Trees, algorithms and logics II.

Theoretical issues raised by a full theorem prover for Graham Priest's *An Introduction to Non-classical Logic*

Marian Călborean (mc@filos.ro) and Andrei Dobrescu (andrei.dobrescu@neurony.ro). 27.02.2025.

A. THE CALCULATOR

1. Introduction

Tableau proof system - 1/2

First proposed by Beth. Also inspired by Hintikka.

Reaches the current form in Smmulyan (1968) :
"Our tableaux, unlike those of Beth, use only one tree instead of two. Hintikka's tableau method also uses only one tree, but each point of the tree is a finite set of formulas, whereas in ours, each point consists of a single formula. The resulting combination has many advantages-indeed we venture to say that if this combination had been hit on earlier, the tableau method would by now have achieved the popularity it so richly deserves"

Proof that **A ⊃ B, B ⊃ C ⊢ A ⊃C**.
From Priest (2008)

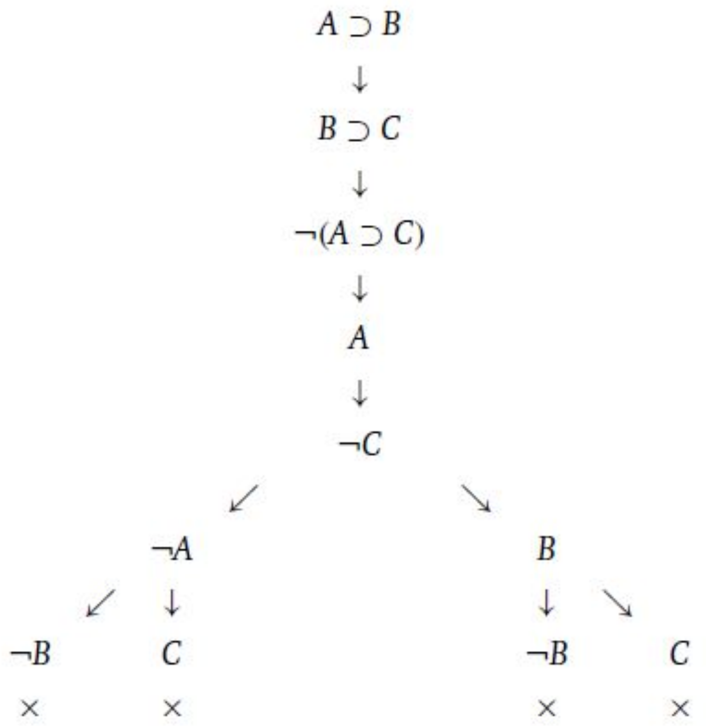


Tableau proof system - 2/2

Also called “semantic tableaux”, this system is strictly syntactic (see rules) just as axiomatic systems or natural deduction.

It does syntactically a counterexample search. Priest: “The tableau procedure is... a systematic search for an interpretation that makes all the formulas on the initial list true. Given an open branch of a tableau, such an interpretation can …be read off from the branch”

Priest on the advantages: “… 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. [...] Tableaux have great pedagogical attractions.”

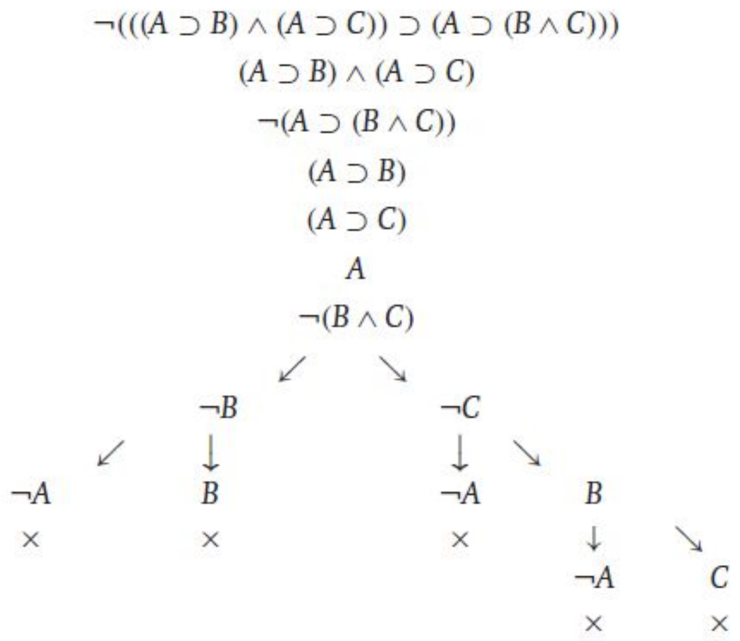
1) $\dfrac{\sim\sim X}{X}$

2) $\dfrac{X \wedge Y}{\begin{array}{c} X \\ Y \end{array}}$ $\qquad$ $\dfrac{\sim(X \wedge Y)}{\sim X \mid \sim Y}$

3) $\dfrac{X \vee Y}{X \mid Y}$ $\qquad$ $\dfrac{\sim(X \vee Y)}{\begin{array}{c} \sim X \\ \sim Y \end{array}}$

4) $\dfrac{X \supset Y}{\sim X \mid Y}$ $\qquad$ $\dfrac{\sim(X \supset Y)}{\begin{array}{c} X \\ \sim Y \end{array}}$

Above, the rules of Smullyan 1968.
Below, a proof from Priest (2008)



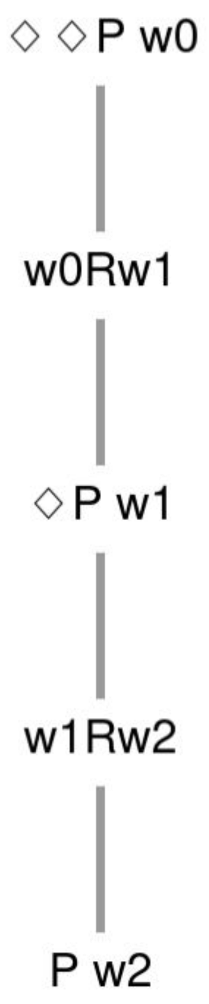
Tableaux in Priest (2008)

Priest's *Introduction to Non-Classical Logics: From If to Is* (2008, Cambridge University Press)

- More than 20 logics in the book, most with possible-world semantics
- The **exact counting is impossible**, since the book indicates how rules can be combined so as to result in new logics (e.g. accessibility relations in modal logic)
- Priest introduces **original variations of the tableau method**, discussed in the literature (e.g. Johnson 2015)

The general tableux-adaptation framework comes from "Labelled Deductive System" (Gabbay 1994) according to the slogan "bring the semantics into the syntax through the labels".

As Olivetti puts it "formulas are equipped with algebraic labels and the rules manipulate both formulas and their labels"



Propositional Logics in INCL - Priest (2008)

	Chapter	Status
1	Classical logic	✅ Propositional 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: for temporal convergence rule, multiple graphs per problem are needed, right now there is a single graph per problem.
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, multiple graphs per problem are needed.
6	Intuitionist logic	✅ Fully implemented.
7	Many-valued logics	✅ Skip, no tableaux on 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	❌ Skip, this is really difficult to implement.
11	Fuzzy logics	✅ Skip, no tableaux on this chapter.
11a	Many-valued modal logics	✅ Lukasiewicz logic, Kleene logic, Logic of Paradox, RMingle3 logic fully implemented.

Quantified Logics in INCL - Priest (2008)

	Chapter	Status
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	❌ Skip, this is really difficult to implement.
25	Fuzzy logics	✅ Skip, no tableaux on this chapter.

Proof $\vdash a = b \supset (\exists xSxa \supset (a = a \land \exists xSxb))$ *From Priest (2008)*

$$\neg(a = b \supset (\exists xSxa \supset (a = a \land \exists xSxb)))$$
$$a = b$$
$$\neg(\exists xSxa \supset (a = a \land \exists xSxb))$$
$$\exists xSxa$$
$$\neg(a = a \land \exists xSxb)$$
$$Sca$$
$$\swarrow \qquad \searrow$$
$$\neg a = a \qquad \neg\exists xSxb$$
$$\times \qquad \forall x\neg Sxb$$
$$\neg Scb$$
$$Scb$$
$$\times$$

2. Towards a calculator

Tree and nodes

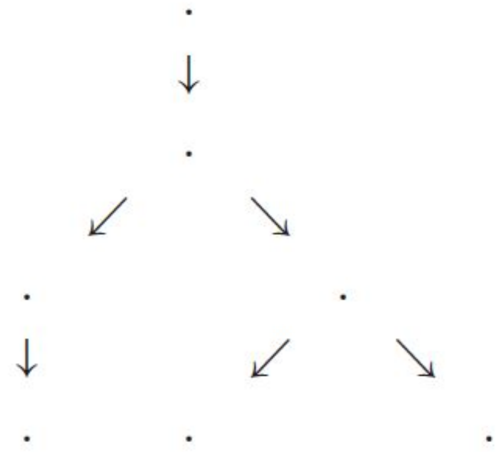
"… a partial order with a unique maximum element, x_0, such that for any element, x_n, there is a unique finite chain of elements $x_n \leq x_{n-1} \leq \cdots \leq x_1 \leq x_0$. " (Priest)

Tree structures:

- **Root node.** In effect there can be more than one: premises and negated conclusion
- **Branch** (finite chain from tip to root). A node can split in two or three etc
- **Tips**. Nodes with no child nodes

Types of nodes *by information* (our classification):

- **Logical formulas**: this is the standard node: "[P]<F>A"
- **Labels with semantic value**. Examples**:**
  - In modal logics node "irj" hints that world *i accesses world j.*
  - In relevant logic B node "$i" hints that world i is normal
- **Combination of formula and some metadata** of semantic value:
  - In modal logics "A V B, i" hints that A V B holds at world i
  - In intuitionistic logic "A V B, -i" hints that A V B is false at world i



Above, a tree. Below, a proof in tense logic K^t that A ⊢ [P]<F>A. Both from Priest (2008)

$A, 0$

$\neg[P]\langle F\rangle A, 0$

$\langle P\rangle \neg \langle F\rangle A, 0$

$1r0$

$\neg \langle F\rangle A, 1$

$[F]\neg A, 1$

$\neg A, 0$

$\times$

Formulas and rules

Formulas of the language (**operators** and **predicates** vary with the logic)
- Modal logics may add dyadic ⥽ and monadic □, ◇, [P], <P>, [F], <F>.
- Free logic adds special predicate $\mathfrak{E}$ etc.

Derivation rules *(our classification)*

a) **Formula-based**
- E.g. From A V B, we split the tree for A and for B
- They *consume* nodes (once applied, nodes can be ignored)

b) **Label-based**
- E.g. if the logic is transitive, for any nodes of the form irj and jrk, add irk
- They *consume* nodes but need care when applied (e.g. rule η after others)

c) **Complex: both formula- and label- based**
- E.g. □ is reapplied for any newly accessible world label
- In non-normal *N*, ◇ is applied only at world 0 or if there's a □-formula on the branch
- These *do not consume nodes (in general)* and need care.

The rules for □, ◇, then for extendability, reflexivity, symmetry, transitivity. Finally, an infinite tableau in K_τ proving that $\nvdash \neg(\Diamond p \land \Box\Diamond p)$. All from Priest (2008)

$$\begin{array}{c} \Box A, i \\ irj \\ \downarrow \\ A, j \end{array} \qquad \begin{array}{c} \Diamond A, i \\ \downarrow \\ irj \\ A, j \end{array}$$

$$\begin{array}{c} \eta \\ \cdot \\ \downarrow \\ irj \end{array} \qquad \begin{array}{c} \rho \\ \cdot \\ \downarrow \\ iri \end{array} \qquad \begin{array}{c} \sigma \\ irj \\ \downarrow \\ jri \end{array} \qquad \begin{array}{c} \tau \\ irj \\ jrk \\ \downarrow \\ irk \end{array}$$

$$\begin{array}{c} \neg\neg(\Diamond p \land \Box\Diamond p), 0 \\ \Diamond p \land \Box\Diamond p, 0 \\ \Diamond p, 0 \\ \Box\Diamond p, 0 \\ 0r1 \\ p, 1 \\ \Diamond p, 1 \\ 1r2 \\ p, 2 \\ 0r2 \\ \Diamond p, 2 \\ 2r3 \\ p, 3 \\ \vdots \end{array}$$

Metalogical Notions – 1/2

Closing and counterexamples

- A branch is (**atomically) closed** iff it contains an (atomic) formula and its negation. Mark it with X.
- A tree is **(atomically) closed** iff all branches are (atomically) closed. If closed, it is atomically closed.
- If a tree does not close, one gets a **counterexample** by assigning values (1 / 0) on an open branch

Note that the tableau method is **sound** and **complete,** but the rules as given have **exponential complexity**.

For our calculator (1):

a) We stop work on a branch at X (any two contradictory atomic or $\Box$ / $\Diamond$-formulas)

b) We order the rules as required and to save space: identities first, non-splitting rules before splitting ones, possibility after necessity, universal instantiation last etc

Metalogical Notions – 2/2

Infinite tree in Kη for ⊬ □p. Corresponding countermodel. Simpler countermodel findable by trial and error. All from Priest (2008)

$$\begin{array}{c} \neg\Box p, 0 \\ \Diamond\neg p, 0 \\ 0r1 \\ \neg p, 1 \\ 1r2 \\ 2r3 \\ \vdots \end{array}$$

An algorithm is **fair** if each rule application that could be made eventually is. In classical logic and modal logic K, a fair algorithm always terminates (proved).

However, in other modal logics and quantified logics:

- For a valid inference, the tree will close, hence the algo will stop.
- Else it may not stop, running into **an infinite counterexample** (see at right)

$$\begin{array}{ccccccc} & & \neg p & & & & \\ w_0 & \rightarrow & w_1 & \rightarrow & w_2 & \rightarrow & \cdots \end{array}$$

For our calculator (2):
a) The algorithm is fair (it iterates ordered rules and has only finite premises). It can result in PROVED, DISPROVED or TIMEOUT

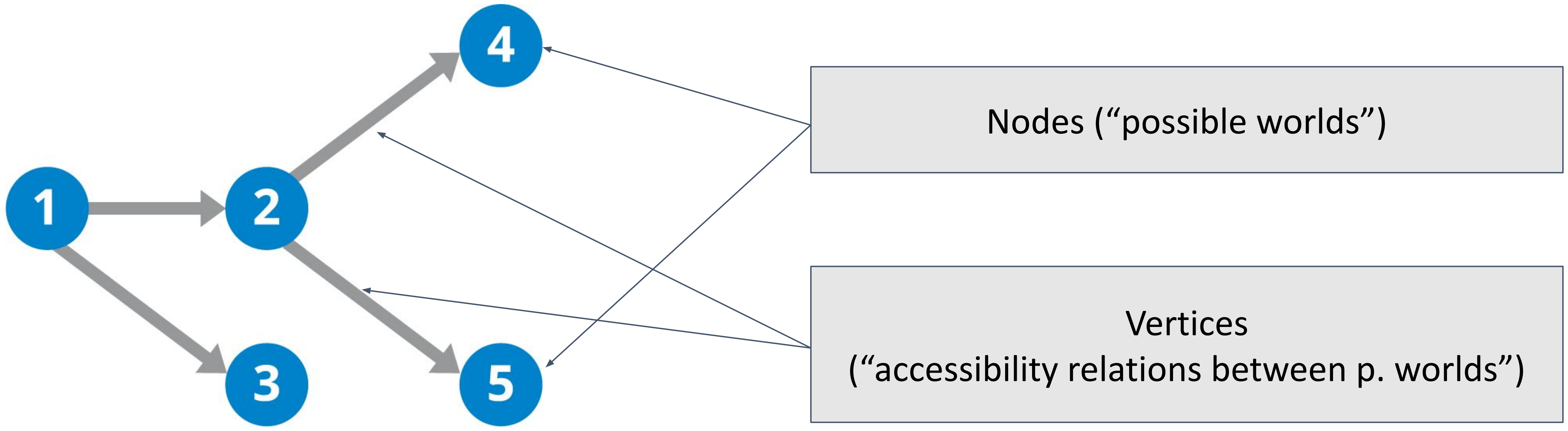
b) We time out after some limits, to prevent the program hanging. Right now, the limits are 25 worlds or 250 nodes for FOL (but 1000 at intuitionistic logic)

$$\begin{array}{c} \curvearrowright \\ w_0 \\ \neg p \end{array}$$

c) After timeout, there’ a **SAT-based search** for finite countermodels (see section B2.)

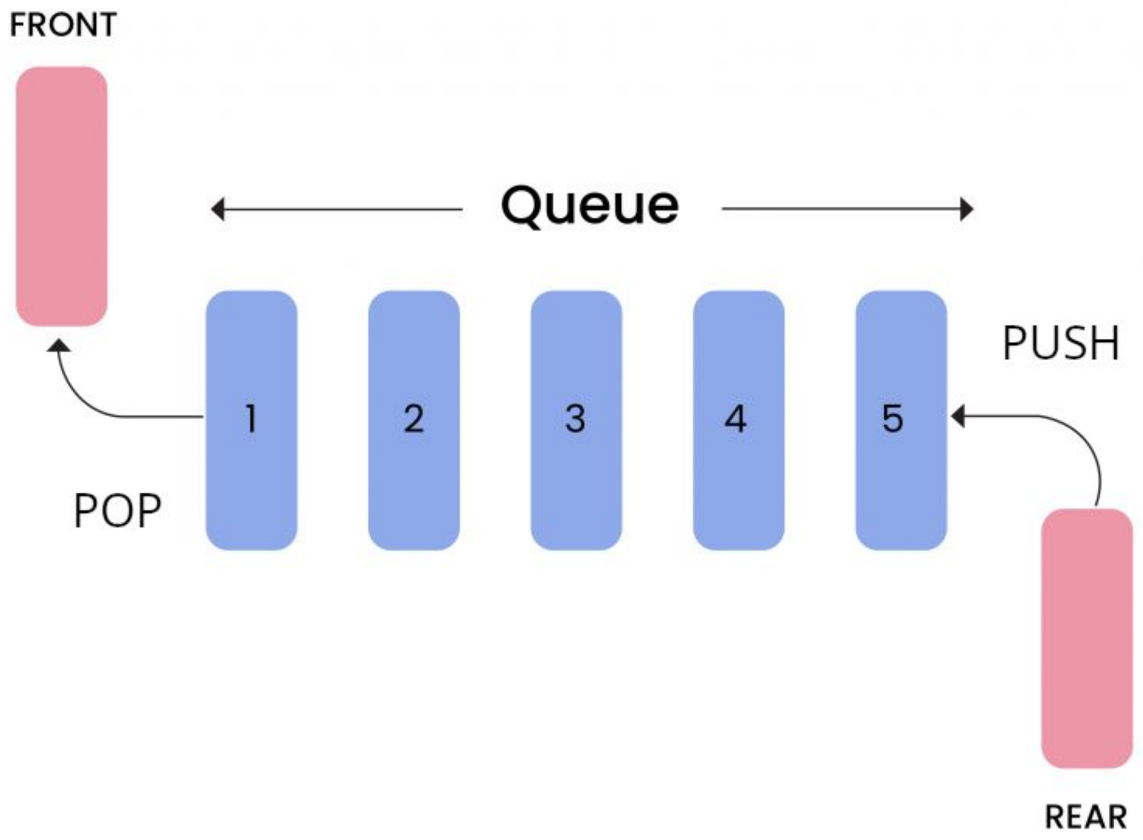
Directed graph (data structure)

- A directed graph is a data structure consisting of a set of **nodes** and a set of **vertices** („arrows” between one node to another).
- Directed graphs can be mapped directly to the concept of **Kripke model**.
- Priest (2008) does not use graphs, only trees. In the tree, possible worlds and their relations are described via w_iRw_j labels. But by using a separate possible world graph, the **code is easier to follow, maintain and extend.**



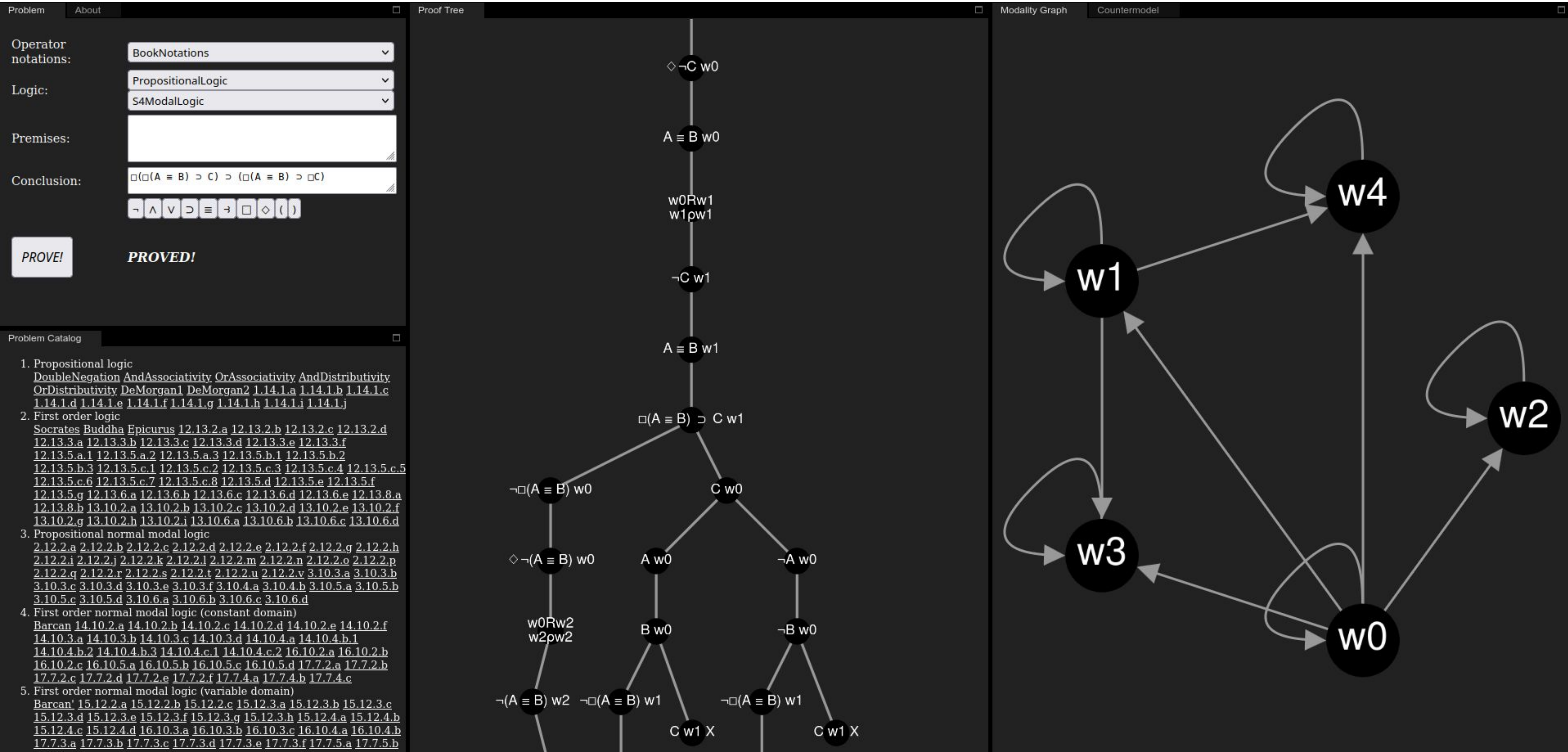
Queue (data structure)

- FIFO (“first in first out”)
- On the rear of the queue, the program pushes formulas (starting with the premises and negation of the conclusion).
- From the front, the program will pop and process formulas one by one according to the priority defined on main operators.
- The program stops when the queue becomes empty (no formulas left).
- Thus, the queue implements:
  - the **iteration** requested by the concept of fair algorithm
  - the idea that the **program ends** when no rules are available.



3. The calculator

Screenshot


Problem
About
Operator notations:
BookNotations
Logic:
PropositionalLogic
S4ModalLogic
Premises:
Conclusion:
□(□(A ≡ B) ⊃ C) ⊃ (□(A ≡ B) ⊃ □C)
¬ ∧ ∨ ⊃ ≡ → □ ◇ ( )
PROVE!
PROVED!
Problem Catalog
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 Epicurus 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 2.12.2.p 2.12.2.q 2.12.2.r 2.12.2.s 2.12.2.t 2.12.2.u 2.12.2.v 3.10.3.a 3.10.3.b 3.10.3.c 3.10.3.d 3.10.3.e 3.10.3.f 3.10.4.a 3.10.4.b 3.10.5.a 3.10.5.b 3.10.5.c 3.10.5.d 3.10.6.a 3.10.6.b 3.10.6.c 3.10.6.d
4. First order normal modal logic (constant domain)
Barcan 14.10.2.a 14.10.2.b 14.10.2.c 14.10.2.d 14.10.2.e 14.10.2.f 14.10.3.a 14.10.3.b 14.10.3.c 14.10.3.d 14.10.4.a 14.10.4.b.1 14.10.4.b.2 14.10.4.b.3 14.10.4.c.1 14.10.4.c.2 16.10.2.a 16.10.2.b 16.10.2.c 16.10.5.a 16.10.5.b 16.10.5.c 16.10.5.d 17.7.2.a 17.7.2.b 17.7.2.c 17.7.2.d 17.7.2.e 17.7.2.f 17.7.4.a 17.7.4.b 17.7.4.c
5. First order normal modal logic (variable domain)
Barcan' 15.12.2.a 15.12.2.b 15.12.2.c 15.12.3.a 15.12.3.b 15.12.3.c 15.12.3.d 15.12.3.e 15.12.3.f 15.12.3.g 15.12.3.h 15.12.4.a 15.12.4.b 15.12.4.c 15.12.4.d 16.10.3.a 16.10.3.b 16.10.3.c 16.10.4.a 16.10.4.b 17.7.3.a 17.7.3.b 17.7.3.c 17.7.3.d 17.7.3.e 17.7.3.f 17.7.5.a 17.7.5.b
Proof Tree
◇¬C w0
A ≡ B w0
w0Rw1
w1ρw1
¬C w1
A ≡ B w1
□(A ≡ B) ⊃ C w1
¬□(A ≡ B) w0
C w0
◇¬(A ≡ B) w0
A w0
¬A w0
w0Rw2
w2ρw2
B w0
¬B w0
¬(A ≡ B) w2
¬□(A ≡ B) w1
C w1 X
¬□(A ≡ B) w1
C w1 X
Modality Graph
Countermodel
w4
w1
w2
w3
w0

Introduction

Written in Rust, available at andob.io/incl

Open source: source code available on GitHub

There is a first draft version (in Kotlin) available here (source code available here)

Dependencies - libraries

Rust compiles to WebAssembly

GoldenLayout for *windowing layout*

Cytoscape.JS graph & tree visualizer

Serde JSON serializer and deserializer

MiniLP linear optimization problem solver - *for fuzzy logic*

LogicNG, which includes a port of the MiniSAT SAT solver

Other small Rust utility libraries

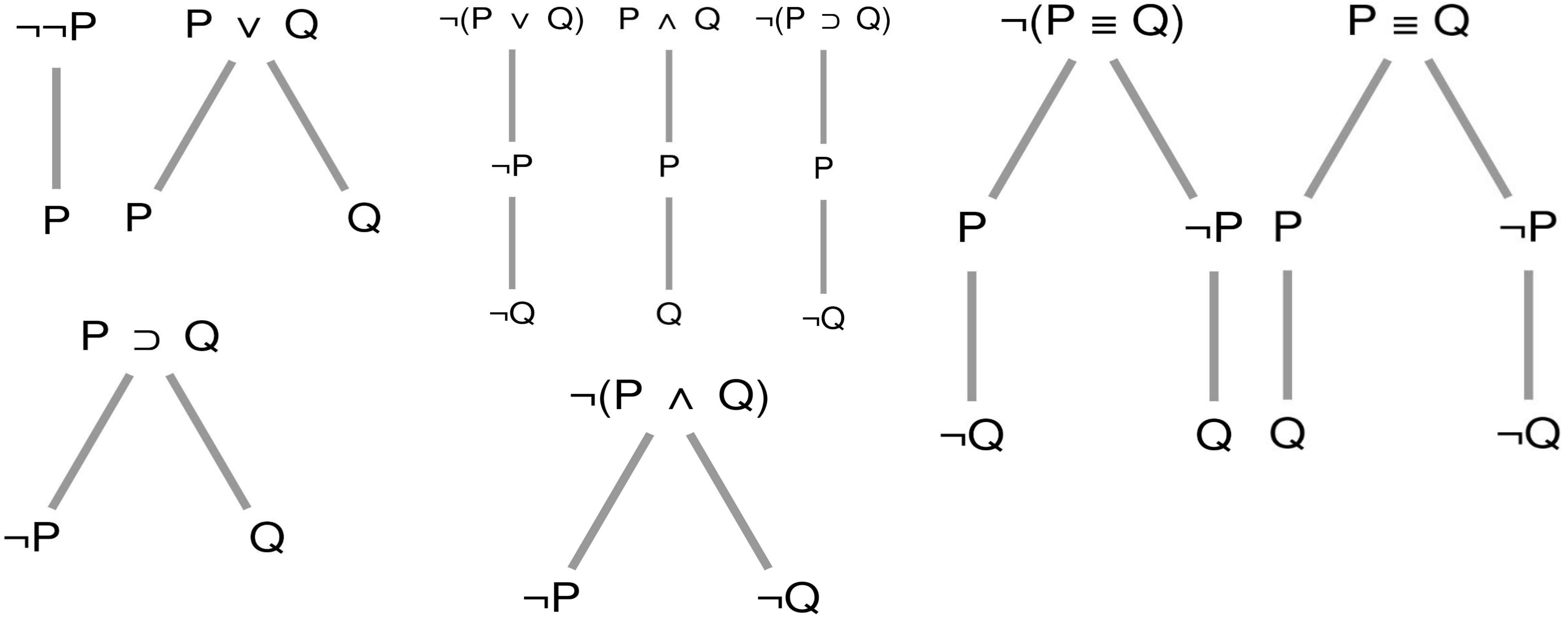
How it works?

- The software takes a *Problem* as input and outputs a *ProofTree*.
- The *Problem* consists of a **logic**, **0..n premises** and a **conclusion**.
- The software can either **prove** or **disprove** the problem or **timeout** while trying to solve the problem.
- A ***ProofTree*** is formed by following specific rules.
- ***Problem*** is proved if the ***ProofTree*** has contradiction on all branches.
- The software will timeout (“TIMEOUT” displayed) above a limit - if the *ProofTree* reaches 250 nodes.
- If the ***Problem*** is disproved, the program can search for a counterexample (finite, see B2)

The algorithm in a nutshell - 1/4

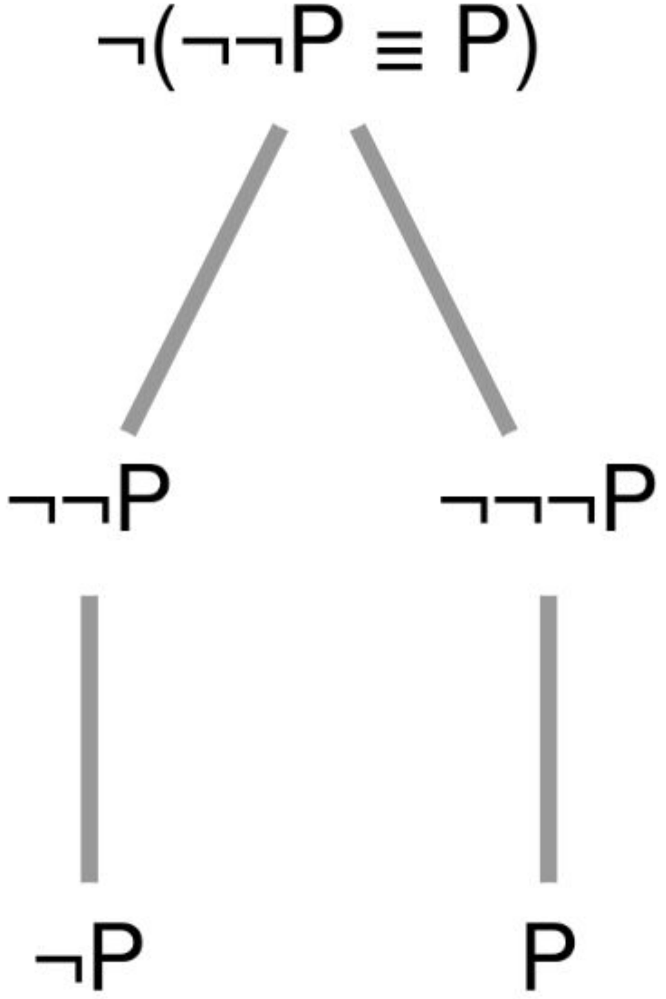
- Initial step: given a *Problem* (logic, premises, conclusion):
  - The program creates a *ProofTree* with root node = non-conclusion, and subnodes for each premise.
  - The program creates a *queue* with these formulas.
- While the *queue* is not empty:
  - Pop one formula from the queue.
  - Find the specific applicable *logic* rule for this formula (each logic has different rules, see next slide).
  - Apply the rule and append the result to the *ProofTree*.
  - Push the resulting formulas to the *queue*.
  - Check for contradictions in the *ProofTree*.

Classical logic rules



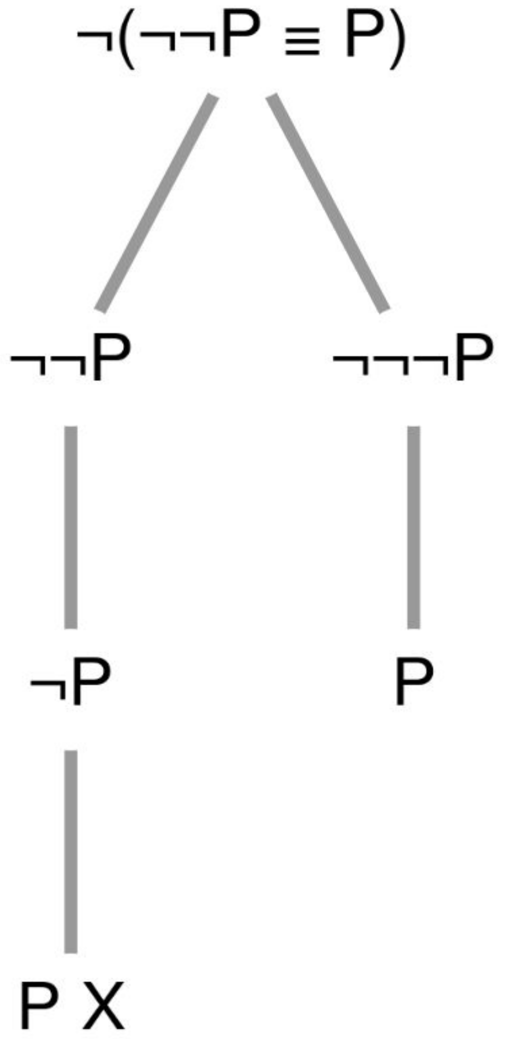
The algorithm in a nutshell - 2/4

- For instance, let's prove that ¬¬P≡P
- Initialization:
  - proofTree = { rootNode: ¬(¬¬P≡P) }
  - queue = { ¬(¬¬P≡P) }
- First iteration: since queue is not empty:
  - The ¬(¬¬P≡P) formula is popped from the queue
  - The queue becomes empty
  - The ¬≡ rule is applied (result: 4 formulas)
  - The result is appended to the tree
  - Check for contradictions: no contradictions
  - All resulting formulas are pushed to the queue
  - The queue becomes { ¬¬P, ¬P, ¬¬¬P, P }



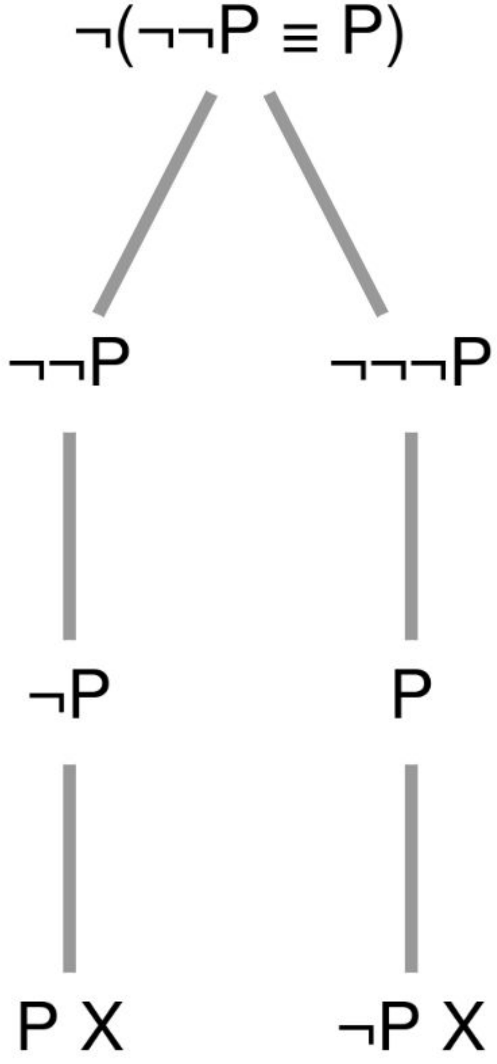
The algorithm in a nutshell - 3/4

- Second iteration
  - The ¬¬P formula is popped from the queue.
  - The queue becomes { ¬P, ¬¬¬P, P }
  - The double negation rule is applied
  - Check for contradictions: found a contradiction
  - Add P to queue. The queue becomes { ¬P, ¬¬¬P, P,P }
- Third iteration
  - The ¬P formula is popped from the queue.
  - The queue becomes { ¬¬¬P, P, P }
  - There is no rule for ¬P. Skip this iteration.



The algorithm in a nutshell - 4/4

- Fourth iteration
  - The ¬¬¬P formula is popped from the queue.
  - Queue: { P, P }
  - The double negation rule is applied
  - Check for contradictions: found another contradiction
  - Add ¬P to queue. The queue becomes { P, P, ¬P }
- 5th / 6th / 7th iteration
  - Pop the queue: { P, P, ¬P }.
  - There are no rules to apply. Skip.
- The queue is empty now. Nothing left to do.
  - We have contradiction on all branches
  - Initially we assumed ¬(¬¬P≡P)
- This can't be right, thus ¬¬P≡P is true ∎

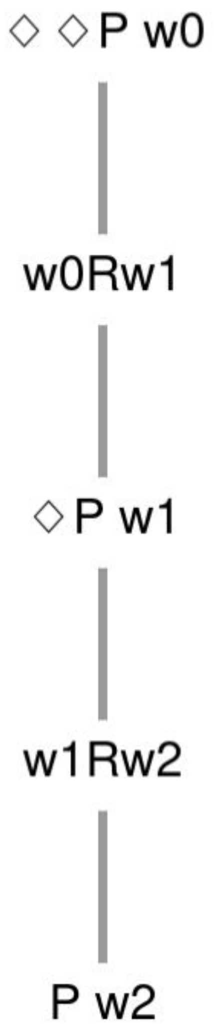


Modal logic rules

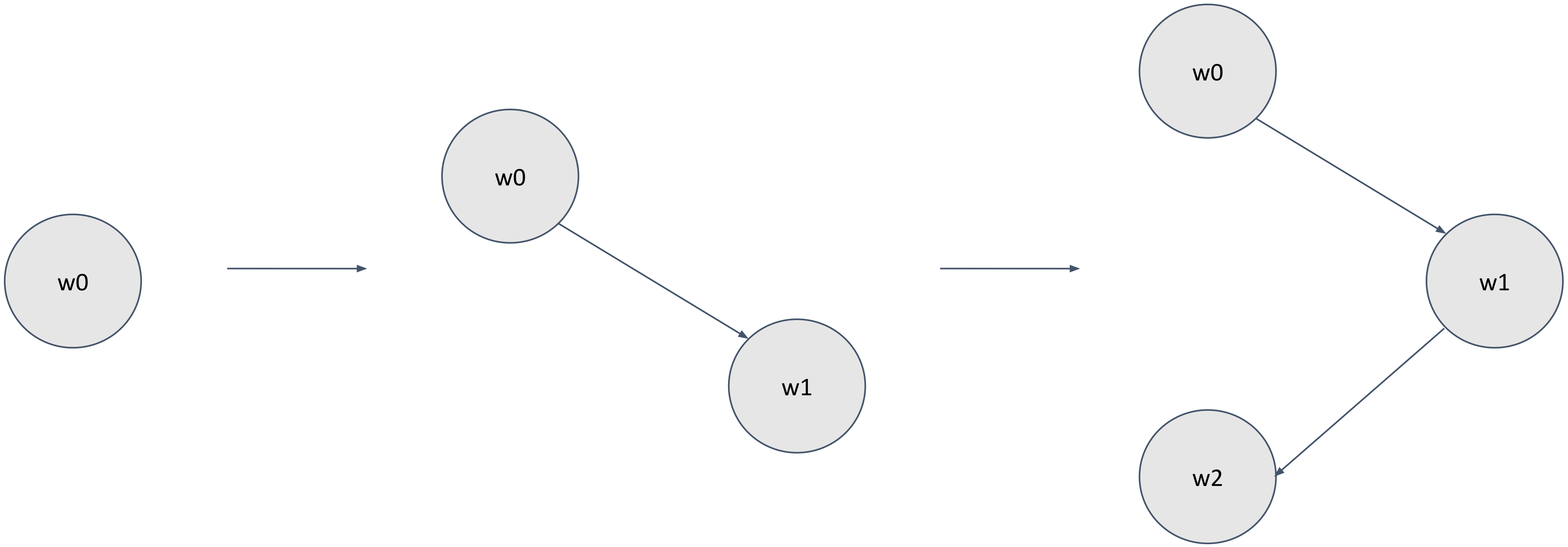
$$
\begin{tikzcd}
\Diamond P, w_i \arrow[d] & \Box P, w_i \arrow[d] & \neg \Diamond P, w_i \arrow[d] & \neg \Box P, w_i \arrow[d] & P, w_i \arrow[d, dash] \\
w_i \medstar w_j \arrow[d] & w_i \rightrightarrows w_j \arrow[d] & \Box \neg P, w_i & \Diamond \neg P, w_i & \neg P, w_i \\
P, w_j & P, w_j & &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

Legend:
☆ spawns a new possible world
⇉ iterates all accessible possible worlds

◇ *Possibility rule - 1/2*

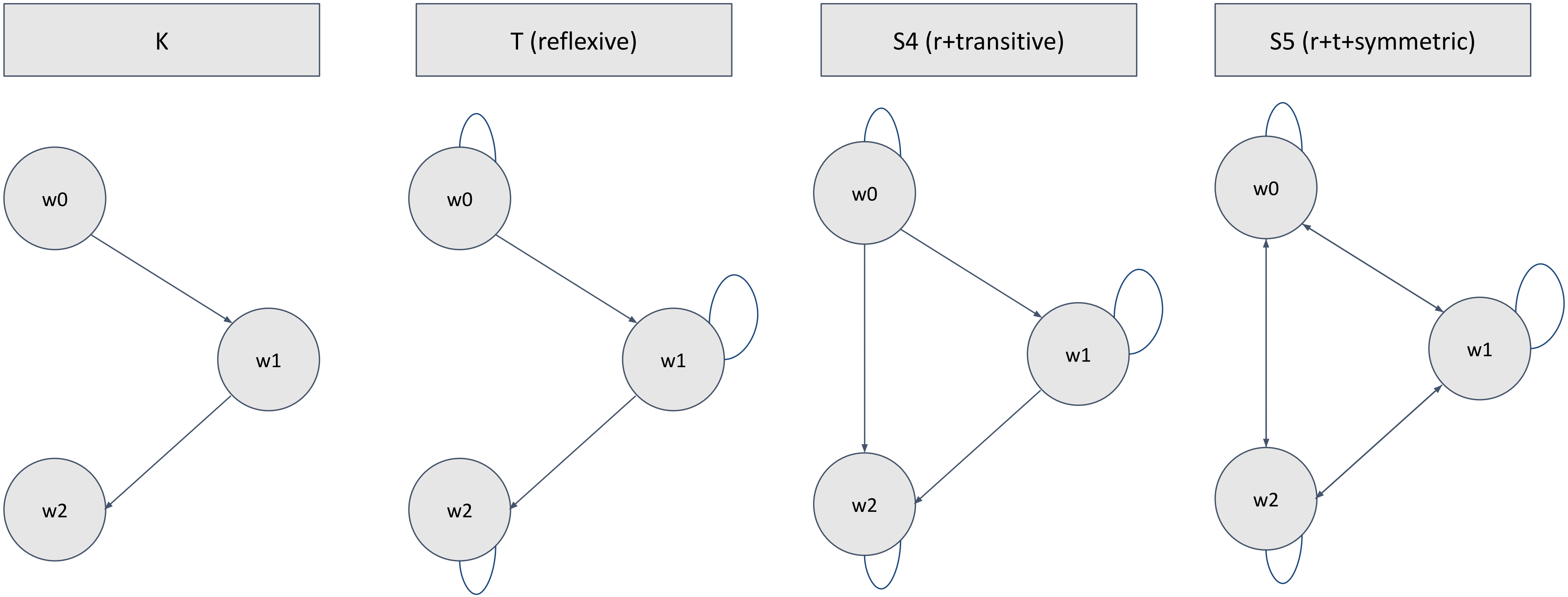


- ◇ rule will spawn a new possible world from the current world.
- ◇ rule will add new nodes and new vertices to the graph.
- ◇ rule will add wiRwj nodes to the tree for each new vertex.

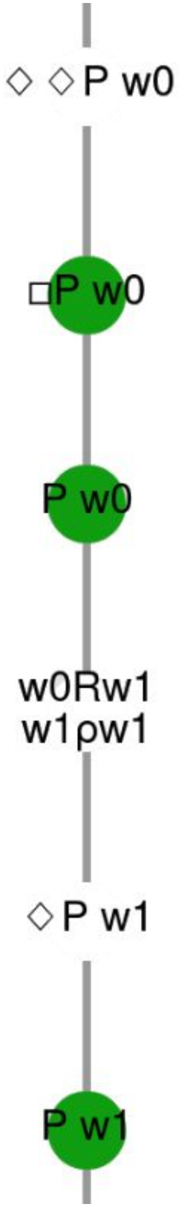


◇ *Possibility rule - 2/2*

◇ rule will also add additional graph vertices, as follows:

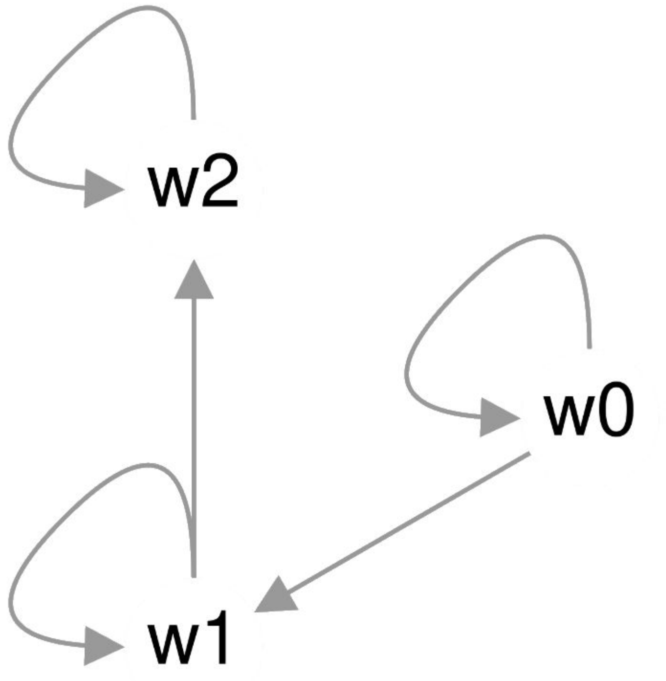


□ *Necessity rule*



□ rule will iterate all graph vertices that start from the current world:

- Example: □P w0 is applied here.
- There are two graph vertices w0→w0 and w0→w1. Worlds w0 and w1 are accessible from world w0.
- Thus the rule will add two nodes to the tree: P w0 and P w1.



Modal logic example

Proving:

$\square A \equiv \neg\Diamond\neg A$

(link)

First-Order logic notations

For easy-parsing, the software uses non-standard notations:

- A predicate with one argument: P[x] (instead of Px)
- A predicate with 2 arguments: P[x,y] (instead of Pxy)
- A predicate with n arguments: P[x_1,x_2,…,x_n] (instead of P$x_1x_2...x_n$)

The software categorizes arguments as follows:

- *Free variables* (in P[a] ∧ ∃x Q[x], a is a free variable).
- *Binding variables* (in P[a] ∧ ∃x Q[x], x is a binding variable).
- *Instantiated objects* (in P[a] ∧ Q[b:x], b:x is an instantiated object).
  - b:x notation means „an object named b of type x”

∃ *Existence rule*

- The ∃ rule will transform *binding variables* into *instantiated objects*.
- The rule will generate unique names, considering already existing names on the tree branch.
- Objects are uniquely identified by their names.
- The rule will also attach a type on each object. Type's name = the name of the original variable.
  - Binding variables		Instantiated objects
  - x		a:x
  - y		b:y
  - z		c:z

∃x(∃y(∃z(P[x,y,z]))) w0

∃y(∃z(P[a:x,y,z])) w0

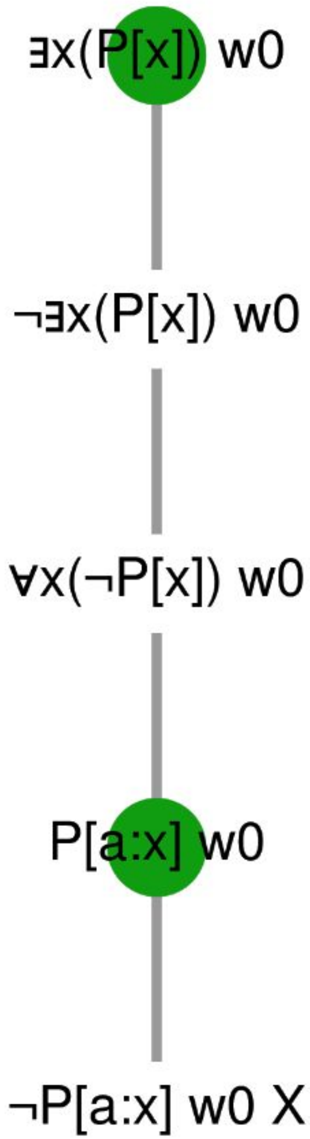
∃z(P[a:x,b:y,z]) w0

P[a:x,b:y,c:z] w0

∀ For All rule

- The ∀x rule will find on tree branch all instantiated objects of type x. For each object, the rule will add to the tree a node, replacing x with the object.

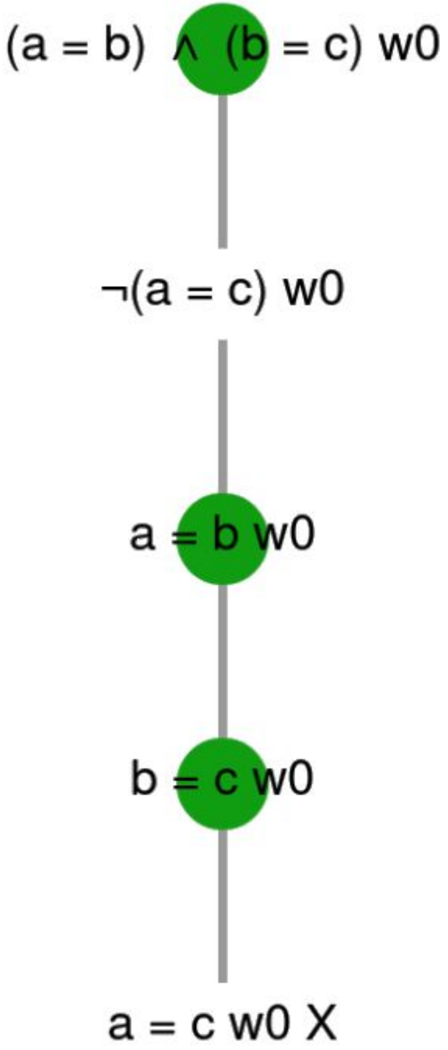
- For instance, in this example:
  - ∃x(P[x]) instantiates x as a:x and adds a P[a:x] node to the tree branch.
  - ∀x(¬P[x]) finds one object of type x (the previously instantiated a:x) and adds one node (¬P[a:x]) to the branch



Equality and Contradiction

- The algorithm supports equalities between a variable / object and another variable / object.
- Transitive equalities are automatically generated: for each $<o_1=o_2, o_2=o_3>$ equality pair, an $o_1=o_3$ equality node will be added, if not already present on branch.

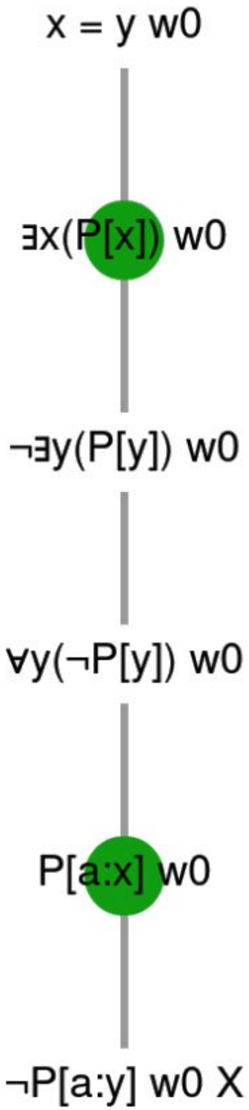
- In FOL, the algorithm detects a contradiction iff:
  - There are two nodes P[a:x1,b:x2,…] and ¬P[a:x1,b:x2,…] (same arguments) on the branch.
  - Or there are two nodes a:x1 = b:x2 and ¬(a:x1 = b:x2) (same objects) on the branch.



Equality and ∀ For All rule

- The ∀x rule will find on tree branch all instantiated objects of type x and any other y type equal to x (x = y).
- For each object, the rule will add a node to the tree branch.

- For instance, in this example:
  - ∃x(P[x]) instantiates x as a:x and adds a P[a:x] node.
  - ∀y(¬P[y]) finds no objects of type y, nothing to add yet.
    - The type y has an equivalent type x (x = y).
    - ∀y(¬P[y]) finds one object of equivalent type x (the previously instantiated a:x) and adds a ¬P[x] node.



First-order modal logic example

Proving Barcan formula:

$$\forall x \, \Box \, A[x] \supset \Box \, \forall x \, A[x]$$

(link)

4. Extending the book

a. Adding Łukasziewicz's fuzzy logic

Priest (2008) does not provide a tableaux method for fuzzy logic. Several tableaux proposals can be found in literature. The software implements the method proposed by Olivetti (2003):

- Proof tree node labels contain the formula, an algebraic expression, and a sign, which can be either + or -.
- The algebraic expression will contain literals and variables (noted as greek letters), whose values are ranged in the [0…1] interval.
- The initial proof tree is now 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.

Fuzzy logic rules - 1/3

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

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

First four rules (top rules) will propagate the algebraic expression. The rules for ⊃ will create a new variable with unique name and new algebraic expressions.

Fuzzy logic rules - 2/3

- Next two rules applies will 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 there is no solution, the tree branch is marked as contradictory.
- The software uses a **linear optimization library (MiniLP)** in order to solve systems of inequalities.
- A system of inequalities maps to a linear optimization problem maximizing f(x) = 0 and constrained by the inequalities of the system.

$\{ \alpha \}, P, +$
(where P is an atom)

$\downarrow$

$\alpha \geq \mu P$

$\{ \alpha \}, P, -$
(where P is an atom)

$\downarrow$

$\alpha < \mu P$

Fuzzy logic rules - 3/3

$$\{\ \alpha\ \}, \neg\neg P, \pm \;\longrightarrow\; \{\ \alpha\ \}, P, \pm$$

$$\{\ \alpha\ \}, \neg(P \wedge Q), \pm \;\longrightarrow\; \{\ \alpha\ \}, \neg P \vee \neg Q, \pm$$

$$\{\ \alpha\ \}, \neg(P \vee Q), \pm \;\longrightarrow\; \{\ \alpha\ \}, \neg P \wedge \neg Q, \pm$$

$$\{\ \alpha\ \}, \neg(P \supset Q), \pm \;\longrightarrow\; \{\ \alpha\ \}, P \supset Q, \mp$$

$\{\ \alpha\ \}, \neg P, +$
(where P is an atom)

$$\downarrow$$

$$\alpha \geq 1 - \mu P$$

$\{\ \alpha\ \}, \neg P, -$
(where P is an atom)

$$\downarrow$$

$$\alpha < 1 - \mu P$$

Fuzzy logic example

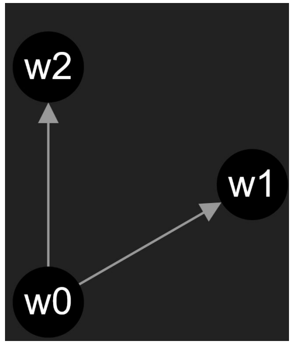
Proving:

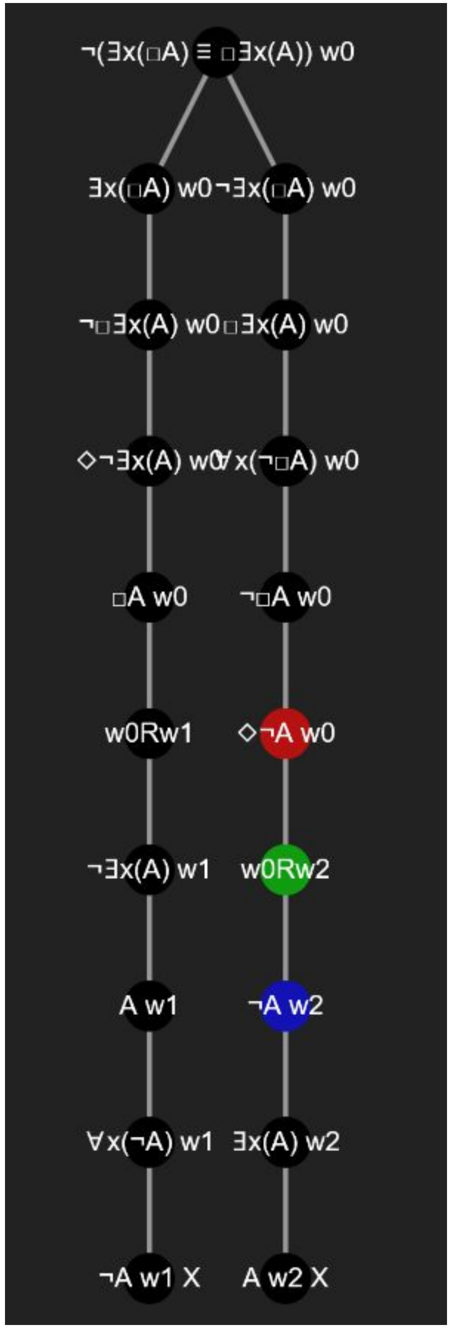
$$(A \supset \neg B) \supset (B \supset \neg A)$$

(link)

b. Didactic tools

1. The parser allows both CommonMath notation (⟶) and Priest's notation (⊃)
2. The catalog contains all exercises from Priest (2008)
3. The user can select the applicable logic
4. The calculator displays the world graph
5. Hovering a tree node with the mouse button highlights the spawner node
6. If the exercise is not proved, there is a search for a finite countermodel to display (see B2)
7. There is an execution log (see B1)





B. THEORETICAL ISSUES

1. Soundness, completeness, complexity

Soundness

It may seem sufficient to inspect the code and see that new nodes are not inserted where the original rules of Priest (2008) would not allow it and that there are no other rules.

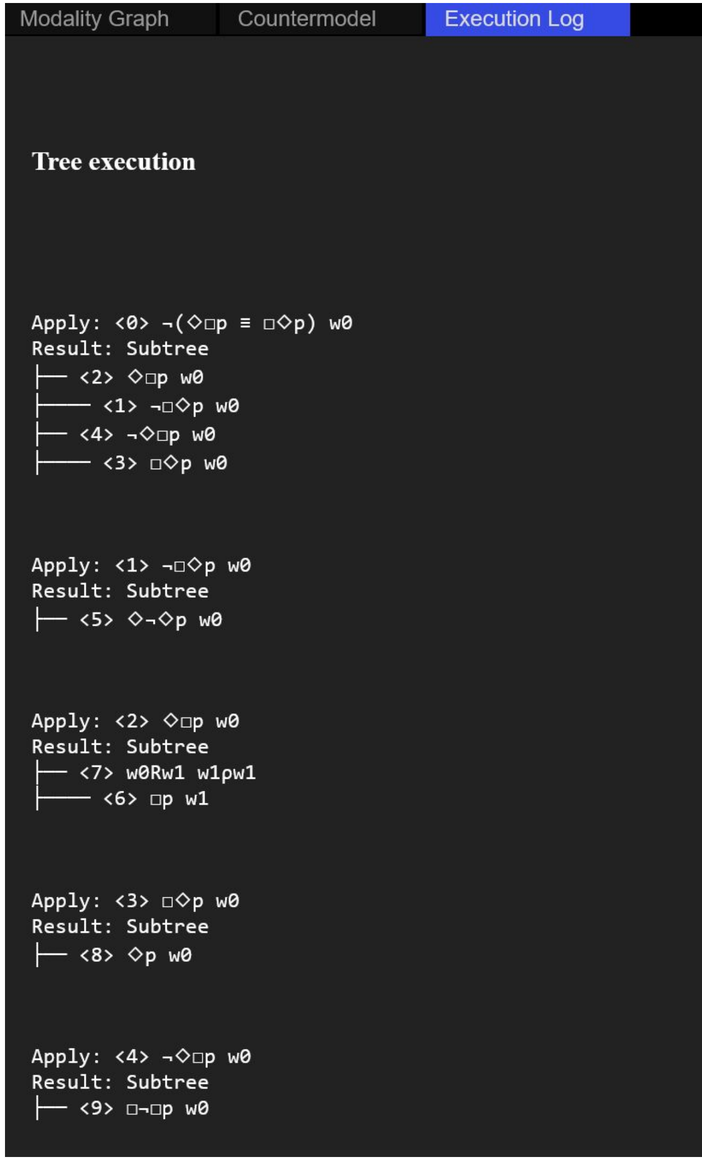
In practice, in such a complex code involving hundreds of rules there are **programming bugs**. The solution is the execution log:

- Throughout code, we inserted calls to populate this log
- Whenever a rule is applied, a node or vertex is added, the log is populated with this info

The argument becomes:

(1) You check the code that whenever a call to populate the log is made, the original rules would allow the move

(2) You check that work on nodes is not done anywhere else



Completeness

It may seem sufficient to inspect the code and for each logic separately, derive a **fully-specified algorithm** in stages, for example on the model of next slide.

Then, map the algorithm to the completeness proofs of Priest (2008), using contraposition: if the procedure does not result in a closed tree, there is a counterexample readable out of it (even if infinite).

In practice, since the code is complex, it is **hard to evaluate "each logic separately"** directly, you can use the same **execution log**:

- It generates a step by step description of rules applications, new nodes, new vertices, new contradictions.
- You can follow its invoking to get the fully-specified algorithm needed.

Fair algorithm example in Fitting and Mendelsohn 2023 p.206-207

Systematic Tableau Construction Algorithm for K There is an infinite list of parameters associated with each prefix, and there are infinitely many prefixes possible. But we have a *countable* alphabet for our logical language, and this implies that it is possible to combine all parameters, no matter what prefix is involved as a subscript, into a single list: $\rho_1, \rho_2, \rho_3, \ldots$. (The subscripts you see here are not the associated prefixes; they just mark the position of the parameter in the list.) Thus, for each prefix, and for each parameter associated with that prefix, that parameter occurs in the list somewhere. (A proof that this can be done involves some simple set theory, and would take us too far afield here. Take our word for it—this can be done.)

What we present is not the only systematic construction possible, but we just need one. It goes in *stages*. Assume we have a closed modal formula Φ for which we are trying to find a **K** tableau proof. For stage 1 we simply put down $1\ \neg\Phi$, getting a one-branch, one-formula tableau.

Having completed stage n, if the tableau construction has not terminated here is what to do for stage $n + 1$. Assume there are b open branches. Number the open branches from left to right, 1, 2, ..., b. Process branch 1, then branch 2, and so on. When branch number b has been processed, this completes stage $n + 1$.

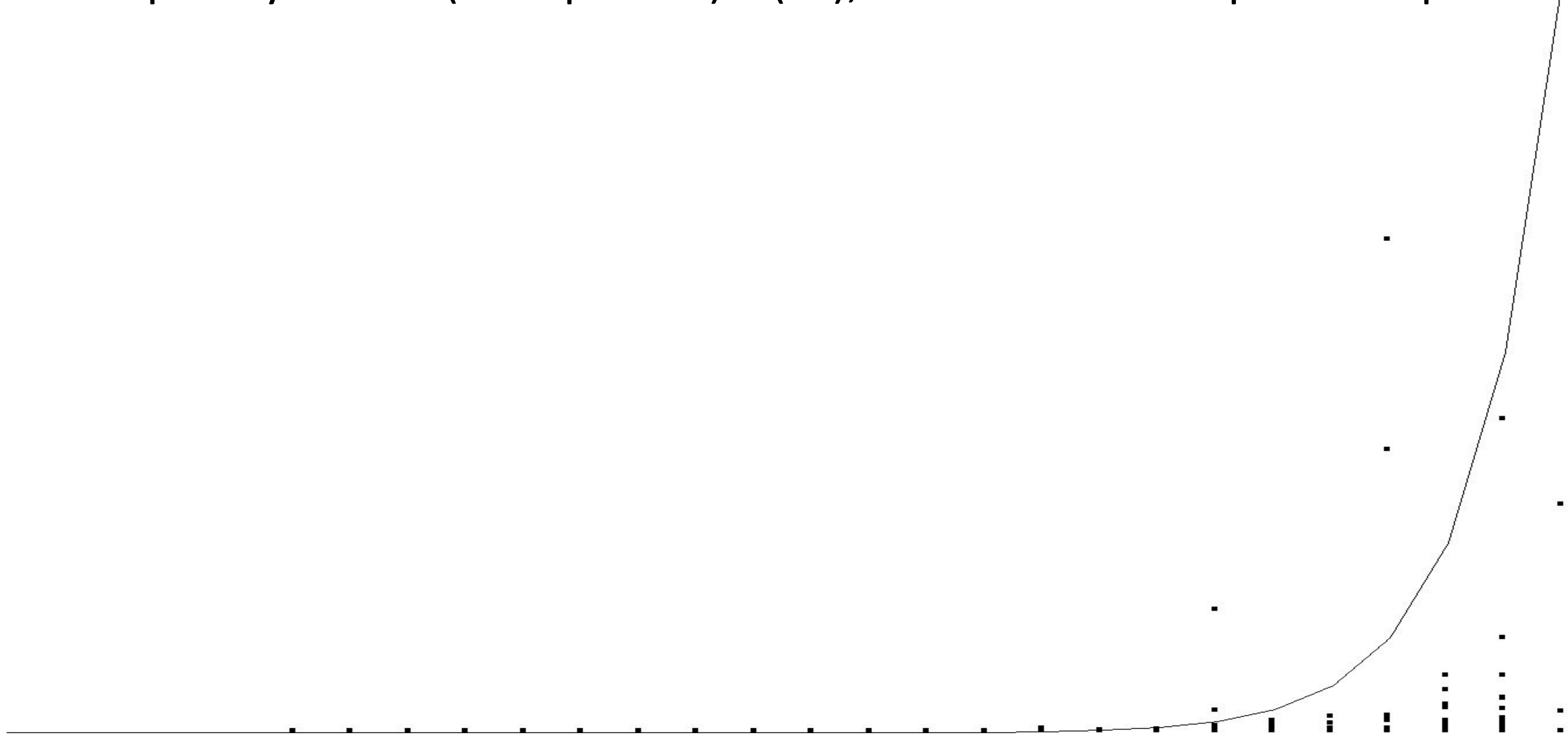
To process an open branch, go through it from bottom to top, processing it prefixed formula by prefixed formula. Each step may add new prefixed formulas to the end of the branch, or even split the end to produce two longer branches. Since processing the original branch proceeds from its bottom to its top, any new formulas that are added to branch ends are not processed at this stage, but are left for the next stage.

To process a prefixed formula occurrence, what to do depends on the form it takes. Let us say we have an occurrence of the prefixed formula Φ. Then for each branch passing through that occurrence of Φ do the following.

1. If Φ is $\sigma\ \neg\neg\Psi$, add $\sigma\ \Psi$ to the branch end, unless it is already on the branch.
2. If Φ is $\sigma\ \Psi \wedge \Omega$, add $\sigma\ \Psi$ to the branch end unless it is already present on the branch, and similarly for $\sigma\ \Omega$. The other conjunctive cases are treated in the same way.
3. If Φ is $\sigma\ \Psi \vee \Omega$, and if neither $\sigma\ \Psi$ nor $\sigma\ \Omega$ occurs on the branch, split the end of the branch and add $\sigma\ \Psi$ to one fork and $\sigma\ \Omega$ to the other. The other disjunctive cases are treated in the same way.
4. If Φ is $\sigma\ \Diamond\Psi$, and if $\sigma.k\ \Psi$ does not occur on the branch for any k, choose the *smallest* integer k such that the prefix $\sigma.k$ does not appear on the branch at all, and add $\sigma.k\ \Psi$ to the branch end. Similarly for the negated necessity case.
5. If Φ is $\sigma\ \Box\Psi$, add to the end of the branch every prefixed formula of the form $\sigma.k\ \Psi$ where this prefixed formula does not already occur on the branch, but $\sigma.k$ does occur as a prefix somewhere on the branch. Similarly for the negated possibility case.
6. If Φ is $\sigma\ (\exists x)\Psi(x)$, and if $\sigma\ \Psi(p_\sigma)$ does not occur on the branch for any parameter with σ as a subscript, then choose the *first* parameter ρ_i in the list $\rho_1, \rho_2, \ldots$ having σ as a subscript, and add $\sigma\ \Psi(\rho_i)$ to the branch end. Similarly for the negated universal case.
7. If Φ is $\sigma\ (\forall x)\Psi(x)$, add to the end of the branch the prefixed formula $\sigma\ \Psi(\rho_i)$ where: ρ_i is the first parameter in the list $\rho_1, \rho_2, \ldots$ having σ as a subscript, but where $\sigma\ \Psi(\rho_i)$ does not already occur on the branch. Similarly for the negated existential case.

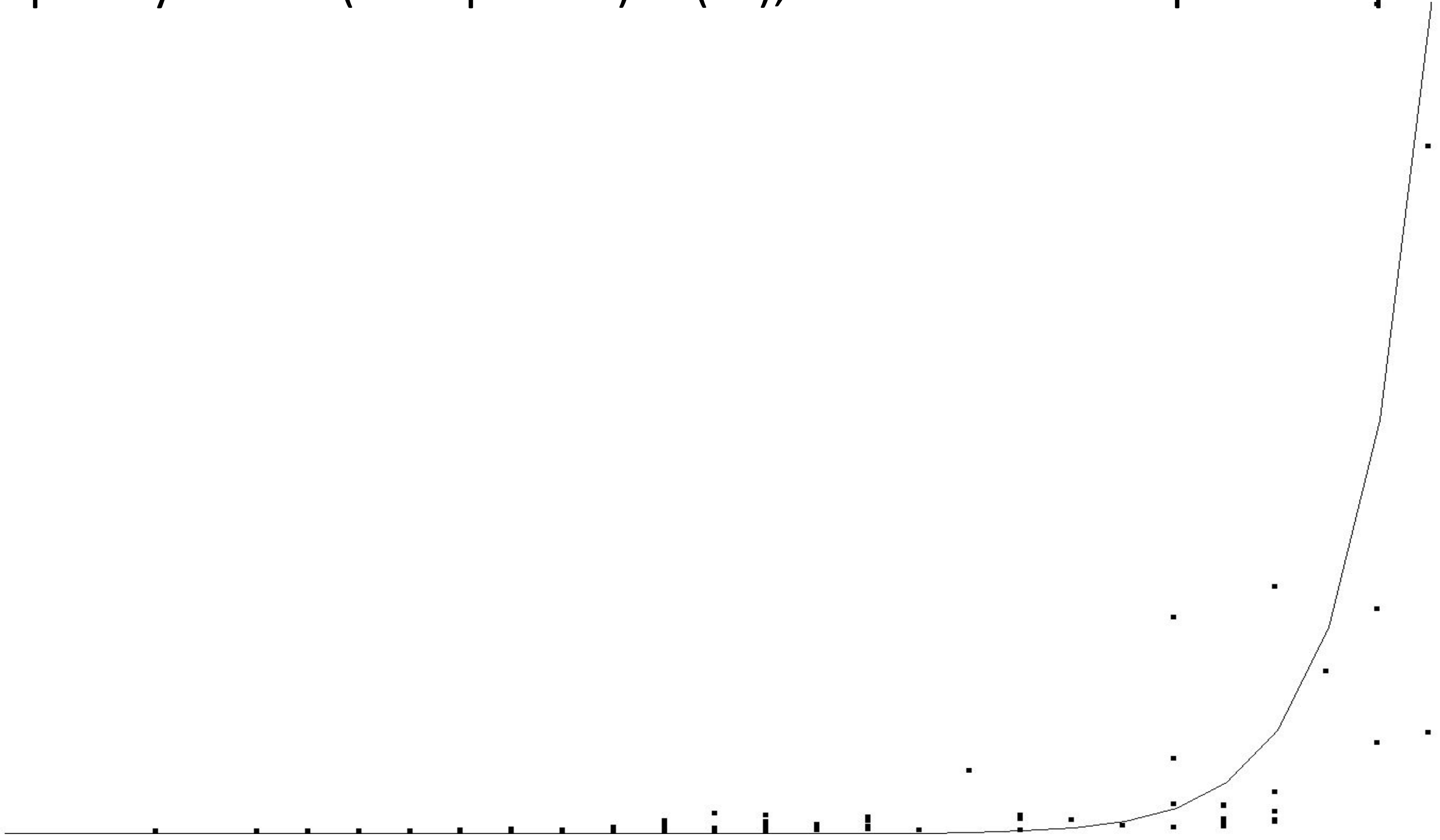
Complexity: measuring RAM?

- We used a library that replaces Rust's default memory allocator with a custom one, then measured the number of bytes allocated while the software ran, w/o counting disallocations.
- We generated a set of 887 random classical propositional logic (CPL) problems.
- Each point represents a problem. X axis: number of operators. Y axis: number of allocated bytes.
- Conclusion: complexity seems (as expected) $O(2^n)$, Note that the slope is not perfectly smooth.



Complexity: measuring CPU cycles?

- We rely on a library that uses Linux's debugging facilities in order to measure the total number of instructions executed by the CPU while the software ran. We took a set of 103 CPL problems.
- Each point represents a problem. X axis: number of operators. Y axis: number of ran instructions.
- Conclusion: complexity seems (as expected) $O(2^n)$, Note that the slope is not perfectly smooth.



Complexity via the execution log (in progress)

```
Apply: <8> ¬(P[b:x] → ∀y(P[y]))        New nodes: {}        New contradictions:     0B (0.0000MB)
Result: Subtree                         New vertices:        {(10, 6)}
├── <10> P[b:x]                         {}
├──── <9> ¬∀y(P[y])
```

The execution log contains:

- Number of **rule applications**
- New **subtrees (nodes)**
- New **worlds** in the graph
- New **vertices** for the world graph
- New **contradictions**
- (in the future): Number of **examined tree nodes**

With all these, the execution log can describe a complete and sound algorithm.

Then, the number of items in the log would measure complexity directly, vs. the number of operators in the premises and conclusion.

For example, for two operators, we get 12 rule applications + 10 new nodes + 145 examined nodes. For three operators, we get more. We think this is the correct approach

2. Finite counterexamples and SAT

Trial-and-error counterexamples

Infinite tree in Kη for ⊬ □p. Corresponding countermodel. Simpler countermodel findable by trial and error. All from Priest (2008)

$$
\begin{array}{c}
\neg\Box p, 0 \\
\Diamond\neg p, 0 \\
0r1 \\
\neg p, 1 \\
1r2 \\
2r3 \\
\vdots
\end{array}
$$

Except for CPL and K modal logic, the tableaux method will result in an infinite tree if there is no proof.

But there are finite counter models which can be found by trial and error. Imagine a student at an exam.

$$
\begin{array}{ccccccc}
 & & \neg p & & & & \\
w_0 & \rightarrow & w_1 & \rightarrow & w_2 & \rightarrow & \cdots
\end{array}
$$

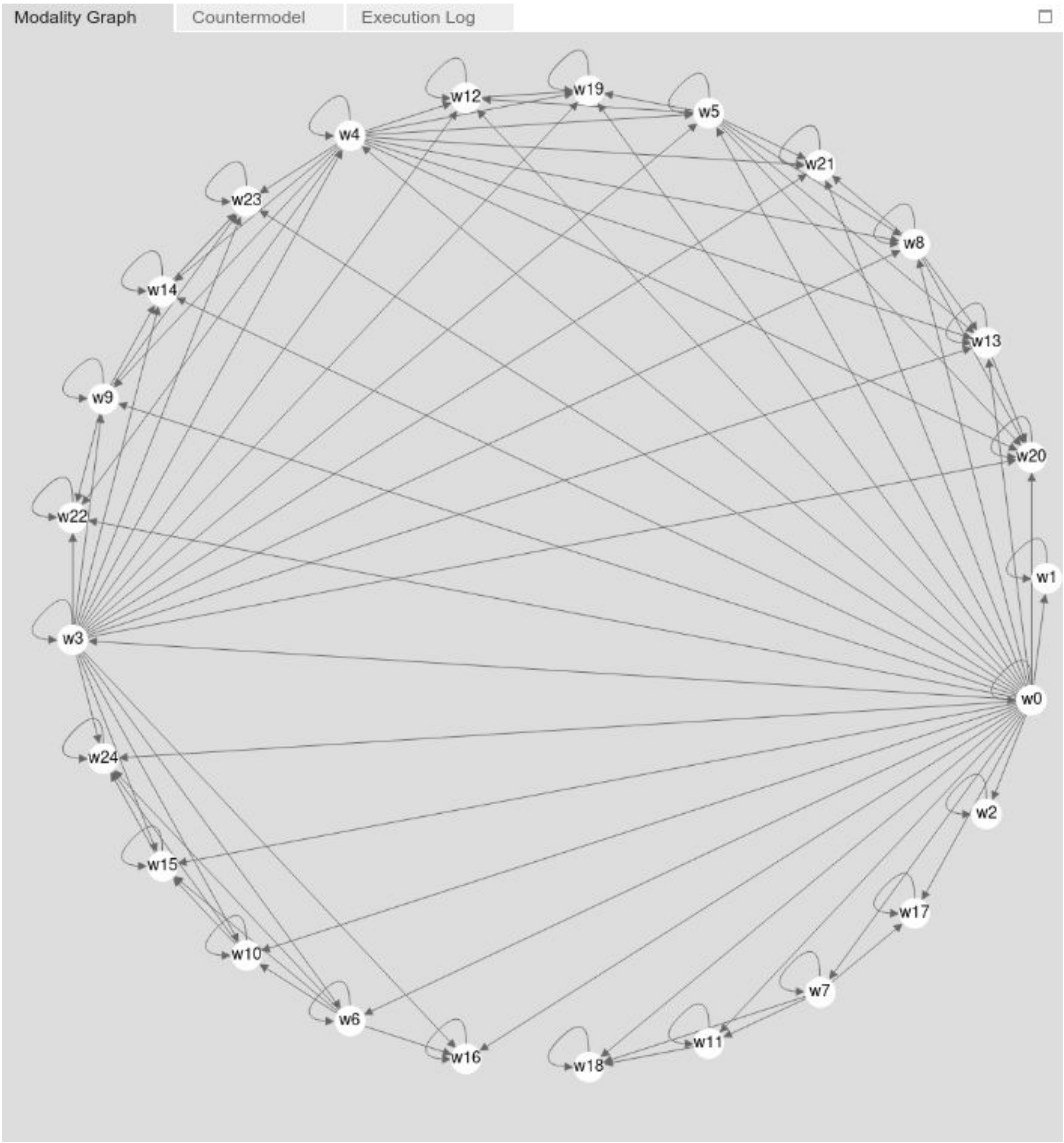
What kind of algorithm corresponds to this trial-and-error?

- **Brute-force?** Simply taking all possibilities and going through them?
- **SAT**? Trying to find a valuation that satisfies the translation of the exercise (with negated conclusion) to propositional logic?

$$
\begin{array}{c}
\curvearrowright \\
w_0 \\
\neg p
\end{array}
$$

The algorithm - 1/2

- Available only on K, T, B, S4 and S5 propositional normal modal logics and their first-order constant domain counterparts.
- The model finding problem is reduced to a boolean satisfiability problem, which is solved using a third-party SAT solver.
- A SAT solver is a software which efficiently finds binary atomic truth values which satisfies (makes true) a set of classical propositional logic formulas.



The algorithm - 2/2

Given a problem with premises $P_1,...,P_n$ and conclusion C, let L be the list of $P_1,...,P_n$ and ¬C

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

- Generate all possible graphs with N nodes
- Filter out invalid graphs given each logic's requirements

For each graph G:

- If the logic is quantified:
  - For each natural number $M \in [1...10]$, generate a domain D of M objects.
  - For each D, eliminate quantifiers: transform all predicate logic formulas in L into propositional logic formulas (see next slide). The procedure is domain-dependent (D)
- Eliminate modalities: transform all propositional modal formulas in L into classical propositional logic formulas. This procedure is graph-dependent (G).
- Bring remaining CPL formulas to conjunctive normal form (CNF) and run a *third-party SAT solver*.
- If solution is found, it will consist of atomic values of all the atoms of all formulas in L. Attach atomic values into G's nodes. Output G as **found finite countermodel.**

How to transform a quantified formula into a propositional one?

Given a domain $\{a_1, a_2, ..., a_m\}$ and a formula, recursively apply the following rules:

- R1: Turn $\exists x(P[x])$ into $P[a_1] \lor P[a_2] \lor \ldots \lor P[a_m]$
- R2: Turn $\forall x(P[x])$ into $P[a_1] \land P[a_2] \land \ldots \land P[a_m]$
- 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 $\exists x \forall y\, P[x,y] \supset x=y$ will be transformed as follows:

- Apply R1: $(\forall y\, P[a,y] \supset a=y) \lor (\forall y\, P[b,y] \supset b=y)$
- Apply R2: $((P[a,a] \supset a=a) \land (P[a,b] \supset a=b)) \lor ((P[b,a] \supset b=a) \land (P[b,b] \supset b=b))$
- Apply R3: $((Paa \supset a=a) \land (P\ ab \supset a=b)) \lor ((Pba \supset b=a) \land (Pbb \supset b=b))$
- Apply R4: $((Paa \supset T) \land (Pab \supset \bot)) \lor ((Pba \supset \bot) \land (Pbb \supset T))$

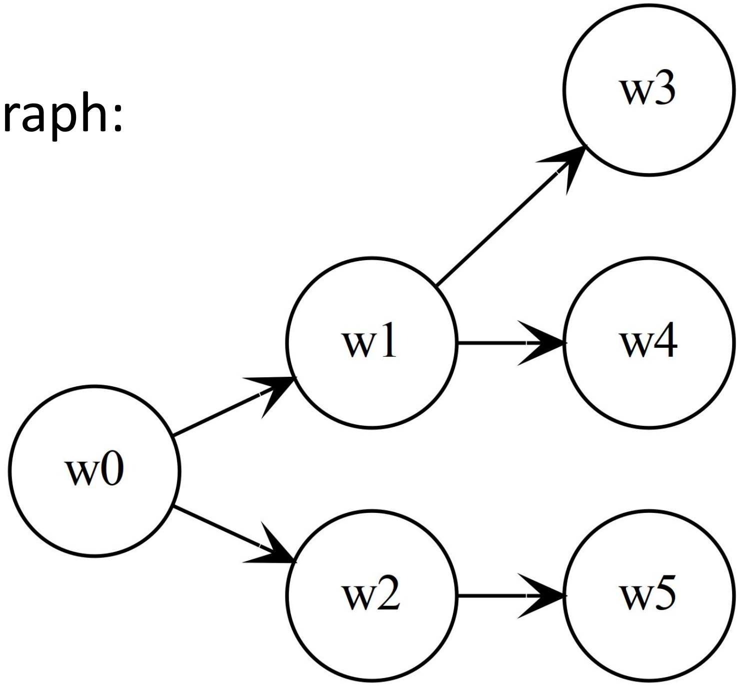
How to transform a modal formula into a classical logic formula?

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 $(P \land Q), w_i$ into $(P, w_i) \land (Q, w_i)$
- 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 $P,w_{j1} \lor P,w_{j2} \lor \ldots \lor P,w_{jn}$ if G has $w_i \rightarrow w_{jk}$ vertices
- R9: Turn $\Box P, w_i$ into $P,w_{j1} \land P,w_{j2} \land \ldots \land P,w_{jn}$ if G has $w_i \rightarrow w_{jk}$ vertices
- R10: Turn P, w_i into Pi if P is an atom.

For instance, $\Diamond(P \land \Box Q)$ will be transformed as follows, given the graph:

- Assume w0 as initial possible world: $\Diamond(P \land \Box Q), w_0$
- Apply R8: $(P \land \Box Q), w_1 \lor (P \land \Box Q), w_2$
- Apply R5: $((P, w_1) \land (\Box Q, w_1)) \lor ((P, w_2) \land (\Box Q, w_2))$
- Apply R10: $(P1 \land (\Box Q, w_1)) \lor (P2 \land (\Box Q, w_2))$
- Apply R9: $(P1 \land ((Q, w_3) \land (Q, w_4))) \lor (P2 \land (Q, w_5))$
- Apply R10: $(P1 \land (Q3 \land Q4)) \lor (P2 \land Q5))$

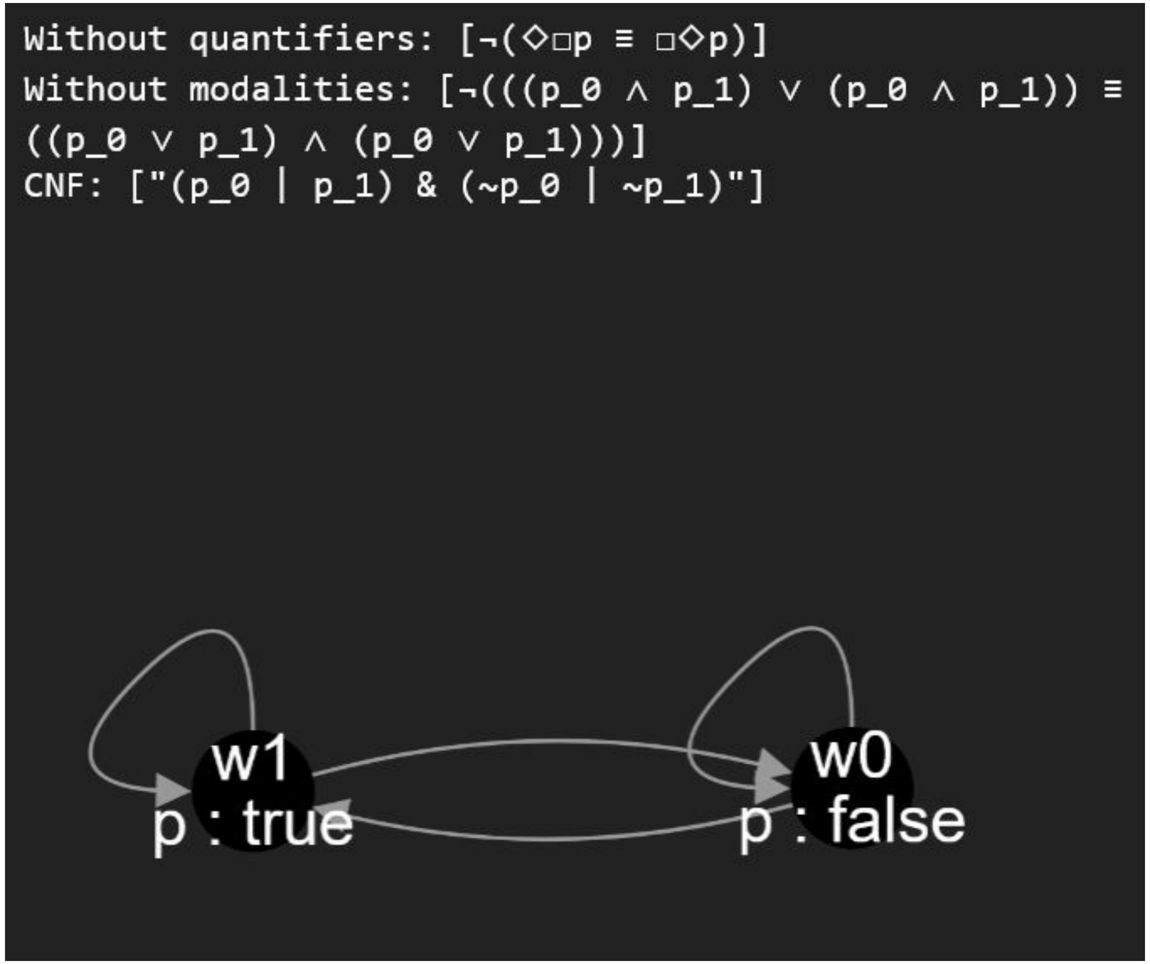


Example

A countermodel in S4 for:
$\Diamond\Box P \equiv \Box\Diamond P$

(link)

SAT solving seems fast in the calculator, but **looping through all possible world graphs is exponential**. Click Shuffle button to get slower setups.



3. Generating new logics

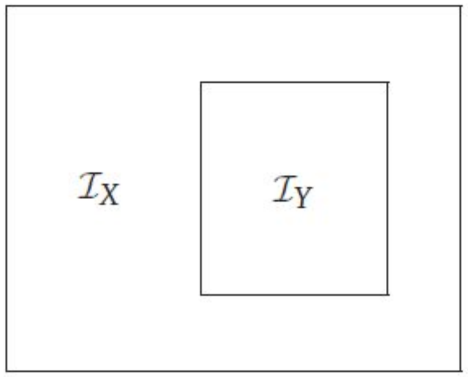
What is a logic - 1/2

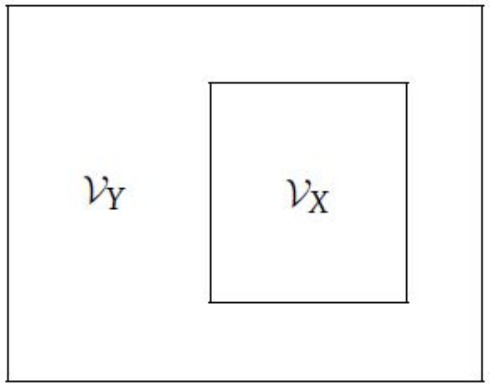
Standard definition: A logic is a formal language with a proof system (a calculus) and a semantics.

Priest (2008) favors semantical priority:

- “Most contemporary logicians would take the semantic notion of validity to be more fundamental than the proof-theoretic one, though the matter is certainly debatable”

- E.g. modal logic Kρ (T) is introduced as “We denote the logic defined in terms of truth preservation over all worlds of all ρ-interpretation, Kρ”





“The logic determined by the class of interpretations Iy is an extension of that determined by the class Ix. [...] Vx and Vy are the sets of the inferences that are valid in the two logics.[..] ***fewer interpretations, more inferences****” Priest (2008)*

What is a logic - 2/2

In the calculator, **a logic** is rather a calculus, a **combination** of:

- An unique name (eg: S5ModalLogic, FirstOrderS5ModalLogic, …)
- The number of available truth values (2, 3, 4 or ∞)
- A syntax: a set of symbols available in that logic (eg: $\neg, \wedge, \vee, \ldots$)
- A set of rules, telling the computer how tree nodes should be generated and how contradictions should be detected.
- Whether it is propositional or quantified.
- Whether it is modal or not.

Modularity - Language and operators

Module	Description	Source files
Classical propositional	classical operators: ¬, ∧, ∨, →, ↔	propositional_logic.rs
Normal Modal propositional	CPL plus modal operators (□, ◇) with additional axioms (K, T, B, S4, S5) as constraints on operators	normal_modal_logic.rs, common_modal_logic.rs
Non-Normal Modal prop.	CPL plus modal operators with rules (e.g. S0.5, N, S2, S3, S3.5)	non_normal_modal_logic.rs
Temporal Modal prop.	Modal plus temporal operators ([P], <P>, [F], <F>)	temporal_modal_logic.rs
Conditional Modal prop.	Conditional logic operators and rules	conditional_modal_logic.rs
Intuitionistic prop.	Intuitionistic logic constraints (tree rules)	intuitionistic_logic.rs
First Degree Entailment / Finite many-valued (prop.)	Kleene, Priest's LP, RMingle3, Ł3, logics with gaps and gluts, paraconsistent logics or logics with extra truth values, with various specific rules on contradiction detection	first_degree_entailment.rs, kleene_modal_logic.rs, lukasiewicz_modal_logic.rs, priest_logic_of_paradox.rs, rmingle3_modal_logic.rs, logic_with_gaps_and_gluts.rs, logic_of_constructible_negation.rs
Fuzzy / Many-Valued (prop.)	Logic with infinite truth values, real numbers in 0..1 interval. Algebraic expressions and linear programming (see above)	fuzzy_logic.rs
First order	Predicate support, ∀, ∃, 𝔈 operators support	first_order_logic.rs

Modularity - Quantification, domain, identity

Logic	Description	Comments
Propositional	Propositional logic and its modal variants	
First order	FirstOrderLogic module	
Constant domain	FirstOrderLogicDomainType::ConstantDomain (a single domain in all worlds)	
Variable domain - increasing	FirstOrderLogicDomainType::VariableDomain (with newly accessible worlds can have new objects)	
Variable domain - decreasing	As above (with newly accessible worlds can have fewer objects)	
Necessary identity	NecessaryIdentity in FirstOrderLogicIdentityType (in all worlds)	
Contingent identity	ContingentIdentity (varies with world)	

Modularity - how LogicFactory instantiates a Logic

LogicFactory instantiates logic objects from a **logic ID** (an unique logic name) specifying the **module** in *Language and operators* slide

First-order logic objects are created by picking the **modules** out of those in *Quantification, domain, identity* slide

Note that some of the combinations would not work properly out of the box:

- Temporal past/future operators with varying domain
- Fuzzy logic with any first order extension

```
impl LogicFactory
{
    pub fn get_logic_by_name(name : &String) -> Result<Rc<dyn Logic>>
    {
        return Self::get_logic_theories().into_iter()
            .find(|logic :&Rc<dyn Logic, Global> | logic.get_name().to_string().as_str() == name.as_str())
            .context(format!("Invalid logic with name {}!", name));
    }

    pub fn get_logic_theories() -> Vec<Rc<dyn Logic>>
    {
        let base_logics : Vec<Rc<dyn Logic>> = vec!
        [
            Rc::new( value: PropositionalLogic {}),
            Rc::new(NormalModalLogic::K()),
            Rc::new(NormalModalLogic::S4()),
            //........................
            Rc::new( value: LukasiewiczFuzzyLogic {}),
        ];

        let domain_types :[FirstOrderLogicDomainType; 3] =
        [
            ConstantDomain,
            VariableDomain(VariableDomainFlags { has_domain_increasing_constraint:false }),
            VariableDomain(VariableDomainFlags { has_domain_increasing_constraint:true }),
        ];

        let mut output_logics :Vec<Rc<...>, Global> = base_logics.clone();
        for domain_type :FirstOrderLogicDomainType in domain_types
        {
            for identity_type :FirstOrderLogicIdentityType in [NecessaryIdentity, ContingentIdentity]
            {
                for base_logic :&Rc<dyn Logic, Global> in &base_logics
                {
                    let first_order_logic :FirstOrderLogic = FirstOrderLogic { domain_type, identity_type, base_logic };
                    output_logics.push(Rc::new(first_order_logic));
                }
            }
        }
        return output_logics;
    }
```

Possible future work - contributions welcome

1. Add node examinations to the execution log for soundness, completeness and complexity determination

2. Cover missing logics from Priest (2008) - relevant logics

3. Extend *SAT reduction* for finite countermodels to more logics

4. Extend the limits configuration (now there are three limits: 25 worlds, 250 nodes in general or 1000 at intuitionistic logic) . Maybe make them custom or RAM/ time dependent

5. Investigate final modularity concerns: which combinations define complete logics and which do not.

Thank you!

Special thanks

- **The students at the 'Logic and software' course, 2024**, Faculty of Philosophy, University of Bucharest.
- **Graham Priest**

Bibliography

- Fitting, Mendelsohn 2023 - *First-Order Modal Logic*
- Girle 2010 - *Modal Logics and Philosophy*
- Johnson 2015 - *Tree Trimming: Four Non-Branching Rules for Priest's Introduction to Non-Classical Logic*
- Olivetti 2003 - *Tableaux for Łukasiewicz Infinite-valued Logic*
- Priest 2008 – *An Introduction to Non Classical Logic*
- Roy 2006 - *Natural Derivations for Priest, An Introduction to Non-Classical Logic*
- Roy 2017 - *More Natural Derivations for Priest*
- Smullyan 1968 – *First-Order Logic*