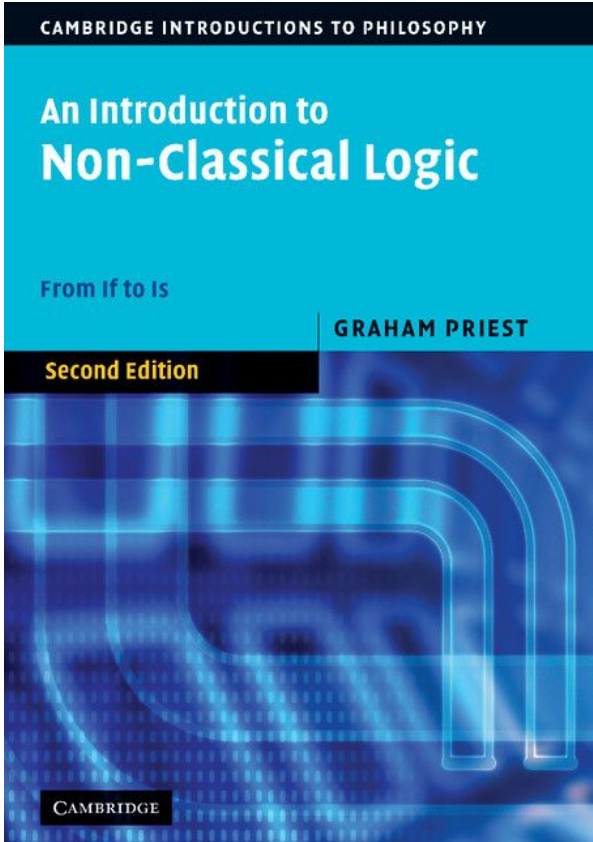
Developing an Automated Proof Calculator for Modal Logic

Based on Graham Priest - "An Introduction to Non-Classical Logic. From if to is (second edition)"

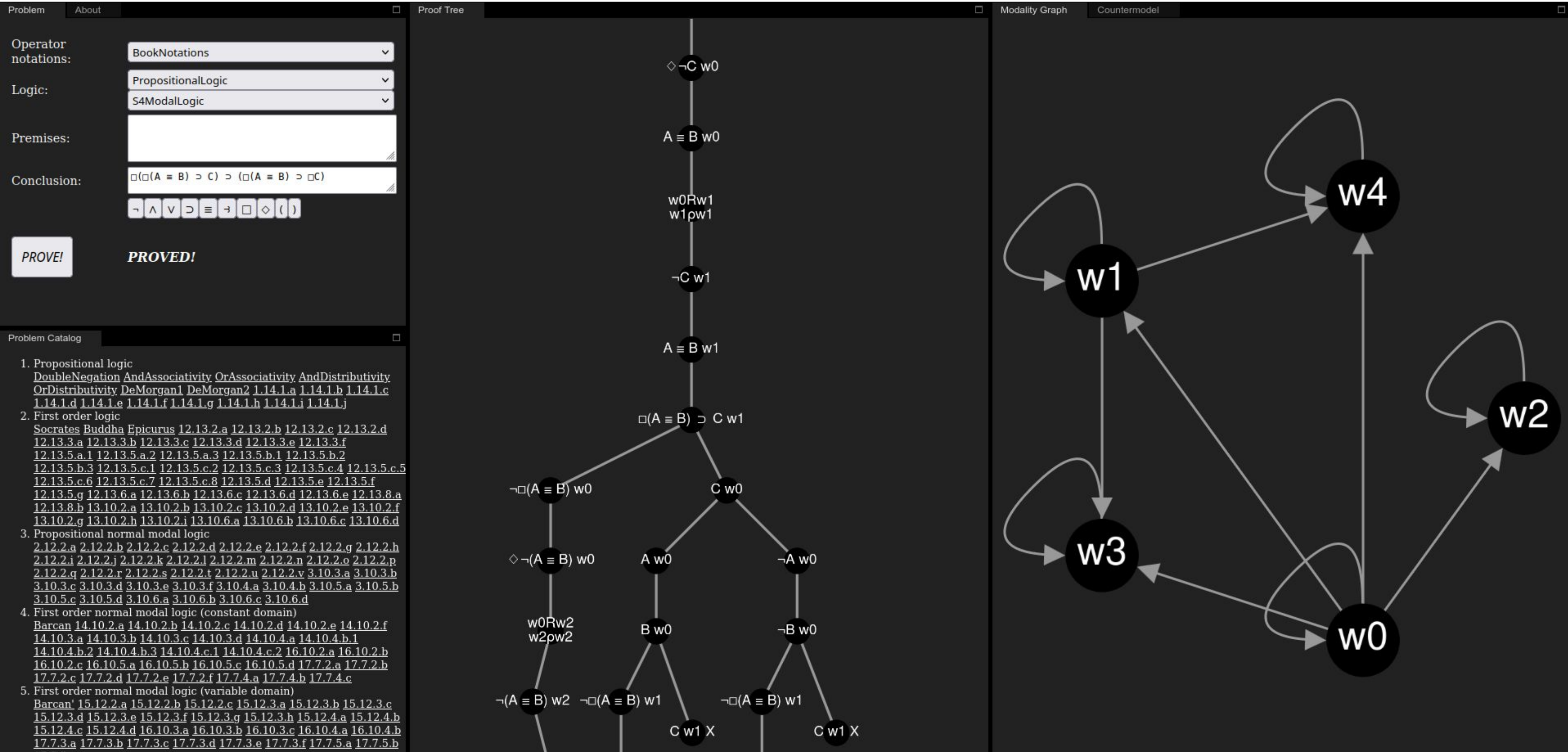
Andrei Dobrescu (andrei.dobrescu@neurony.ro)
and Marian Călborean (mc@filos.ro)
(work in progress)

Tableaux proof system

- A strictly syntactic proof system, just as natural deduction and axiomatic systems.
- In “An Introduction to Non-Classical Logic. From if to is” (2008, right), Graham Priest theorizes tableaux systems for various non-classical logics: modal logics, intuitionistic logic, many-valued logics and their first-order counterparts.
- The software is free and open-source.
  - Available online at: https://andob.io/incl/
  - Written in Rust, source code here.



The calculator software



Implementation coverage

(Part I of the book: Propositional logics)

	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.

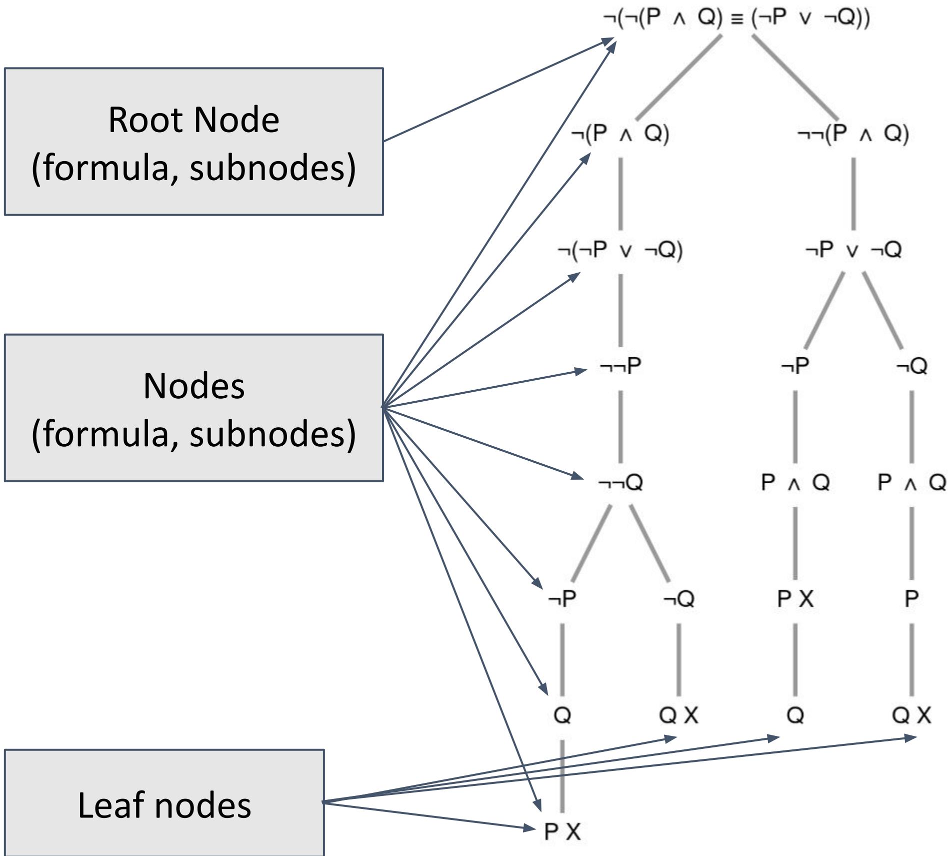
Implementation coverage

(Part II of the book: First-Order counterparts)

	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.

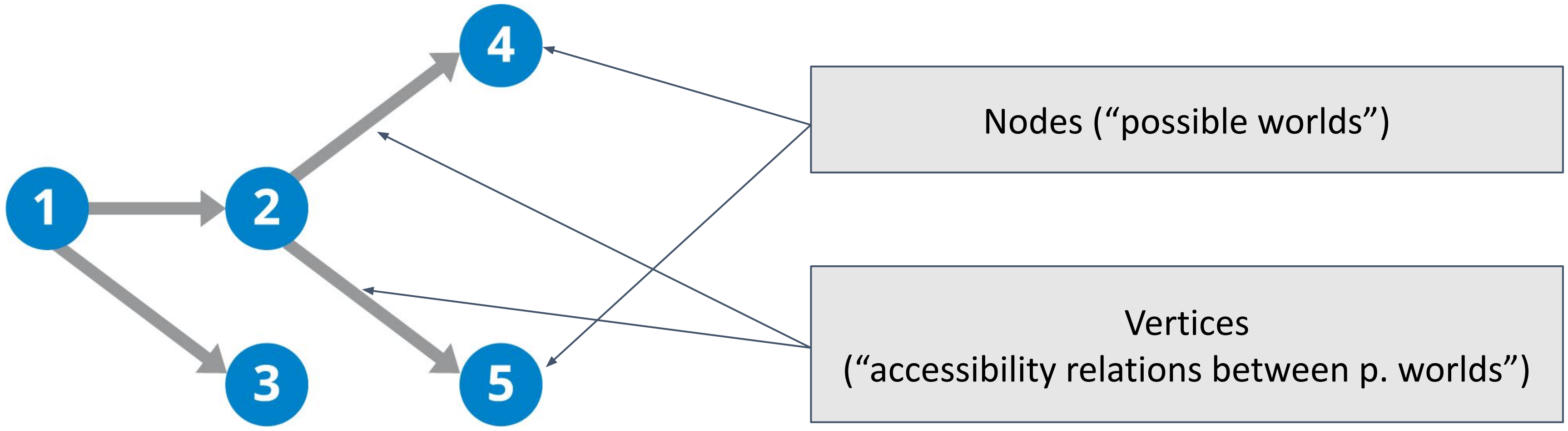
What is a tree?

- A data structure consisting of **nodes** disposed in an arborescent manner.
- Each node has a formula and many **subnodes**.
- Except for the **root node**, each node has a parent node.
- The nodes which have zero subnodes are called **leaf nodes**.
- A path containing all adjacent nodes from the root node to a leaf node is called a **tree branch**.

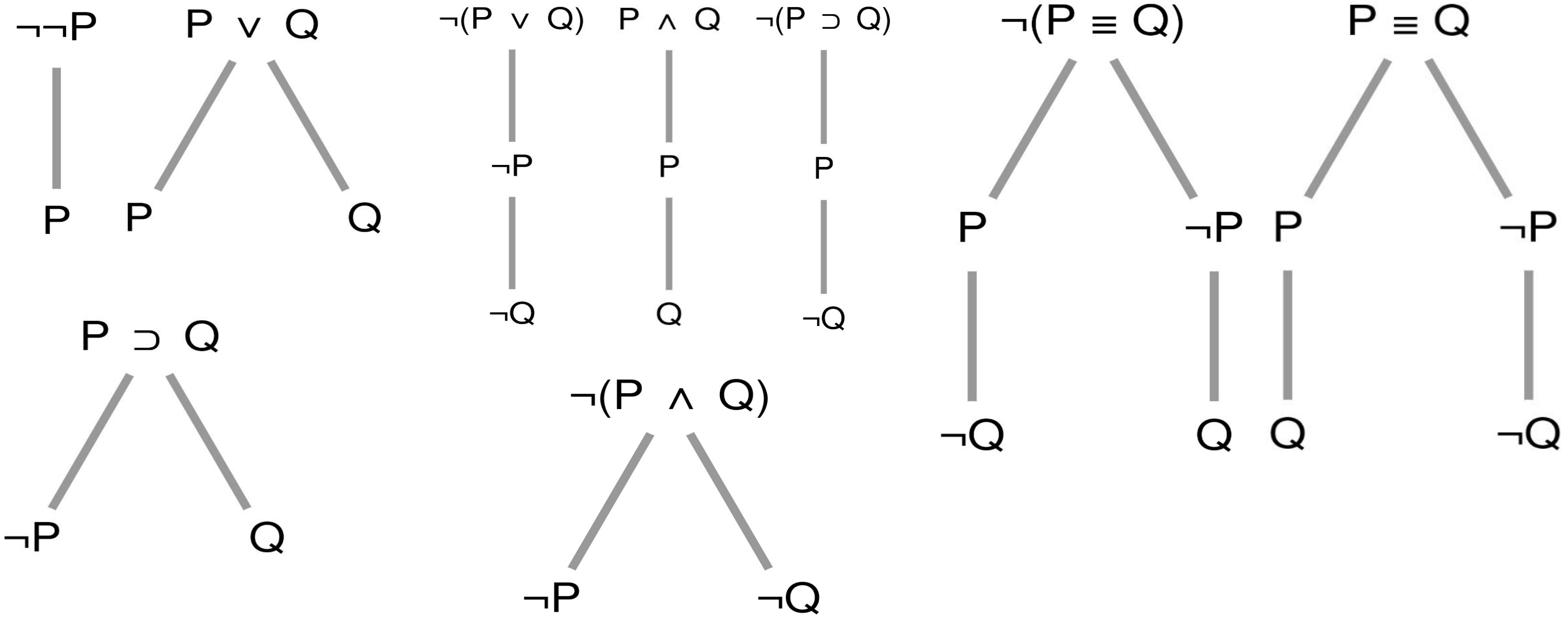


What is a (directed) graph?

- A directed graph is a data structure consisting of a set of **nodes** and a set of **vertices** („arrows” between one node to another).
- The computer science concept of directed graph can be directly mapped to the philosophical concept of a **Kripke model**.



Classical logic rules



Classical logic: example

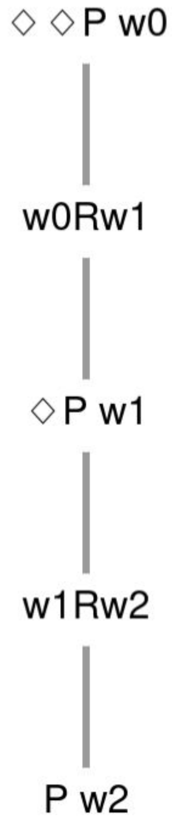
- Proving DeMorgan rule:
  - $\neg(P \lor Q) \equiv \neg P \land \neg Q$
    - (link)

Modal logic rules

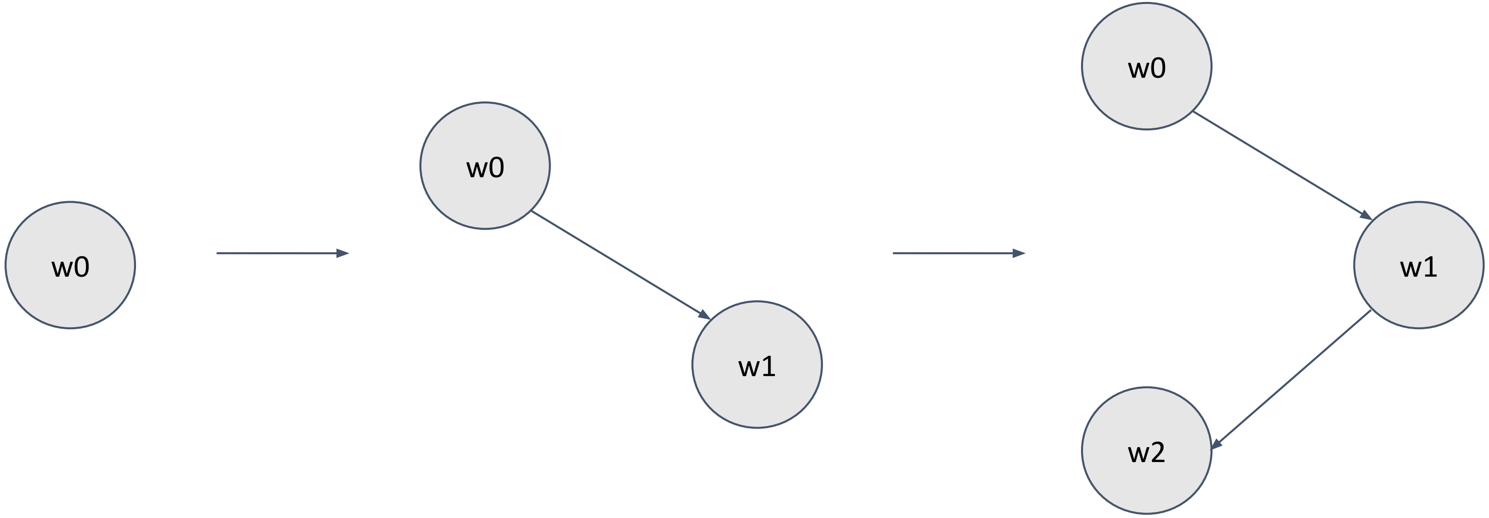
$$\begin{array}{c} \Box A, i \\ irj \\ \downarrow \\ A, j \end{array} \qquad \begin{array}{c} \Diamond A, i \\ \downarrow \\ irj \\ A, j \end{array}$$

$$\begin{array}{c} \neg \Diamond P \\ | \\ \Box \neg P \end{array} \qquad \begin{array}{c} \neg \Box P \\ | \\ \Diamond \neg P \end{array} \qquad \begin{array}{c} P ⥽ Q \\ | \\ \Box (P \supset Q) \end{array} \qquad \begin{array}{c} \neg (P ⥽ Q) \\ | \\ \neg \Box (P \supset Q) \end{array}$$

◇Possibility rule

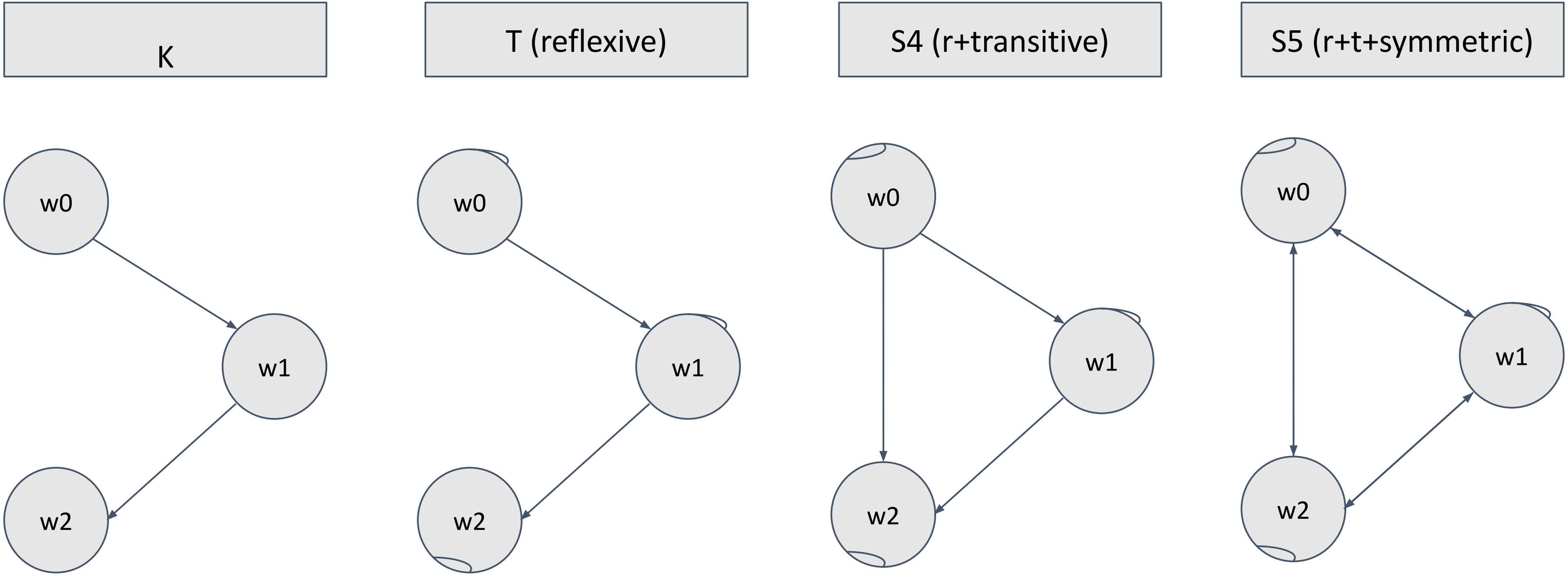


- ◇ rule will spawn a new possible world from the current world.
- ◇ rule will add new nodes and new vertices to the graph.
- ◇ rule will add w_iRw_j nodes to the tree for each new vertex.

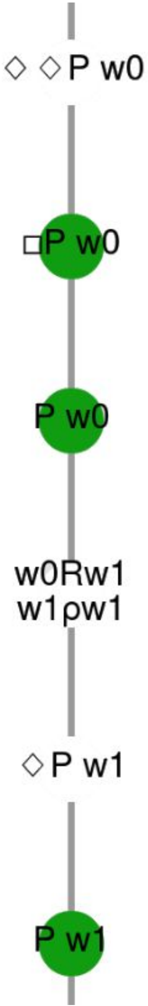


◇Possibility rule

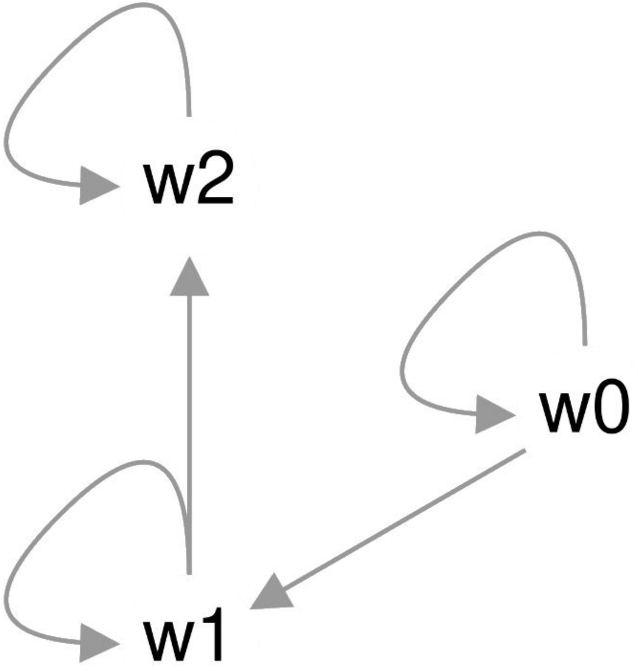
- ◇ rule will also add additional graph vertices, as follows:



□ Necessity rule



- □ rule will iterate all graph vertices that start from the current world:
  - Example: □P w0 is applied here.
  - There are two 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:**
- **$\Box A \equiv \neg \Diamond \neg A$**
- (link)

First-order modal logic example

- **Proving Barcan formula:**
- **$\forall x \, \Box \, A[x] \equiv \Box \, \forall x \, A[x]$**
- (link)

Philosophical notes

What is a logic?

- In code I define a logic as an object with the following properties:
  - An unique name (eg: PropositionalLogic, S5ModalLogic, FirstOrder+S5ModalLogic, Lukasiewicz+KModalLogic,…)
  - The number of available truth values (2, 3, 4 or ∞)
  - A syntax: a set of symbols available in that logic (eg: $\neg, \wedge, \vee, \rightarrow$,…)
  - A set of rules, telling the computer how tree nodes should be generated and how contradictions should be detected.
- This is, of course, not a definition of a logic as a mathematical theory, just an engineered model of the theory.

Why did I use a graph?

- The algorithm, as theorized in the book, does not use a graph data structure, only a tree data structure. In the tree, possible worlds and the relations between them are described via w_iRw_j nodes (meaning: there are two worlds w_i, w_j and a $w_i \to w_j$ relation between them).
- Since a Kripke model can be built from these nodes, and there is a direct equivalence between a Kripke model and a graph data structure, I chose, based on my programming experience, to use a graph in tandem with the proof tree.
- There are technical reasons for my choice: by using two separate data structures, the code becomes easier to follow, maintain and extend.
- In programming, simplicity over complexity is preferred, even if by simplifying we introduce new constructs.

Thank you!

- **Special thanks**
- Graham Priest, Marian Călborean, Alexandru Dragomir

- **Bibliography**

- Graham Priest - “An Introduction to Non-Classical Logic. From if to is (second edition)” (2008)
- Graham Priest – “Logic: A Very Short Introduction” (2000)
- Melvin Fitting, Richard Mendelsohn – “First-Order Modal Logic” (1998)
- James Storer – “An Introduction to Data Structures and Algorithms” (2002)
- Steve Klabnik, Carol Nichols – “The Rust Programming Language” (2023)

Questions?

Annex: Technical details

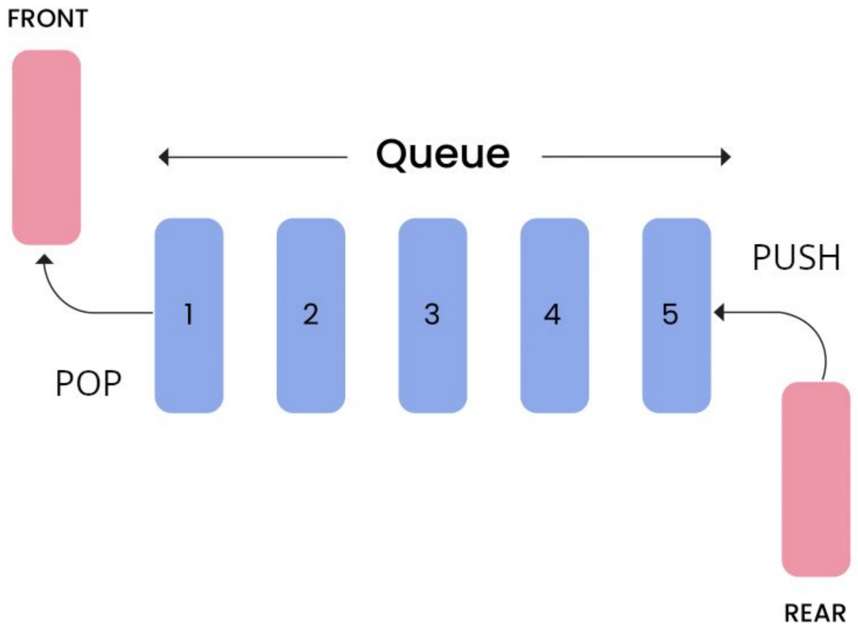
How it works?

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.
- The proof procedure starts by building the *ProofTree* with vertically disposed sequential nodes. For each premise, there will be a node for that premise. There will be another node with the non-conclusion.
- A *ProofTree* is formed by following specific decomposition rules.
- Problem is proved iff the *ProofTree* has contradiction on all brances (thus the initial assumption about the conclusion is false).
- If the *Problem* is disproved, the program provides a counterexample.
- The software will timeout (neither prove nor disprove) above a limit (if the *ProofTree* reaches 250 nodes).

What is a queue?

- A FIFO ("first in first out") data structure.
- On the rear of the queue, the program pushes formulas (starting with the premises and the non-conclusion).
- From the front of the queue, the program will sequentially pop and process formulas.
- The program stops when the queue becomes empty (when there are no formulas left to process).

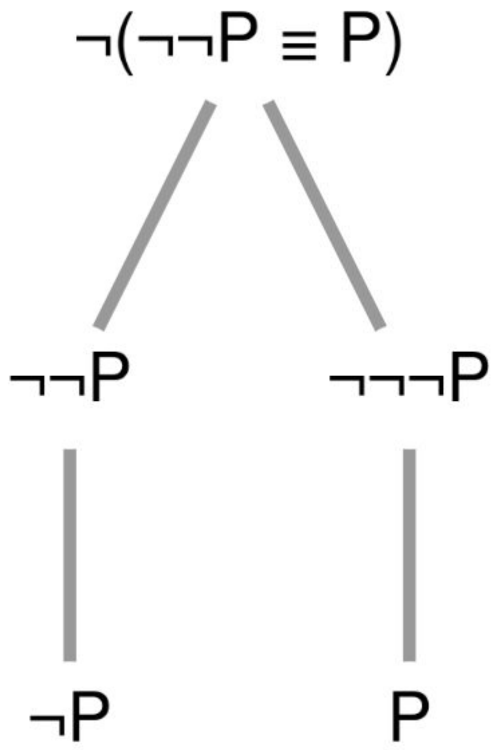


The algorithm in a nutshell - 1/4

- 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.

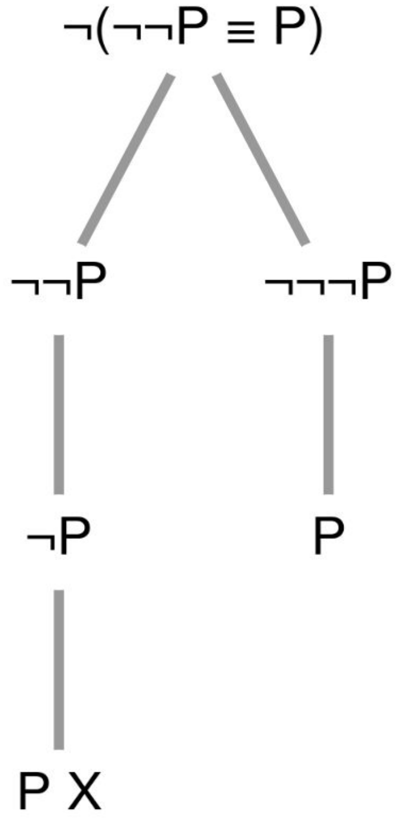
The algorithm in a nutshell - 2/4

- 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: there are no contradictions
  - All resulting formulas are pushed to the queue
  - The queue becomes { ¬¬P, ¬P, ¬¬¬P, P }



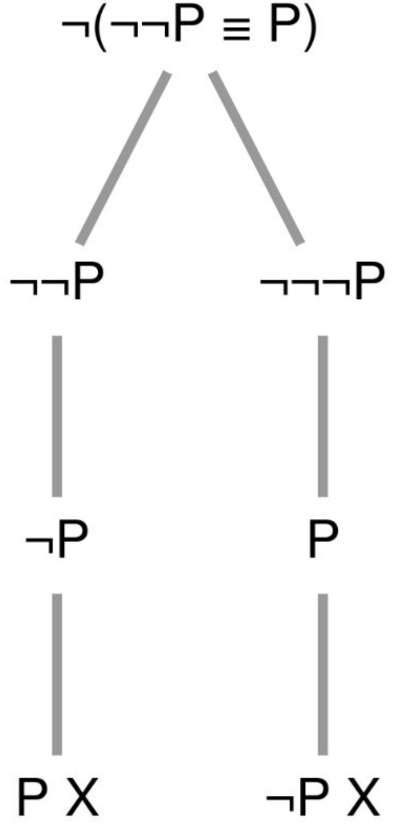
The algorithm in a nutshell - 3/4

- 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.
    - ¬¬P≡P was proved! ▮



◇Possibility rule (details)

◇ rule will add additional graph vertices, as follows:

- K modal logic requires no other changes to the graph.
- T modal logic requires a reflexive graph.
  - The algorithm adds the missing reflexive vertices as follows: for all nodes w_i, a $w_i \rightarrow w_i$ vertex will be added to the graph, if missing.
- S4 modal logic requires a reflexive and transitive graph.
  - The algorithm adds the missing transitive vertices as follows: for all nodes w_i, $w_{\square}$, $w_{\square}$, if the graph contains $w_i \rightarrow w_{\square}$ and $w_{\square} \rightarrow w_{\square}$ vertices, and the graph does not contain a $w_i \rightarrow w_{\square}$ vertex, then a $w_i \rightarrow w_{\square}$ vertex will be added to the graph.
- S5 modal logic requires a reflexive, symmetric and transitive graph.
  - The algorithm adds the missing symmetric vertices as follows: for all nodes w_i, $w_{\square}$, if the graph contains a $w_i \rightarrow w_{\square}$ vertex and the graph does not contain a $w_{\square} \rightarrow w_i$ vertex, then a $w_{\square} \rightarrow w_i$ vertex will be added to the graph.

□ Necessity rule (details)

□ rule is applied reactively:

- When the program first applies a □P w_i rule, it will look at the graph and find all possible worlds w_j that have a $w_i \rightarrow w_j$ vertex. Then for each w_j possible world, it will add a P w_j node to the tree.
- Afterwards, the algorithm can apply ◇P w_i rules, which will add more w_k possible worlds and more $w_i \rightarrow w_k$ vertices to the graph.
- The initial □P w_i rule gets remembered. Each time a new $w_i \rightarrow w_k$ vertex is added to the graph, □P w_i will be reapplied for each newly created worlds. Thus, for each new accessible possible world w_k, the software will add a P w_k node to the tree.

First-Order logic notations

For technical reasons, 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$x_1 x_2 ... x_n$)

- 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

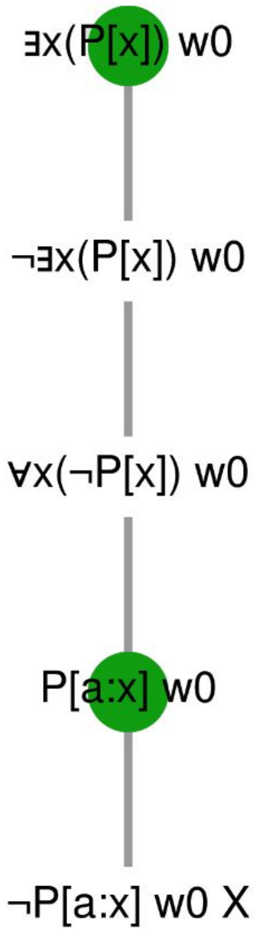
∃z(P[a:x,b:y,z]) w0

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

∀ For All rule

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

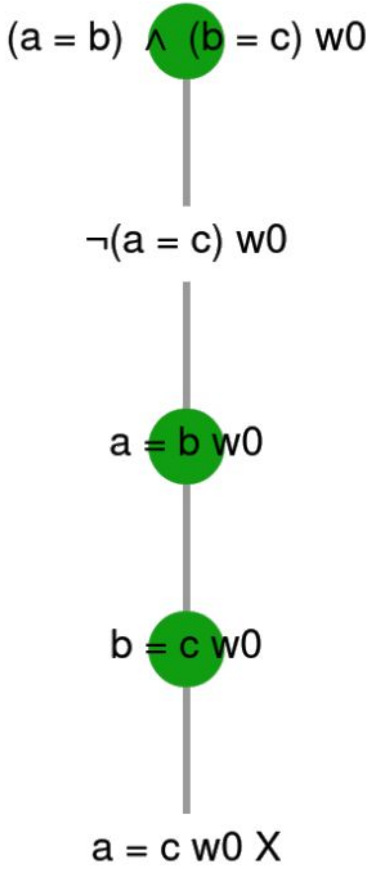
- For instance, in this example:
  - ∃x(P[x]) instantiates x as a:x and adds a P[a:x] node to the tree branch.
  - ∀x(¬P[x]) finds one object of type x (the previously instantiated a:x) and adds one node (¬P[a:x]) to the tree branch.



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.

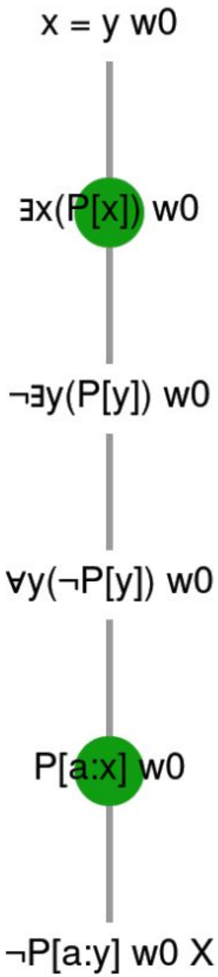
- In FOL, the algorithm detects a contradiction iff:
  - There are two nodes $P[a:x_1,b:x_2,\ldots]$ and
  - $\neg P[a:x_1,b:x_2,\ldots]$ (same arguments) on the branch.
  - Or there are two nodes $a:x_1 = b:x_2$ and
  - $\neg(a:x_1 = b:x_2)$ (same objects) on the branch.



Equality and ∀ For All rule

- The $\forall x_i$ rule will find on tree branch all instantiated objects of type x_i and any other y_i type equivalent to x_i ($x_i = y_i$).
- 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, has nothing to add.
  - 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.



More links

The software is available at: andob.io/incl
Source code (Rust): here

An initial, beta version: filos.ro/ls/inclcalculator
Source code (Kotlin): here
A presentation of the beta version: here