ \pm \arrow[d] \\
P[a{:}x],\ w_i,\ + & \neg\mathfrak{E}a{:}x,\ w_i,\ + & & P[a{:}x],\ w_i,\ + & \exists x\ \neg P[x],\ w_i,\ \pm
\end{tikzcd}$$

Figure 21: Universal quantifier rule variants and the non-∀ rule

Furthermore, on modal FOL with necessary identity, object equalities are inherited to all accessible possible worlds. The algorithm will use the following extra rules[1; p.350]:

$$\begin{tikzcd}
a{:}x = b{:}y,\ w_i,\ \pm \arrow[d] & \neg(a{:}x = b{:}y),\ w_i,\ \pm \arrow[d] \\
w_i \rightrightarrows w_j \arrow[d] & w_i \rightrightarrows w_j \arrow[d] \\
a{:}x = b{:}y,\ w_j,\ \pm & \neg(a{:}x = b{:}y),\ w_j,\ \pm
\end{tikzcd}$$

Figure 22: Extra rules on modal FOL with necessary identity

On variable domain modal FOL with domain increasing constraint, $\mathfrak{E}a$ formulas are inherited to all accessible possible worlds. The algorithm will use the following extra rules[1; p.337]:

$$\begin{tikzcd}
\mathfrak{E}a{:}x,\ w_i,\ \pm \arrow[d] & \neg\mathfrak{E}a{:}x,\ w_i,\ \pm \arrow[d] \\
w_i \rightrightarrows w_j \arrow[d] & w_i \rightrightarrows w_j \arrow[d] \\
\mathfrak{E}a{:}x,\ w_j,\ \pm & \neg\mathfrak{E}a{:}x,\ w_j,\ \pm
\end{tikzcd}$$

Figure 23: Extra rules on variable domain FOL with domain increasing constraint

Part II: Theoretical extensions

II.1. Extending with Fuzzy logic

Fuzzy logics stipulate the existence of an infinite number of truth values, besides true (1) and false (0), as real numbers between 0 and 1. In Łukasziewicz's fuzzy logic (Ł$_\aleph$):

- As a notation, let $\mu(P)$ be the value of formula P; $\mu(P) \in [0\ldots1]$
- $\mu(\neg P) \overset{\text{def}}{=} 1 - \mu(P)$
- $\mu(P \wedge Q) \overset{\text{def}}{=} \min(\mu(P), \mu(Q))$
- $\mu(P \vee Q) \overset{\text{def}}{=} \max(\mu(P), \mu(Q))$
- $\mu(P \supset Q) \overset{\text{def}}{=} 1 - \mu(P) + \mu(Q)$ if $\mu(P) > \mu(Q)$ or 1 if $\mu(P) \leq \mu(Q)$[1; p. 224]

Priest's *Introduction to Non-Classical Logic* does not provide a tableaux method for Łukasziewicz's fuzzy logic. However, several tableaux proposals can be found in the literature. Nicola Olivetti proposes a tableaux method[8] with the following rules:

Rule 1:
$$\begin{tikzcd} \alpha, P \wedge Q, + \arrow[d] \\ \alpha, P, + \arrow[d] \\ \alpha, Q, + \end{tikzcd}$$

Rule 2:
$$\begin{tikzcd} & \alpha, P \wedge Q, - \arrow[dl] \arrow[dr] & \\ \alpha, P, - & & \alpha, Q, - \end{tikzcd}$$

Rule 3:
$$\begin{tikzcd} & \alpha, P \vee Q, + \arrow[dl] \arrow[dr] & \\ \alpha, P, + & & \alpha, Q, + \end{tikzcd}$$

Rule 4:
$$\begin{tikzcd} \alpha, P \vee Q, - \arrow[d] \\ \alpha, P, - \arrow[d] \\ \alpha, Q, - \end{tikzcd}$$

Rule 5:
$$\begin{tikzcd} & \alpha, P \supset Q, + \arrow[dl] \arrow[dr] & \\ \beta, P, - & & \alpha + \beta, Q, + \end{tikzcd}$$

Rule 6:
$$\begin{tikzcd} \alpha, P \supset Q, - \arrow[d] \\ \beta, P, + \arrow[d] \\ \alpha + \beta, Q, - \end{tikzcd}$$

Rule 7: (on atoms)
$$\begin{tikzcd} \alpha, P, + \arrow[d] \\ \alpha \geq \mu P \end{tikzcd}$$

Rule 8: (on atoms)
$$\begin{tikzcd} \alpha, P, - \arrow[d] \\ \alpha < \mu P \end{tikzcd}$$

Rule 9: (falsum)
$$\begin{tikzcd} \alpha, \perp, \pm \arrow[d] \\ \alpha \leq \mu \perp \end{tikzcd}$$

Figure 24: Łukasziewicz fuzzy logic (Ł$_\aleph$) tableaux rules

Proof tree node’s inner data will now contain (in this order) an algebraic expression, a formula and a sign, which can be either + or −. The algebraic expression can contain literals and variables (marked as greek letters), whose values are ranged in the [0...1] interval.

The initialization step of the algorithm is modified. The initial proof tree is generated as follows: given a problem with premises P1...Pn and conclusion C, for each premise Pi, a tree node with 0, Pi, + will be created; then a tree node with 0, C, − will be created.

Rules 1, 2, 3, 4 will propagate the previous algebraic expression. Rule 5 and rule 6 will create a new variable with unique name and new algebraic expressions. E.g.: $0 + \alpha + \beta$, $P \supset Q$, + will be transformed into γ, P, − on the left side, $0 + \alpha + \beta + \gamma$, Q, + on the right side.

Rule 7 and rule 8 apply only to atoms and will generate tree nodes with inequalities. Contradiction on a tree branch is detected by solving a system of all inequalities from that branch. If the system has no solution, the tree branch is marked as contradictory. Technically, the software uses a linear programming library[9] in order to solve systems of inequalities. Linear programming is a technique of maximizing a function given a set of constraints[10; p.25]. A system of inequalities maps to a linear program maximizing f(x) = 0 and constrained by the inequalities of the system.

Rule 9 defines the rule for falsum ($\perp$), an atom which is always false. Olivetti’s tableaux does not provide rules for non, as $\neg P \equiv P \supset \perp$. However, the software does not implement rule 9. Instead, it implements its own set of rules for non. I propose the following tableaux rules for non:

$$\begin{tikzcd}
\text{Rule 10:} & \text{Rule 11:} & \text{Rule 12:} & \text{Rule 13:} \\
\alpha, \neg\neg P, \pm \arrow[d] & \alpha, \neg(P \land Q), \pm \arrow[d] & \alpha, \neg(P \lor Q), \pm \arrow[d] & \alpha, \neg(P \supset Q), \pm \arrow[d] \\
\alpha, P, \pm & \alpha, \neg P \lor \neg Q, \pm & \alpha, \neg P \land \neg Q, \pm & \alpha, P \supset Q, \mp
\end{tikzcd}$$

$$\begin{tikzcd}
\text{Rule 14: (where P is an atom)} & \text{Rule 15: (where P is an atom)} \\
\alpha, \neg P, + \arrow[d] & \alpha, \neg P, - \arrow[d] \\
\alpha \geq 1 - \mu P & \alpha < 1 - \mu P
\end{tikzcd}$$

Figure 25: My rules for non on $Ł_{\aleph}$ tableaux

Here are sketches of proofs of rules 10...15:

- Rule 10: α, $\neg\neg P$, $\pm$ expands to α, P, $\pm$
  - → The value of $\neg\neg P$ is $\mu(\neg\neg P) = 1 - \mu(\neg P) = 1 - (1 - \mu(P)) = \mu(P)$.
  - → The value of a "α, $\neg\neg P$" node is $\mu(\neg\neg P) = \alpha$. But $\mu(\neg\neg P) = \mu(P)$. Hence, a "α, $\neg\neg P$" node should expand to a "α, P" node.
- Rule 11: α, $\neg(P \wedge Q)$, $\pm$ expands to α, $\neg P \vee \neg Q$, $\pm$
  - → The value of $\neg(P \wedge Q)$ is $\mu(\neg(P \wedge Q)) = 1 - \mu(P \wedge Q) = 1 - \min(\mu(P), \mu(Q)) = \max(1 - \mu(P), 1 - \mu(Q)) = \max(\mu(\neg P), \mu(\neg Q)) = \mu(\neg P \vee \neg Q)$
  - → Hence, a "α, $\neg(P \wedge Q)$" node should expand to a "α, $\neg P \vee \neg Q$" node.
- Rule 12: α, $\neg(P \vee Q)$, $\pm$ expands to α, $\neg P \wedge \neg Q$, $\pm$
  - → The value of $\neg(P \vee Q)$ is $\mu(\neg(P \vee Q)) = 1 - \mu(P \vee Q) = 1 - \max(\mu(P), \mu(Q)) = \min(1 - \mu(P), 1 - \mu(Q)) = \min(\mu(\neg P), \mu(\neg Q)) = \mu(\neg P \wedge \neg Q)$
  - → Hence, a "α, $\neg(P \vee Q)$" node should expand to a "α, $\neg P \wedge \neg Q$" node.
- Rules 10, 11, 12 are sign invariant:
  - → Consider a "α, $\neg\neg P$, $-$" node. This is equivalent to a "$1 - \alpha$, $\neg\neg P$, $+$" node, which would expand to a "$1 - \alpha$, P, $+$" node. But this is equivalent to a "α, P, $-$" node. Hence, a "α, $\neg\neg P$, $+$" node should expand to a "α, P, $+$" node. Also, a "α, $\neg\neg P$, $-$" node should expand to a "α, P, $-$" node. Thus, the rule is sign invariant.
  - → Same goes for rules 11 and 12. Proofs are trivial.
- Rule 13: α, $\neg(P \supset Q)$, $\pm$ expands to α, $P \supset Q$, $\mp$
  - → A "α, $\neg(P \supset Q)$, $+$" node is equivalent with a "$1 - \alpha$, $P \supset Q$, $+$" node. But this is equivalent with a "$1 - (1 - \alpha)$, $P \supset Q$, $-$" node.
  - → Thus, a "α, $\neg(P \supset Q)$, $+$" should expand to a "α, $P \supset Q$, $-$" node.
  - → A "α, $\neg(P \supset Q)$, $-$" node is equivalent with a "$1 - \alpha$, $P \supset Q$, $-$" node. But this is equivalent with a "$1 - (1 - \alpha)$, $P \supset Q$, $+$" node.
  - → Thus, a "α, $\neg(P \supset Q)$" node should expand to a "α, $P \supset Q$, $+$" node.
- Rule 14: α, $\neg P$, $+$ expands to $\alpha \geq 1 - \mu(P)$ iff P is an atom
  - → Consider noting $\neg P$ as Q. Then a "α, $\neg P$, $+$" node is equivalent to a "α, Q, $+$" node, which expands into a "$\alpha \geq \mu(Q)$" node. But $\mu(Q) = \mu(\neg P) = 1 - \mu(P)$
  - → Hence, a "α, $\neg P$, $+$" node should expand to a "$\alpha \geq 1 - \mu(P)$" node.

- Rule 15: $\alpha, \neg P, -$ expands to $\alpha < 1 - \mu(P)$ iff P is an atom
  - → Consider noting $\neg P$ as Q. Then a "$\alpha, \neg P, -$" node is equivalent to a "$\alpha, Q, +$" node, which expands into a "$\alpha < \mu(Q)$" node. But $\mu(Q) = \mu(\neg P) = 1 - \mu(P)$
  - → Hence, a "$\alpha, \neg P, -$" node should expand to a "$\alpha < 1 - \mu(P)$" node.

Figure 26 shows a tableau generated by the software as it proves $(P \supset \neg Q) \supset (Q \supset \neg P)$ in Ł$_{\aleph}$:

$$\begin{tikzcd}
 & 0, (P \supset \neg Q) \supset (Q \supset \neg P), - \arrow[d] & \\
 & \alpha, P \supset \neg Q, + \arrow[d] & \\
 & 0 + \alpha, Q \supset \neg P, - \arrow[d] & \\
 & \beta, Q, + \arrow[d] & \\
 & 0 + \alpha + \beta, \neg P, - \arrow[dl] \arrow[dr] & \\
\gamma, P, - \arrow[d] & & \alpha + \gamma, \neg Q, + \arrow[d] \\
0 + \alpha + \beta < 1 - \mu(P) \arrow[d] & & 0 + \alpha + \beta < 1 - \mu(P) \arrow[d] \\
\beta \geq \mu(Q) \arrow[d] & & \beta \geq \mu(Q) \arrow[d] \\
\gamma < \mu(P) & & \alpha + \gamma \geq 1 - \mu(Q) \\
\lightning & & \lightning
\end{tikzcd}$$

Figure 26: Proof that $(P \supset \neg Q) \supset (Q \supset \neg P)$ in Ł$_{\aleph}$

II.2. Extending with an alternative counter-model finder algorithm

When a problem gets disproved, a countermodel is built from a non-contradictory proof tree branch, by reading all its nodes of form P, w_i or ¬P, w_i (where P is an atom). *Figure 27* shows the relation between a proof tree, its modality graph and its countermodel.

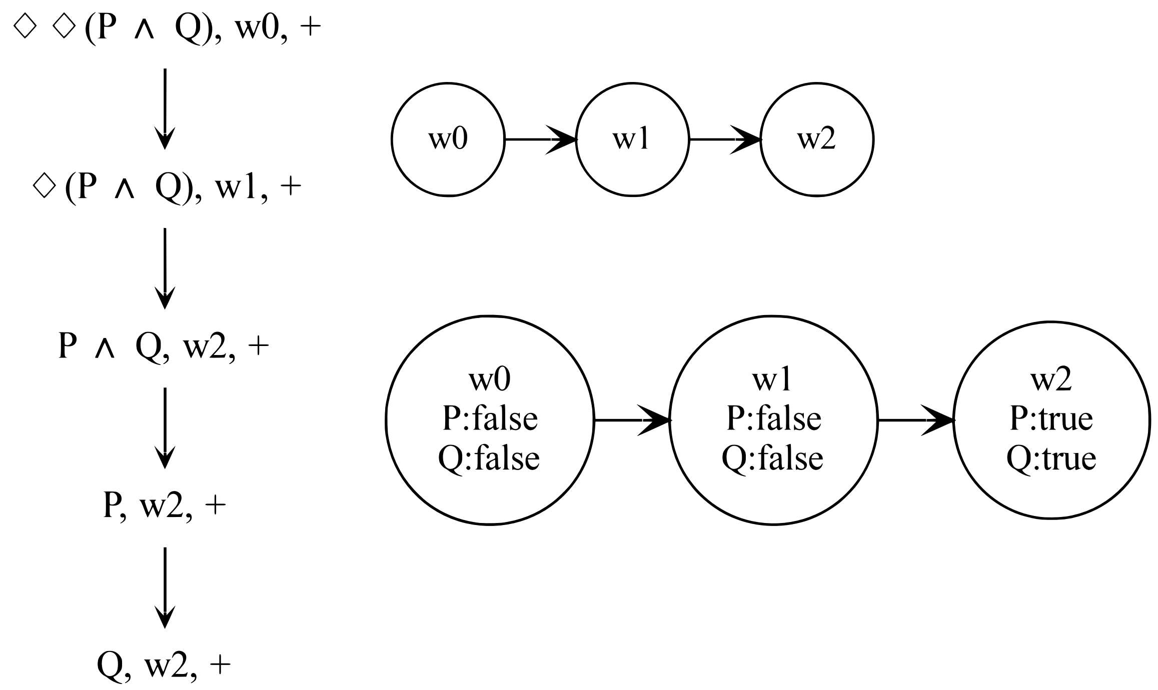


Figure 27: A disproved problem: its proof tree (left), its modality graph (top right) and its countermodel (bottom right)

After solving a problem, the software will mark one of the following statuses:

- Proved: the problem is proved, no countermodel is provided.
- Disproved: the problem is disproved, a countermodel is provided by reading all atomic values from a non-contradictory proof tree branch.
- Timeout: the problem is neither proved nor disproved since the tableau is an infinite tableau (a proof tree with infinite number of tree nodes). Since computer resources are limited, infinite tableaux can only be partially computed. The software will stop with a timeout status when the modality graph reaches 25 nodes or the proof tree reaches 250 nodes.

Infinite tableaux will yield infinite countermodels, as shown in *Figure 28*. However, such problems do have finite countermodels (if disproved)[1; p.42-44]. I have developed an alternative countermodel finder algorithm in order to find finite countermodels for this use case.

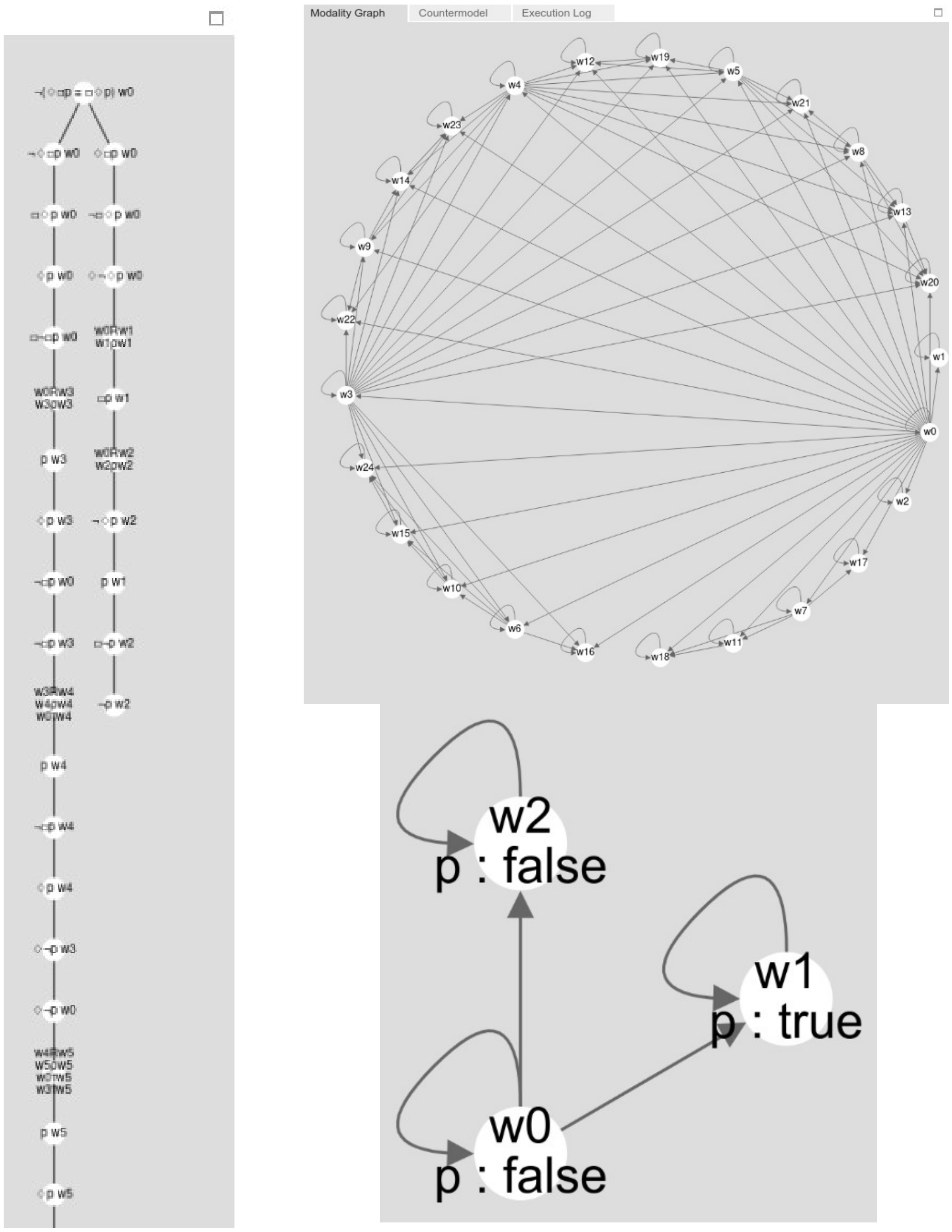


Figure 28 A finite part of an infinite proof tree (left), a finite part of its corresponding infinite countermodel (top right) and a finite countermodel found by the alternative algorithm (bottom right)

The alternative countermodel finder algorithm is available only on K, T, B, S4 and S5 propositional normal modal logics and their first-order (constant domain, contingent identity) counterparts. It works by reducing the model finding problem into a boolean satisfiability problem, which is solved using a third-party SAT solver[11]. A SAT solver is a software that efficiently finds atomic truth values that satisfy (make true) a set of classical propositional logic formulas.[12; p.27]

The algorithm follows these steps:

Given a problem with premises $P_1,...,P_n$ and conclusion C:

For each natural number $N \in [1...10]$:

- Generate all possible graphs with N nodes*
  - → If the logic is reflexive, keep only reflexive graphs. A graph is reflexive iff for each graph node w_i, there is a $w_i \rightarrow w_i$ vertex in the graph.
  - → If the logic is symmetric, keep only symmetric graphs. A graph is symmetric iff for each graph vertex $w_i \rightarrow w_j$ ($i \neq j$), there is another $w_j \rightarrow w_i$ vertex.
  - → If the logic is transitive, keep only transitive graphs. A graph is transitive iff for each vertex $w_i \rightarrow w_j$ and for each vertex $w_j \rightarrow w_k$ ($i \neq j \neq k$), there is another $w_i \rightarrow w_k$ vertex.
- For each graph G:
  - → If the logic is propositional, follow the steps below. If the logic is quantified, for each natural number $M \in [1...10]$, generate a domain with M unique objects. Then, for each domain D:
    - Let L be a list with all the premises $P_1,...,P_n$ and the non-conclusion $\neg C$
    - If the logic is quantified, eliminate quantifiers: transform all predicate logic formulas in L into propositional logic formulas.* The procedure is domain (D) dependent.
    - Eliminate modalities: transform all propositional modal formulas in L into classical propositional logic formulas.* This procedure is graph (G) dependent.
    - Transform all classical logic formulas in L into their conjunctive normal forms (CNF)[11].
    - Using a third-party SAT solver[11], try to find the solution of L. If no solution is found, continue searching. Otherwise, if a solution is found:
      - The solution will contain the truth values of all the atoms of all formulas of L.
      - Attach atomic values into G's nodes. Output G as found countermodel.

(*) As described on pages 31-33.

Graph generation procedure: *How to generate all graphs of N nodes?*

In order to generate all graphs of N nodes, I used the concept of adjacency matrix. An adjacency matrix is a square binary matrix that represents a graph[7; p.524]. Given a graph with N nodes, its adjacency matrix is a NxN matrix with the following contents: for each column index i, for each row index j, the value at coordinates (i,j) is 1 iff the graph has a $w_i \rightarrow w_j$ vertex, or 0 otherwise.

Given an adjacency matrix is a binary matrix, it can be encoded as a binary number, by placing each of its rows one after another. Thus, all adjacency matrices of all graphs of N nodes can be generated by iterating all binary numbers of N×N bits. This, in turn, is equivalent to iterating all natural numbers from 0 to $2^{N \times N}$-1.

Figure 29 shows the process of generating all possible graphs of 2 nodes, by iterating all codes in 0… $2^{2 \times 2}$-1 range and decoding them into graphs:

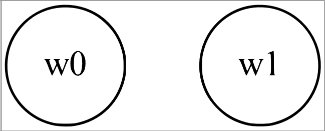
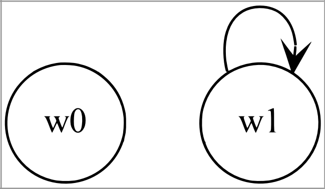
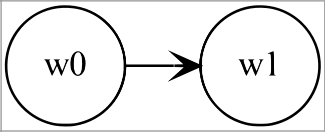
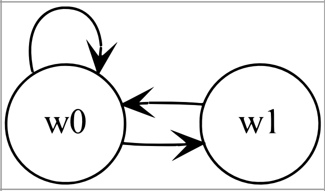
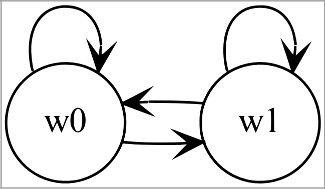
Code (decimal)	Code (binary)	Adjacency matrix	Generated graph
0	0000	00 00	
1	0001	00 01	
2	0010	00 10	
………………………	………………………	………………………	………………………
14	1110	11 10	
15	1111	11 11	

Figure 29: Generating all graphs of 2 nodes

Quantifier elimination procedure: *How to transform a quantified formula into a propositional one?*

I have developed the following algorithm, in order to eliminate quantifiers from a formula:

- Given a domain $\{a_1, a_2, ..., a_m\}$ and a formula, recursively apply the following rules:
  - R1: Turn $\exists x(P[x])$ into $P[a_1] \vee P[a_2] \vee ... \vee P[a_m]$
  - R2: Turn $\forall x(P[x])$ into $P[a_1] \wedge P[a_2] \wedge ... \wedge P[a_m]$
  - R3: Turn $P[a_1]$ into Pa_1, where $P[a_1]$ is a predicate, Pa_1 is a proposition
  - R4: Turn $a = a$ into T, $a = b$ into $\bot$, $\neg(a = a)$ into $\bot$, $\neg(a = b)$ into T

For instance, given a domain $\{a, b\}$, the formula $\forall x\ \exists y\ P[x,y] \supset x=y$ will be transformed as follows:

- Apply R2: $(\exists y\ P[a,y] \supset a=y) \wedge (\exists y\ P[b,y] \supset b=y)$
- Apply R1: $((P[a,a] \supset a=a) \vee (P[a,b] \supset a=b)) \wedge ((P[b,a] \supset b=a) \vee (P[b,b] \supset b=b))$
- Apply R3: $((P_{aa} \supset a=a) \vee (P_{ab} \supset a=b)) \wedge ((P_{ba} \supset b=a) \vee (P_{bb} \supset b=b))$
- Apply R4: $((P_{aa} \supset T) \vee (P_{ab} \supset \bot)) \wedge ((P_{ba} \supset \bot) \vee (P_{bb} \supset T))$

Modality elimination procedure: *How to transform a modal formula into a classical logic formula?*

I have developed the following algorithm, in order to eliminate modalities from a formula:

- Given a graph G and a formula, recursively apply these rules, assuming w_0 as initial world:
  - R5: Push w_i metadata inward, e.g. turn $(P \wedge Q), w_i$ into $(P, w_i) \wedge (Q, w_i)$
  - R6: Turn $\Diamond P, w_i$ into $\bot$ if G has no vertex $w_i \rightarrow w_j$
  - R7: Turn $\Box P, w_i$ into T if G has no vertex $w_i \rightarrow w_j$
  - R8: Turn $\Diamond P, w_i$ into $P,w_{j1} \vee P,w_{j2} \vee ... \vee P,w_{jn}$ if G has $w_i \rightarrow w_{jk}$ vertices
  - R9: Turn $\Box P, w_i$ into $P,w_{j1} \wedge P,w_{j2} \wedge ... \wedge P,w_{jn}$ if G has $w_i \rightarrow w_{jk}$ vertices
  - R10: Turn P, w_i into P_i if P is an atom.

For instance, given the graph of *Figure 30*, the $\Box(P \vee \Diamond Q)$ formula will be transformed as follows:

- Assume w0 as initial possible world: $\Box(P \vee \Diamond Q), w_0$
- Apply R9: $((P \vee \Diamond Q), w_1) \wedge ((P \vee \Diamond Q), w_2)$
- Apply R5: $((P, w_1) \vee (\Diamond Q, w_1)) \wedge ((P, w_2) \vee (\Diamond Q, w_2))$
- Apply R10: $(P_1 \vee (\Diamond Q, w_1)) \wedge (P_2 \vee (\Diamond Q, w_2))$
- Apply R8: $(P_1 \vee (Q, w_3)) \wedge (P_2 \vee ((Q, w_4) \vee (Q, w_5)))$
- Apply R10: $(P_1 \vee Q_3) \wedge (P_2 \vee (Q_4 \vee Q_5))$

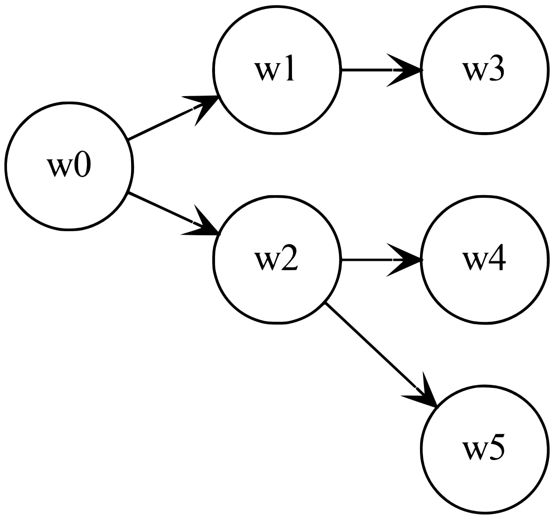


Figure 30: Modality elimination example graph

Hence, $\Box(P \vee \Diamond Q)$ is turned into $R \equiv (P_1 \vee Q_3) \wedge (P_2 \vee (Q_4 \vee Q_5))$. R is already in its conjunctive normal form. The SAT solver takes R as an input and outputs the solution: $P_1 = F$, $P_2 = T$, $Q_3 = T$, $Q_4 = F$, $Q_5 = F$. These are the values that make R true. Then, a countermodel is built by appending atomic values to graph's node. *Figure 31* shows the rendering of the found countermodel.

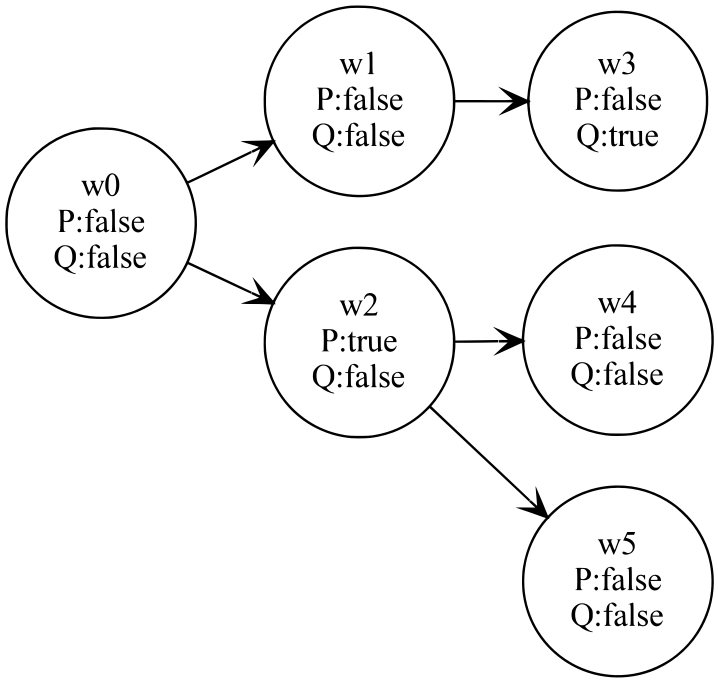


Figure 31: The resulting countermodel

Conclusion

In conclusion, I have developed an automated proof calculator (aka an automated theorem prover) for propositional and quantified, classical and non-classical logics, by implementing and adapting the tableaux proof systems of Priest[1]. I have extended the software with Łukasiewicz's fuzzy logic by implementing the tableaux proof system of Olivetti[8]. I have also developed an alternative counter-model finder algorithm for first-order normal modal logics.

During development, the software was thoroughly tested against all problems from Priest's book[1]. Correctness was carefully verified. Most solutions generated by the software were compared with Louis Barson's solutions[13]. After its initial release, the software was tested by roughly twenty users from University of Bucharest and City University of New York.

The software is free and open source, available online at https://andob.io. Its source code is published at https://github.com/andob/INCL-automated-theorem-prover. Future contributions are welcome!

Lastly, it is worth noting that "Philosophy is not a theory but an activity"[14; 4.112]. That is, it is not the theory which is the most important, but the act of living philosophically. This includes reflecting, debating, critically analyzing arguments, looking for definitions and foundations of what is *truth*, what is *knowledge*, what is *right* and what is *wrong*, wondering how to become a *virtuous* person and how to achieve a *good life*, or even searching for *meaning* in a *meaningless* world.

Mutatis mutandis, this could also be said about the craft of computer programming. That is, it is not the computer science theory which matters the most, but the actual computer engineering activity, that bridges the gap between theory and practice.

Annex I: Catalog of rules for propositional logics

Classical logic rules[1; p.6-9]

$$\begin{tikzcd} \neg\neg P \arrow[d] \\ P \end{tikzcd}$$

$$\begin{tikzcd} P \land Q \arrow[d] \\ P \arrow[d] \\ Q \end{tikzcd}$$

$$\begin{tikzcd} & P \lor Q \arrow[dl] \arrow[dr] & \\ P & & Q \end{tikzcd}$$

$$\begin{tikzcd} & \neg(P \land Q) \arrow[dl] \arrow[dr] & \\ \neg P & & \neg Q \end{tikzcd}$$

$$\begin{tikzcd} \neg(P \lor Q) \arrow[d] \\ \neg P \arrow[d] \\ \neg Q \end{tikzcd}$$

$$\begin{tikzcd} & P \supset Q \arrow[dl] \arrow[dr] & \\ \neg P & & Q \end{tikzcd}$$

$$\begin{tikzcd} & P \equiv Q \arrow[dl] \arrow[dr] & \\ P \arrow[d] & & \neg P \arrow[d] \\ Q & & \neg Q \end{tikzcd}$$

$$\begin{tikzcd} \neg(P \supset Q) \arrow[d] \\ P \arrow[d] \\ \neg Q \end{tikzcd}$$

$$\begin{tikzcd} & \neg(P \equiv Q) \arrow[dl] \arrow[dr] & \\ P \arrow[d] & & \neg P \arrow[d] \\ \neg Q & & Q \end{tikzcd}$$

Modal logic rules[1; p.24]

$$\begin{tikzcd} \Diamond P, w_i \arrow[d] \\ w_i \text{ ☆ } w_j \arrow[d] \\ P, w_j \end{tikzcd}$$

$$\begin{tikzcd} \Box P, w_i \arrow[d] \\ w_i \rightrightarrows w_j \arrow[d] \\ P, w_j \end{tikzcd}$$

$$\begin{tikzcd} \neg\Diamond P, w_i \arrow[d] \\ \Box\neg P, w_i \end{tikzcd}$$

$$\begin{tikzcd} \neg\Box P, w_i \arrow[d] \\ \Diamond\neg P, w_i \end{tikzcd}$$

$$\begin{tikzcd} & P \mathrel{\text{⥽}} Q, w_i \arrow[d] & \\ & w_i \rightrightarrows w_j \arrow[dl] \arrow[dr] & \\ \neg P, w_j & & Q, w_j \end{tikzcd}$$

$$\begin{tikzcd} \neg(P \mathrel{\text{⥽}} Q), w_i \arrow[d] \\ w_i \text{ ☆ } w_j \arrow[d] \\ P, w_j \arrow[d] \\ \neg Q, w_j \end{tikzcd}$$

Normal modal logics include: K, T, B, S4, S5

Non-normal modal logics include: S0.5, N, S2, S3, S3.5

On S0.5, $\Box P, w_i$ is applied iff $w_i=w_0$ (only w_0 is normal). In other non-normal modal logics, $\Box P, w_i$ is applied iff $w_i=w_0$ or there is another $\Box(\dots), w_i$ node on the branch.

Graph requirements:

	K	N	T	B	S0.5	S2	S3	S3.5	S4	S5
Reflexive	X	X	√	√	X	√	√	√	√	√
Symmetric	X	X	X	√	X	X	X	√	X	√
Transitive	X	X	X	X	X	X	√	√	√	√

Temporal modal logic rules[1; p.50]

K tense logic extends K modal logic with four modal operators instead of two: $\Diamond^f$, $\Box^f$, $\Diamond^p$, $\Box^p$ (possible / necessary in future / in past). Past operators will act backwards. Instead of creating a $w_i \rightarrow w_j$ vertex, $\Diamond^p$ will create a $w_j \rightarrow w_i$ vertex. Instead of iterating all w_j possible worlds from $w_i \rightarrow w_j$ vertices, $\Box^p$ will iterate all w_js from $w_j \rightarrow w_i$ vertices.

$$\begin{tikzcd}
\neg\Diamond^{p\text{ or }f}P, w_i \arrow[d] & \neg\Box^{p\text{ or }f}P, w_i \arrow[d] & \Diamond^{f}P, w_i \arrow[d] & \Diamond^{p}P, w_i \arrow[d] & \Box^{f}P, w_i \arrow[d] & \Box^{p}P, w_i \arrow[d] \\
\Box^{p\text{ or }f}\neg P, w_i & \Box^{p\text{ or }f}\neg P, w_i & w_i \text{☆} w_j \arrow[d] & w_j \text{☆} w_i \arrow[d] & w_i \rightrightarrows w_j \arrow[d] & w_j \rightrightarrows w_i \arrow[d] \\
 & & P, w_j & P, w_j & P, w_j & P, w_j
\end{tikzcd}$$

Conditional logic rules[1; p.86]

Conditional logic extends K modal logic by adding the conditional ($\triangleright$) operator. Each modality graph vertex will carry a tag. $\neg(P \triangleright Q), w_i$ create a $w_i \rightarrow w_j$ vertex tagged with P. $P \triangleright Q, w_i$ will iterate only $w_i \rightarrow w_j$ vertices tagged with P.

$$\begin{tikzcd}
\neg(P \triangleright Q), w_i \arrow[d] & P \triangleright Q, w_i \arrow[d] \\
\begin{array}{c} w_i \text{☆} w_j \\ \text{tag } w_i \rightarrow w_j \text{ vertex with P} \end{array} \arrow[d] & \begin{array}{c} w_i \rightrightarrows w_j \\ \text{only } w_i \rightarrow w_j \text{ vertices tagged with P} \end{array} \arrow[d] \\
\neg Q, w_j & Q, w_j
\end{tikzcd}$$

Intuitionist logic rules[1; p.108]

Intuitionist logic semantics maps over modal logic semantics. Tableaux rules are:

$$\begin{tikzcd}
P \land Q, w_i, + \arrow[d] \\
P, w_i, + \arrow[d] \\
Q, w_i, +
\end{tikzcd}$$

$$\begin{tikzcd}
& P \lor Q, w_i, + \arrow[dl] \arrow[dr] & \\
P, w_i, + & & Q, w_i, +
\end{tikzcd}$$

$$\begin{tikzcd}
& P \supset Q, w_i, + \arrow[d] & \\
& w_i \rightrightarrows w_j \arrow[dl] \arrow[dr] & \\
P, w_j, - & & Q, w_j, +
\end{tikzcd}$$

$$\begin{tikzcd}
\neg P, w_i, + \arrow[d] \\
w_i \rightrightarrows w_j \arrow[d] \\
P, w_j, -
\end{tikzcd}$$

$$\begin{tikzcd}
P, w_i, + \arrow[d] \\
w_i \rightrightarrows w_j \arrow[d] \\
P, w_j, +
\end{tikzcd}$$

$$\begin{tikzcd}
& P \land Q, w_i, - \arrow[dl] \arrow[dr] & \\
P, w_i, - & & Q, w_i, -
\end{tikzcd}$$

$$\begin{tikzcd}
P \lor Q, w_i, - \arrow[d] \\
P, w_i, - \arrow[d] \\
Q, w_i, -
\end{tikzcd}$$

$$\begin{tikzcd}
P \supset Q, w_i, - \arrow[d] \\
w_i \text{☆} w_j \arrow[d] \\
P, w_j, + \arrow[d] \\
Q, w_j, -
\end{tikzcd}$$

$$\begin{tikzcd}
\neg P, w_i, - \arrow[d] \\
w_i \text{☆} w_j \arrow[d] \\
P, w_j, +
\end{tikzcd}$$

$$\begin{tikzcd}
P, w_i, + \arrow[d, dash] \\
P, w_i, - \\
\text{↯}
\end{tikzcd}$$

First degree entailment (FDE)[1; p.144]

First degree entailment (FDE) rules acts as a set of base rules for 3-valued and 4-valued logics.

$$\begin{tikzcd}
\neg(P \land Q), w_i, \pm \arrow[d] \\
\neg P \lor \neg Q, w_i, \pm
\end{tikzcd}$$

$$\begin{tikzcd}
\neg(P \lor Q), w_i, \pm \arrow[d] \\
\neg P \land \neg Q, w_i, \pm
\end{tikzcd}$$

$$\begin{tikzcd}
\neg\neg P, w_i, \pm \arrow[d] \\
P, w_i, \pm
\end{tikzcd}$$

$$\begin{tikzcd}
P, w_i, + \arrow[d, dash] \\
P, w_i, - \\
\text{↯}
\end{tikzcd}$$

$$\begin{tikzcd}
P \wedge Q, w_i, + \arrow[d] \\
P, w_i, + \arrow[d] \\
Q, w_i, +
\end{tikzcd}
\qquad
\begin{tikzcd}
 & P \vee Q, w_i, + \arrow[dl] \arrow[dr] & \\
P, w_i, + & & Q, w_i, +
\end{tikzcd}
\qquad
\begin{tikzcd}
 & P \wedge Q, w_i, - \arrow[dl] \arrow[dr] & \\
P, w_i, - & & Q, w_i, -
\end{tikzcd}
\qquad
\begin{tikzcd}
P \vee Q, w_i, - \arrow[d] \\
P, w_i, - \arrow[d] \\
Q, w_i, -
\end{tikzcd}$$

Three-valued non-classical logics

Łukasiewicz (Ł3) logic rules[1; p.150]

Ł3 extend FDE rules and modal logic rules as follows:

$$\begin{tikzcd}
 & P \supset Q, w_i, + \arrow[dl] \arrow[d] \arrow[dr] & \\
\neg P, w_i, + & Q, w_i, + & P \vee \neg P, w_i, - \arrow[d] \\
 & & Q \vee \neg Q, w_i, -
\end{tikzcd}
\qquad
\begin{tikzcd}
 & P \supset Q, w_i, - \arrow[dl] \arrow[dr] & \\
P, w_i, + \arrow[d] & & \neg P, w_i, - \arrow[d] \\
Q, w_i, - & & \neg Q, w_i, +
\end{tikzcd}$$

$$\begin{tikzcd}
\neg(P \supset Q), w_i, + \arrow[d] \\
P, w_i, + \arrow[d] \\
\neg Q, w_i, +
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \neg(P \supset Q), w_i, - \arrow[dl] \arrow[dr] & \\
P, w_i, - & & \neg Q, w_i, -
\end{tikzcd}
\qquad
\begin{tikzcd}
P, w_i, + \arrow[d, no head] \\
P, w_i, - \\
\lightning
\end{tikzcd}
\qquad
\begin{tikzcd}
P, w_i, + \arrow[d, no head] \\
\neg P, w_i, + \\
\lightning
\end{tikzcd}$$

R-Mingle 3 (RM3) logic rules[1; p.150]

RM3 extend FDE rules and modal logic rules as follows:

$$\begin{tikzcd}
 & P \supset Q, w_i, + \arrow[dl] \arrow[d] \arrow[dr] & \\
P, w_i, - & \neg Q, w_i, - & P \wedge \neg P, w_i, + \arrow[d] \\
 & & Q \wedge \neg Q, w_i, +
\end{tikzcd}$$

$$\begin{tikzcd}
 & P \supset Q, w_i, - \arrow[dl] \arrow[dr] & \\
P, w_i, + \arrow[d] & & \neg P, w_i, - \arrow[d] \\
Q, w_i, - & & \neg Q, w_i, +
\end{tikzcd}$$

$$\begin{tikzcd}
\neg(P \supset Q), w_i, + \arrow[d] \\
P, w_i, + \arrow[d] \\
\neg Q, w_i, +
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \neg(P \supset Q), w_i, - \arrow[dl] \arrow[dr] & \\
P, w_i, - & & \neg Q, w_i, -
\end{tikzcd}$$

$$\begin{tikzcd}
P, w_i, + \arrow[d, no head] \\
P, w_i, - \\
\text{⚡}
\end{tikzcd}
\qquad
\begin{tikzcd}
P, w_i, - \arrow[d, no head] \\
\neg P, w_i, - \\
\text{⚡}
\end{tikzcd}$$

Kleene (K3) logic rules; Priest's logic of paradox (LP) rules[1; p.250]

K3 and LP extend FDE rules and modal logic rules as follows:

$$\begin{tikzcd}
\text{(on both logics)} \\
P, w_i, + \arrow[d, no head] \\
P, w_i, - \\
\text{⚡}
\end{tikzcd}
\qquad
\begin{tikzcd}
\text{(on K3 logic)} \\
P, w_i, + \arrow[d, no head] \\
\neg P, w_i, + \\
\text{⚡}
\end{tikzcd}
\qquad
\begin{tikzcd}
\text{(on LP logic)} \\
P, w_i, - \arrow[d, no head] \\
\neg P, w_i, - \\
\text{⚡}
\end{tikzcd}$$

Four-valued non-classical logics: Logics with gaps, gluts and worlds[1; p.165]

K4 and N4 logics extend FDE rules and modal logic rules as follows:

$$\begin{tikzcd}
& P \triangleright Q, w_i, + \arrow[d] & \\
& w_i \rightrightarrows w_j \arrow[dl] \arrow[dr] & \\
P, w_j, - & & Q, w_j, +
\end{tikzcd}$$

$$\begin{tikzcd}
P \triangleright Q, w_i, - \arrow[d] \\
w_i \text{☆} w_j \arrow[d] \\
P, w_j, + \arrow[d] \\
Q, w_j, -
\end{tikzcd}$$

$$\begin{tikzcd}
\neg(P \triangleright Q), w_i, + \arrow[d] \\
w_i \text{☆} w_j \arrow[d] \\
P, w_j, + \arrow[d] \\
\neg Q, w_j, +
\end{tikzcd}$$

$$\begin{tikzcd}
& \neg(P \triangleright Q), w_i, - \arrow[d] & \\
& w_i \rightrightarrows w_j \arrow[dl] \arrow[dr] & \\
P, w_j, - & & \neg Q, w_j, -
\end{tikzcd}$$

$$\begin{tikzcd}
P, w_i, + \arrow[d, dash] \\
P, w_i, - \\
\lightning
\end{tikzcd}$$

Logics of constructible negation[1; p.176]

I3, I4 and W logics extend FDE rules and modal logic rules as follows:

$$\begin{tikzcd}
& P \supset Q, w_i, + \arrow[d] & \\
& w_i \rightrightarrows w_j \arrow[dl] \arrow[dr] & \\
P, w_j, - & & Q, w_j, +
\end{tikzcd}$$

$$\begin{tikzcd}
P \supset Q, w_i, - \arrow[d] \\
w_i \text{☆} w_j \arrow[d] \\
P, w_j, + \arrow[d] \\
Q, w_j, -
\end{tikzcd}$$

(on I3,I4 logics)

$$\begin{tikzcd}
\neg(P \supset Q), w_i, + \arrow[d] \\
w_i \text{☆} w_j \arrow[d] \\
P, w_j, + \arrow[d] \\
\neg Q, w_j, +
\end{tikzcd}$$

(on I3,I4 logics)

$$\begin{tikzcd}
& \neg(P \supset Q), w_i, - \arrow[d] & \\
& w_i \rightrightarrows w_j \arrow[dl] \arrow[dr] & \\
P, w_j, - & & \neg Q, w_j, -
\end{tikzcd}$$

$$\begin{tikzcd}
P, w_i, + \arrow[d, dash] \\
P, w_i, - \\
\lightning
\end{tikzcd}$$

(on W logic)

$$\begin{tikzcd}
\neg(P \supset Q), w_i, \pm \arrow[d] \\
P \supset \neg Q, w_i, \pm
\end{tikzcd}$$

$$\begin{tikzcd}
P, w_i, + \arrow[d] \\
w_i \rightrightarrows w_j \arrow[d] \\
P, w_j, +
\end{tikzcd}$$

$$\begin{tikzcd}
\neg P, w_i, + \arrow[d] \\
w_i \rightrightarrows w_j \arrow[d] \\
\neg P, w_j, +
\end{tikzcd}$$

(on I3 logic)

$$\begin{tikzcd}
P, w_i, + \arrow[d, dash] \\
\neg P, w_i, + \\
\lightning
\end{tikzcd}$$

Infinite-valued non-classical logics

Łukasiewicz fuzzy logic rules[8]

$$\begin{tikzcd}
\alpha, \neg\neg P, \pm \arrow[d] & \alpha, P \wedge Q, + \arrow[d] & & \alpha, P \wedge Q, - \arrow[dl] \arrow[d] & \alpha, \neg(P \wedge Q), \pm \arrow[d] \\
\alpha, P, \pm & \alpha, P, + \arrow[d] & \alpha, P, - & \alpha, Q, - & \alpha, \neg P \vee \neg Q, \pm \\
 & \alpha, Q, + & & &
\end{tikzcd}$$

$$\begin{tikzcd}
\alpha, P, + \arrow[d] & & \alpha, P \vee Q, + \arrow[dl] \arrow[dr] & & \alpha, P \vee Q, - \arrow[d] & \alpha, \neg(P \vee Q), \pm \arrow[d] \\
\alpha \geq \mu P & \alpha, P, + & & \alpha, Q, + & \alpha, P, - \arrow[d] & \alpha, \neg P \wedge \neg Q, \pm \\
 & & & & \alpha, Q, - &
\end{tikzcd}$$

$$\begin{tikzcd}
\alpha, P, - \arrow[d] & & \alpha, P \supset Q, + \arrow[dl] \arrow[dr] & & \alpha, P \supset Q, - \arrow[d] & \alpha, \neg(P \supset Q), \pm \arrow[d] \\
\alpha < \mu P & \beta, P, - & & \alpha + \beta, Q, + & \beta, P, + \arrow[d] & \alpha, P \wedge \neg Q, \pm \\
 & & & & \alpha + \beta, Q, - &
\end{tikzcd}$$

$$\begin{tikzcd}
\alpha, \neg P, + \arrow[d] & \alpha, \neg P, - \arrow[d] & \alpha \geq \beta \arrow[d] \\
\alpha \geq 1 - \mu P & \alpha < 1 - \mu P & \alpha < \beta \\
 & & \lightning
\end{tikzcd}$$

Annex II: Notes for software developers

The project contains four Rust crates:

1. `prover`: the core component
2. `target-linux`: the prover as a command-line Linux application
3. `target-wasm`: the prover as a WebAssembly application
4. `benchmark`: various performance benchmarks.

The prover crate contains the following modules:

- `countermodel`: the main and the alternative countermodel finder algorithms
- `formulas`: formula data models and utilities
- `graph`: the modality graph data structure implementation
- `logic`: contains the implementations of all logics
- `parser`: a parser converting strings into formulas
- `problem`: problem data models and utilities
- `proof`: prover algorithm's core module
- `semantics`: various semantics (binary, many-valued, fuzzy)
- `tree`: the proof tree data structure implementation
- `utils`: various general-purpose utilities

The `book.json` file:

- Contains all problems available in the book, plus other extra problems.
- These problems appear in the *Problem catalog* UI sections.
- These problems are used for automatic testing.

Testing and building:

- Each time you modify the code, run `cargo test test_proof_status` and fix the bugs
- In order to build the WASM application, cd into `target-wasm` then run `build.sh`
- In order to deploy the WASM application, build it then copy the contents of `target-wasm/src` to your server. The server must serve these files as static resources.

In order to **edit an existing logic**, simply edit its corresponding module from `logic` directory:

- ➢ Classical propositional logic: the `propositional_logic.rs` file
- ➢ Modal logics: `common_modal_logic.rs, normal_modal_logic.rs, non_normal_modal_logic.rs`
- ➢ Many-valued logics: `first_degree_entailment.rs, FDE/kleene_modal_logic.rs` and so on
- ➢ First-order logics: `first_order_logic.rs`; Fuzzy logic: `fuzzy_logic.rs`; and so on...

In order to **add a new logic**, you should:

- ➢ If the logic requires new (not yet defined) operators:
  - → Add a new enum variant to the `Formula` enum. Build the project and fix the errors. Modify the codebase, implement the new case everywhere it is required.
  - → Modify the parser: add a new enum variant to the `TokenTypeID` enum. Implement a new token type in the `TokenType::get_types()` function.
- ➢ If the logic requires new (not yet defined) semantics:
  - → Add a new source file in `semantics/`. Define a struct. Implement the `Semantics` trait.
- ➢ Add a new source file in `logic/` directory. Define a struct. Implement the `Logic` trait.
  - → Make sure `get_name()` returns an unique name. Logic's name is used as an ID.
  - → Link the semantics to the logic via the `get_semantics()` function.
  - → Specify logic's syntax via the `get_parser_syntax()` function. Make sure you specify all available token type IDs (including atomics, open and closed parenthesis)
  - → Define a new struct for logic rules. Implement the `LogicRule` trait. Link the logic to its rules via the `get_rules()` function.
- ➢ Instantiate your logic in the `LogicFactory::get_logic_theories()` function.
- ➢ Add test problems to `book.json`. Run the `test_proof_status` test. Fix the bugs.

While modifying and extending the codebase, please carefully study and follow its design. Tips:

- ➢ Do not reinvent the wheel. Reuse existing components where possible.
- ➢ Declarative, functional style is preferred over imperative style.
- ➢ Use pattern matching and recursion extensively.
- ➢ Write clean, easy-to-follow, self-documenting code.

Contributions are welcome!

Bibliography

1. Graham Priest, *An Introduction to Non-Classical Logic. From If to Is (2nd edition)*. Cambridge University Press, 2008.
2. Steve Klabnik, Carol Nichols, *The Rust Programming Language (2nd edition)*. No Starch Press, 2023.
3. Graham Priest, *Logic: A Very Short Introduction (2nd edition)*. Oxford University Press, 2001.
4. William Kneale, Martha Kneale, *The Development of Logic*. Oxford University Press, 1985.
5. Arai Noriko, Pitassi Toniann, Urquhart Alasdair, *The Complexity of Analytic Tableaux*, *Journal of Symbolic Logic*, vol. 71, 2003.
6. Mircea Dumitru, *Possible minds, promises and limits of artificial intelligence*. Accessed: Jun. 26, 2024. [Online]. Available: https://youtu.be/jAHf1VLRGBE
7. Robert Sedgewick, Kevin Wayne, *Algorithms (4th edition)*. Addison-Wesley Professional, 2011.
8. Nicola Olivetti, *Tableaux for Łukasiewicz Infinite-valued Logic*, *Studia Logica*, vol. 73, 2003.
9. Alexey Zatelepin, *MiniLP*. Accessed: Nov. 2, 2024. [Online]. Available: https://github.com/ztlpn/minilp
10. Frederick S. Hillier, *Introduction to Operations Research (10th edition)*. McGraw-Hill Education, 2014.
11. BooleWorks GmbH, *LogicNG - The Next Generation Logic Framework*. Accessed: Dec. 20, 2024. [Online]. Available: https://logicng.org/

12. Daniel Kroening, Ofer Strichman, R.E. Bryant, *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.

13. Louis Barson, Graham Priest, *Solutions to Introduction to Non-Classical Logic*. Accessed: Apr. 26, 2025. [Online]. Available: https://grahampriest.net/publications/solutions-to-introduction-to-non-classical-logic

14. Ludwig Wittgenstein, *Tractatus Logico-Philosophicus*, 1921

Partial results of the research and development process underlying this work were publicly presented (prior to its submission) by myself and Marian Călborean at three academic events:

A) *Trees, algorithms and logics. A calculator for Priest's Introduction to Non-Classical Logics*, at *Logic seminar*, Institute for Logic and Data Science (ilds.ro), Faculty of Mathematics and Computer Science, University of Bucharest, Jun. 6, 2024

B) *Developing an Automated Proof Calculator for Modal Logic*, at *Perspectives about Truth III*, National University of Science and Technology Politehnica Bucharest, Nov. 15, 2024

C) *Trees, algorithms and logics II. Theoretical issues raised by a full theorem prover for Graham Priest's An Introduction to Non-classical Logic* at *Logic seminar*, Institute for Logic and Data Science (ilds.ro), Faculty of Mathematics and Computer Science, University of Bucharest, Feb. 27, 2025

All three presentations are available online at github.com/andob/INCL-automated-theorem-prover/tree/master/docs and figshare.com/authors/Marian_C_lborean/18615460.

These presentations should be regarded as works in progress that have contributed to the development of the current work.