university of groningen / faculty of science and engineering

A Tableau Prover for Many-Valued Logics

Bachelor's Project Thesis

Yvonne Hoven, s2986280, y.j.hoven@student.rug.nl,
Supervisor: Prof Dr L.C. Verbrugge

Abstract: A semantic tableau prover was implemented in Prolog for the following three many-valued logics: First Degree Entalment (FDE), (Strong) Kleene's Logic (K_3), and the Logic of Paradox (LP). These logics have one or two truth values more than classical logic, which only has the truth values true and false. The tableau rules for these three logics are similar, but differ in regard to their use of the non-classical truth values. The tableau prover uses the sound and complete tableau rules of the corresponding logics, and uses the same precedence hierarchy for the connectives as humans. The system was tested in a survey on interface and usability with regard to two types of proofs and counterexamples. In the first case, the proof tree was made with a depth-first approach, and the second proof tree was made with a breadth-first approach. In this survey, the breadth-first search proof was preferred. No examination of efficiency was done.

1 Introduction

In contrast with classical logic, the kinds of logics used in this paper are not two-valued (true and false), but many-valued logics. These logics are also different from the fuzzy logics, as there is no graduation in truthfullness and falseness. Each formula is either true, false, neither or both. When considering classical logic, one can come across some interesting paradoxes, like the liar paradox (Priest, 2008). Consider the sentence "*This sentence is false*". If it is the case that the sentence is true, then what it says is the case. Hence it is false. So true concludes into false, and vice versa: if the sentence is false, then it is true. That is not ordinary. Thus, one comes upon truth value gluts (both true and false seem to be the case, as with the liar paradox) and truth value gaps (neither true nor false seems to be the case, which one could argue holds as well for the liar paradox).

As for programming, one can have a program that concludes yes/true/succeed, no/false/fail, in addition to the failure option of a non-terminating program, which keeps running forever. In that case, one does not know the resulting output value of that program.

Kleene found a solution to this kind of phenomenon in logic. He introduced the third truth value, being undefined (u) (Kuznetsov, 1994; Malinowski, 2014). This means that the value is not yet defined as either true or false, but it can get such a value later on, when more information is provided. We refer to this logic as K_3.

Later on, in order to not only fill the truth value gap, but also to include the truth value gluts in logic, the First Degree Entailment (FDE) logic was founded by Belnap (1977). This logic has the truth values true, false, both (true and false) and neither. For more information about this kind of logic see (Fitting, 2018; Hanson, 1980; Restall, 2017; Shramko, Zaitsev, and Belikov, 2017).

Consider Kleene's logic (K_3) as a super-logic of FDE, which has only truth value gaps besides true and false. One can also guess that there is a counterpart logic opposed to Kleene's logic, which does have truth value gluts, not not gaps. Indeed it exists, and it is called the logic of paradox, introduced by Priest (Petrukhin, 2017; Restall, 2017). We refer to this logic as LP.

This combination of logics is viewed in a nice way in lattices, in particular Ginsberg's bi-lattices (Ginsberg, 1986, 1988, as cited in Fitting, 1990). See Figure 1.1. When the top node does not exist, the semi-complete lattice for Kleene's logic is acquired. When the bottom node does not exist, the semi-complete lattice for the logic of paradox

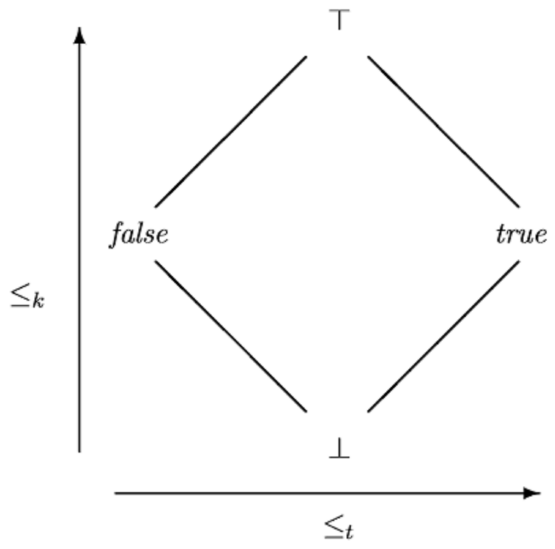

Figure 1.1: Bi-lattice representation of the FDE four-valued logic (Fitting, 2018).

is acquired. These lattices are often used in reports to explain the relations between the four possible truth values (Fitting, 1989, 1990, 1994, 2018; Loyer, Spyratos, and Stamate, 2004).

Bi-lattices show the relations between true, false, both and neither in two different ways: horizontally, the truth value (t) increases from left to right, and vertically, the amount of knowledge or information (k) increases from top to bottom. Let 'true' stand for true, let 'false' stand for false, let '$\bot$' stand for neither (i.e. underdefined), and let '$\top$' stand for both true and false (i.e. overdefined).

False and true are the least and greatest elements under the truth ordering (t); $\bot$ and $\top$ are least and greatest elements under the knowledge ordering (k). Such a lattice is a complete lattice, and it is a distributive one; it has 12 distributive laws. This means that the meet and join operations for each partial ordering are monotone with respect to the other ordering.

So, when one takes the meet ($\wedge$) of true and false ($true \wedge false$) in the truth value ordering ($\leq_t$), $false$ is derived (the truest/rightmost thing being less true/to the left of both of them). The join ($\vee$) of true and false ($true \vee false$) is $true$ (it takes on the value of the truest thing of the two). Similarly, $\top \vee \bot = true$ and $\top \wedge \bot = false$. These are operations on the truth ordering ($\leq_t$), the similar meet and join operations of the knowledge ordering ($\leq_k$) are $\otimes$ and $\oplus$.

The $\otimes$ is the consensus operator, thus $x \otimes y$ refers to the highest knowledge value that x and y can agree on (in the $\leq_k$ ordering): It refers to the truth value that is the highest one below both x and y, like the intersection operation on sets. Therefore, $true \otimes false = \bot$ and $\top \otimes true = true$. The $\oplus$ is the gullibility operator, which is similar to the union operation on sets. It accepts any knowledge value, thus $true \oplus \bot = true$ and $\bot \oplus false = false$ and $true \oplus false = \top$.

Likewise, there is an operation in the truth ordering for negation ($\neg$), which swaps the truth values left-right, so $\neg$false becomes true, $\neg$true becomes false, but $\top$ and $\bot$ remain unchanged (as there is no such opposite in the bi-lattice). The negation in the up-down order is called conflation ($-$). Thus, $-\bot$ becomes $\top$, $-\top$ becomes $\bot$, and true and false remain unchanged now, as there is no opposite of them in the bi-lattice.

1.1 This research project

The research question for this Bachelor's project was: Is it possible to build a complete and correct semantic tableau prover for the many-valued logics FDE, K_3 and LP in Prolog that has a satisfactory interface and usability, including counter-examples, being depth-first search and breadth-first search oriented, and which of those two is preferred?

The hypothesis was that the breadth-first search interface would be preferred, as it is the most similar to a tableau proof on paper.

The rest of this article is structured as follows: In the next section, the semantics and the tableau rules for FDE, K_3 and LP are presented. The third section describes the methods used for creating this semantic tableau prover system. The fourth section contains the results of the survey among students. In the final section, the discussion and conclusion are considered. Finally, in an appendix, the instructions for the participants and the tested queries can be found, along with the survey used and comments on the system.

2 Many-Valued Logics

In this section, more information can be found about the semantic tableau rules and proof trees of the three many-valued logics: FDE, K_3 and LP. One can skip this section, if one knows these logics already. For more information on these logics, see e.g. (Priest, 2008).

To start off, take a look at the four-valued First Degree Entailment logic. Later on, one can derive the semantics of Kleene's logic and the logic of paradox, by removing one of the four truth values and adding one extra branch closure rule to the proof trees. The closure rules are used to get either an inconsistent (closed) or consistent (open) tableau. Open complete tableaux lead to counter-examples for the to-be-proven statement. A tableau with at least one open branch is already an open tableau. For a closed tableau, all branches need to be closed.

2.1 First Degree Entailment

In contrast with classical logic, next to not having the implication binary connectives (denoted $\rightarrow$ or $\supset$) and thus no equivalence relation ($P \equiv Q := (P \supset Q) \wedge (Q \supset P)$), there is a need to express four different truth values in FDE. This is done by the interpretation $\mathcal{V}$, which is the mapping from propositional parameters to the set of truth values [0, 1, neither, both] (see Figure 2.4).

An input formula (say Q) may relate to true ($Q\rho 1$); it may relate to false ($Q\rho 0$); it may relate to both ($Q\rho 1$ and $Q\rho 0$); or it may relate to neither (). So, the fact that a formula is false (relates to 0) does not mean that it is untrue: it may also relate to 1. This is the main idea behind the semantics for FDE. See Figure 2.1 for some definitions of these semantics (given that the atomic formulas are known), where 'iff' stands for 'if and only if'. The figures in this paper are mainly used from (Kooi and Verbrugge, 2018), as they are clear to read.

In the proof trees in this paper, 'related to true' is noted as a plus sign (+) next to the corresponding clause, and 'related to false' is noted as a minus sign (−) next to the corresponding clause. Intuitively, $A, +$ means that A is true, and $A, -$ means that it is not related to true. Note that $\neg A, +$ no longer means the same, intuitively, as $A, -$.

$(A \wedge B)\rho 1$	iff	$A\rho 1$ and $B\rho 1$
$(A \wedge B)\rho 0$	iff	$A\rho 0$ or $B\rho 0$
$(A \vee B)\rho 1$	iff	$A\rho 1$ or $B\rho 1$
$(A \vee B)\rho 0$	iff	$A\rho 0$ and $B\rho 0$
$(\neg A)\rho 1$	iff	$A\rho 0$
$(\neg A)\rho 0$	iff	$A\rho 1$

Figure 2.1: Semantics of logics based on FDE (Kooi and Verbrugge, 2018).

Designated values $\mathcal{D}$ are the subset of the interpretations $\mathcal{V}$ that are considered valid. To check whether an inference is valid, one needs to check the premises (preconditions or antecedent), which contain knowledge that is given beforehand, to check whether conclusion can be derived from those premises. An inference is therefore valid (e.g. considered true) if for every interpretation $\mathcal{V}$, (1) one of the premises has a non-designated value) or if (2) the conclusion is related to a designated value while the premises are also related to a designated value. An inference without premises is a logical truth if for any interpretation $\mathcal{V}$, the interpretation of the conclusion is a designated value. The definition to check for validity can be seen in Figure 2.2.

$\Sigma \models A$ iff for every interpretation ρ, if $B\rho 1$ for all $B \in \Sigma$, then $A\rho 1$.

Special case, logical truth: $\models A$ iff for every interpretation ρ, $A\rho 1$.

Figure 2.2: Validity of inferences based on FDE (Kooi and Verbrugge, 2018).

Since validity in FDE is defined in terms of truth preservation, the designated values consist only of the values that are considered to be (at least) true in FDE, so 'true' and 'both true and false' are considered designated values in FDE.

To test the validity of an inference in a logic, FDE in this case, one checks whether the inference $A_1, \ldots, A_n \vdash_{FDE} B$ is valid in the FDE logic using a tableau. This is done by starting with an initial list of the form in Figure 2.3. The initial list contains a list of knowledge that one knows beforehand. The absence of the relation to true of the conclusion is added to that list ($B, -$ is added if the

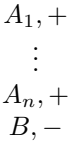

Figure 2.3: Deriving an inference of a logic based on FDE: the initial list of a tableau (Kooi and Verbrugge, 2018).

conclusion was B). In this way, one investigates in a methodological manner whether a counter-example exists for the inference, namely, an interpretation $\mathcal{V}$ that makes $A_1, \ldots, A_n$ true, but not B.

Next, the semantic tableaux rules in Figure 2.5 (with underlying truth tables for FDE found in Figure 2.4) are used to come to the conclusion whether the inference is indeed valid or not valid.

The (only) branch closure rule of FDE applies when the proof tree contains both $A, +$ and $A, -$ on the same branch for some formula A. Only then will the branch of the proof tree close.

$f_\neg$	
1	0
b	b
n	n
0	1

$f_\wedge$	1	b	n	0
1	1	b	n	0
b	b	b	0	0
n	n	0	n	0
0	0	0	0	0

$f_\vee$	1	b	n	0
1	1	1	1	1
b	1	b	1	b
n	1	1	n	n
0	1	b	n	0

Figure 2.4: The 3 truth tables for FDE (Kooi and Verbrugge, 2018).

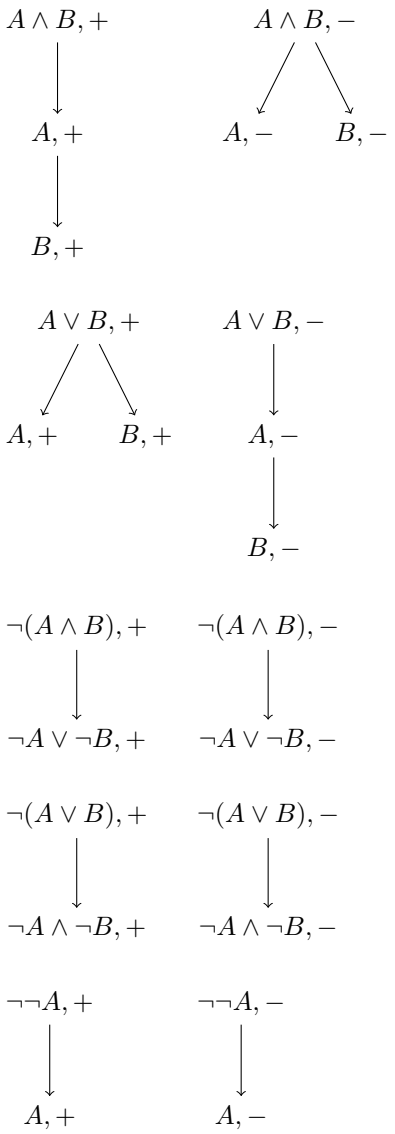

Figure 2.5: Semantic tableau rules for FDE, K_3 and LP (Kooi and Verbrugge, 2018).

2.2 Kleene's logic

Kleene's logic (K_3) and its semantics can be derived from the FDE logic, when the truth value 'both true and false' is removed, while an extra branch closure rule is added. When one non-classical logic truth value is removed, the other one is called 'i' by convention (instead of 'n', as before with FDE).

Consequently, in Kleene's logic, there is only a truth value given to the truth value gaps (neither true nor false being the case). This is also called the exclusion constraint, as there is no possibility that a propositional parameter A both relates to true and to false ($A\rho 1, A\rho 0$).

The designated value in K_3 is only the truth value 'true'. Otherwise, the validity is handled in a similar way: An inference is valid if for every interpretation $\mathcal{V}$, (1) one of the premises is false (not a designated value, e.g. 'false' or 'i') or if (2) the

conclusion is related to a designated value while the premises are also related to a designated value e.g. 'true' for K_3. This is still conveyed by Figure 2.2.

Furthermore, in Kleene's logic, the law of the excluded middle is not valid, so $\nvdash_{K_3} p \vee \neg p$ (check for yourself when the valuation of P is 'i' in the truth tables in Figure 2.6). Actually, there are no logical truths (tautologies) in K_3. $(p \wedge \neg p) \vdash_{K_3} q$ is valid in K_3, since the '$p \wedge \neg p$' part (p and not p) can never assume a designated value, and therefore q vacuously follows (see Figure 2.2), hence contradictions can entail anything.

The extra closure rule for K_3 compared to FDE is: close the branch if it contains both $A, +$ and $\neg A, +$.

$f_\neg$	
1	0
i	i
0	1

$f_\wedge$	1	i	0
1	1	i	0
i	i	i	0
0	0	0	0

$f_\vee$	1	i	0
1	1	1	1
i	1	i	i
0	1	i	0

Figure 2.6: Truth tables for K_3 and LP (Kooi and Verbrugge, 2018).

2.3 Logic of paradox

When instead of the exclusion constraint, the exhaustion constraint is used, the LP logic is acquired. The truth value 'neither true nor false' is dropped from the FDE logic, and the 'both true and false' value is named 'i'. This means that now truth value gluts are considered (instead of truth value gaps). A propositional parameter A in this logic may thus relate to true ($A\rho 1$), to false ($A\rho 0$), to both true and false ($A\rho 1, A\rho 0$), but never to neither true nor false ().

The designated values in LP are 'i' (both true and false) and 'true', similar to the 'at least true' of the FDE logic. The validity is handled in the same way as with FDE: the inference is only not valid if the premises have a designated value while the conclusion does not have a designated value.

In contrast with Kleene's logic, in the logic of paradox, contradictions do not entail everything: $p \wedge \neg p \nvdash_{LP} q$, take the value 'i' for p and the value 0 for q to see a counter-example, as the premise's value 'i' is a designated value in LP. Modus ponens is invalid: $p, \neg p \vee q \nvdash_{LP} q$, where the counter-example assigns the value 'i' to p, and 0 to q, see Figure 2.6.

The extra branch closure rule for LP compared to FDE is: close the branch if it contains both $A, -$ and $\neg A, -$.

2.4 Counter-models

If the original inference is valid (meaning that the conclusion B has a designated value, thus B being related to true (+) whenever all premises have designated values (related to true), the proof tree would have come to contradictions on every branch when starting with with the initial list (which includes the conclusion B, being related to false (−)). The (only) closure condition of FDE occurs when a branch of the proof tree contains both $A, +$ and $A, -$ for some formula A.

Otherwise, when the inference is not valid, there would be (at least one) branch of the proof tree which is complete (all rules that can be applied, have been applied) and open (no contradiction found in the proof tree regarding $A, +$ and $A, -$ for FDE). Counter-models can be read off from the open complete branches of tableaux. For every propositional parameter/atom p, if there is a node of the form $p, +$, set $p\rho 1$ (p is related to true); if there is a node of the form $\neg p, +$, set $p\rho 0$ (p is related to false). No other facts about p obtain. One can then make the counter-example for the inference with all provided relations ρ. Propositional parameters without explicit relations are not given any relations, as nothing about p obtains (in contrast to classical logic, where one can come up with arbitrary values (true/false)).

When the proof tree is not closed off in the logic K_3, henceforth the inference being invalid in K_3, a counter-model can be read off from a open complete branch of the tableau in the same way as with the First Degree Entailment logic.

When the inference is invalid due to an open complete branch in the proof tree in the logic of paradox (LP), a counter-model can be read off from the open branch of the tableau in a different way: if

$p, -$ is not on the branch (but $p, +$ may be), set p related to 1 (true). If $\neg p, -$ is not on the branch (but $\neg p, +$ may be), set p related to 0 (false). No other facts about p obtain.

If one would add the extra branch closure rules of both K_3 and LP to FDE, e.g. if one makes an interpretation of FDE with both exclusion and exhaustion, then one would end up with classical logic again. This follows naturally, as both non-classical truth values are removed from the semantics.

Counter-examples from tableaux in Kleene's logic and the logic of Paradox tableaux are also FDE counter-examples. Additionally, there are more branches left open with the FDE logic than with K_3 or LP, thus more counter-examples can be found with FDE. Consequently, less inferences are valid in FDE than in K_3 and LP, as FDE has less closed-off proof trees.

3 Methods

While most provers are about classical logic (two truth values) and/or modal logic (multiple worlds) (Loo, 2017), only some of them using the programming language Prolog (Manthey and Bry, 1998; Stickel, 1984, 1988), this one is about the three non-modal many-valued logics FDE, K_3 and LP.

Whenever such a many-valued logic prover is built (by for example: Beckert, Hahnle, Oel, and Sulzmann (1970); Beckert, Gerberding, Hahnle, and Kernig (1992); Beckert and Posegga (1995)), all kinds of programming languages are used, but rarely Prolog. In this paper however, Prolog is used as the programming language. In particular, the SWI-Prolog interpreter is used (version 7.6.3, but newer versions should be compatible too), which can be downloaded from https://www.swi-prolog.org/download/stable.

To understand what our system is about, Subsection 3.1 explains how a tableau, i.e. a proof tree is built, and how one can get the counter-examples from it. After that, Subsection 3.2 explains what the programming language Prolog actually is about and how it works. Additionally, Subsection 3.3 explains the design choices that have been made. The final Subsection 3.4 explains the survey held regarding the many-valued semantic tableau prover.

3.1 Building a tableau

When buidling a tableau, one first needs to consult Figure 2.3 to start the inference, and then consult the semantic tableaux rules of Figure 2.5 to draw the actual tableau. Only after that, one can check for the closing rules (being different for FDE, K_3 and LP) of the branches of the tableau.

Doing this with pen and paper requires just that, but programming a system to build those tableaux and counter-examples automatically takes some more effort (Paulson, 1989, 1996, 1999). The semantic tableau rules need to be implemented in the programming language, which requires some consideration. All rules need to be handled correctly, the relations of true, false, both and neither need to be represented, and the tableaux need to be represented.

In this paper, the tableau is actually printed out by the system. Additionally, this system will give the actual counter-examples that can be found from those tableaux.

Humans try to solve tableaux in the simplest way possible, and this system uses the same approach. This means that first all rules are applied that do not split the tableau into different branches, to make the tableau as small and comprehensible as possible. Only when there is no other rule applicable anymore, rules that split the tableau into branches are applied. Consequently, the rules for $A \wedge B, +$ and $A \vee B, -$ and negation are preferably used before the rules for $A \wedge B, -$ and $A \vee B, +$.

There is no consensus about what branch one applies the next semantic tableaux rule on. Henceforth, one can either use a so-called depth-first search (DFS) or a breadth-first search (BFS) approach to make and solve tableaux. This is up to the preference of the user of a tableau. For this system, both DFS and BFS are an option to use for completing the tableaux.

When considering DFS, one first completes the most leftward branch, before applying rules to another branch (vertical). When considering BFS, one applies rules in a horizontal way, by switching between branches after applying one rule. See Figure 3.1 for the differences between DFS and BFS.

DFS goes in this order: (1), (2), (3), (4), (5), $q, +$, $p, +$, x (closing the branch), then next to it $\neg p, +$, back to $\neg q, +$ at level 6, then $p, +$, x (closing the branch), and finally, next to it, $\neg p, +$.

BFS goes in this order: (1), (2), (3), (4) (5), (both $q, +$ and $\neg q, +$), (6), followed by all 4 mentioned on level 7, from left to right), and then closes the two branches (from left to right).

So, DFS checks $q \vee \neg q$, then $p \vee \neg p$ on the left, and then $q \vee \neg q$ and $p \vee \neg p$ again on the right, while BFS checks $q \vee \neg q$ once, and then $p \vee \neg p$ once.

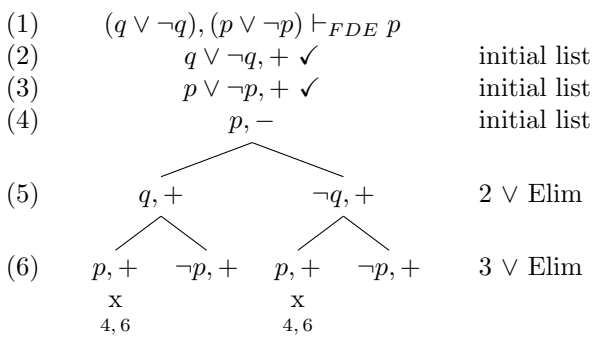

Figure 3.1: Differences between depth-first and breadth-first search order on the inference: $(q \vee \neg q), (p \vee \neg p) \vdash_{FDE} p$

In Figure 3.1, 'initial list' stands for the assertion of antecendents e.g. the premises, and for the assertion of the lack of 'related to true' of the conclusion, and '$\vee$ Elim' and '$\wedge$ Elim' stand for the elimination rule of the disjunction symbol and conjunction symbol respectively (with the row number of that symbol in front). Where a branch of this proof tree closes, the numbers are shown which led to the closure of the branch.

The two branches are closed off because $p, -$ and $p, +$ may not appear on the same branch according to FDE (the branch closure rule), and thus subsequently for its super-logics K_3 and LP. Simultaneously, no closure rule in FDE, K_3 or LP exist for having both $p, -$ and $\neg p, +$ on the same branch, so those branches remain open, consequently being the branches from which one can read off counterexamples showing that the inference is not valid.

3.2 Prolog

To get a really complete understanding of what Prolog is about and how to implement it, it is recommended to check out the Prolog book by Bratko (1990) or Blackburn, Bos, and Striegnitz (2006). Here a necessary but not complete/sufficient explanation is given.

3.2.1 Sentences, knowledge bases, queries and symbols

Sentences in Prolog code (called clauses) are separated into two parts: a knowlegde base which holds the factual information, and rules to derive new information.

To run the system, one asks a query/question to the prover, which it will then try to verify by deducing new facts using the existing knowledge base and the rules by modus ponens. The knowledge base and rules are searched through from top to bottom, and the clauses are processed from left to right, so therefore Prolog has a natural depth-first search strategy.

When it succeeds, it prints 'true', and when it fails, it prints 'fail' to the output. Whenever a true is printed out, it could also be the case that in addition to that answer, there is another possibility to derive the answer to the query. That alternative answer can then be found by the additional input ';'. Prolog will then backtrack to the goal at the last choice point, in order to try a different path to derive and satisfy the same goal. When there is no alternative answer, the output will be 'fail'.

There are atoms, functions, and variables in Prolog. An atom starts with a lowercase letter and has no arguments. A function, in contrast, has one or more arguments, while still starting with a lowercase letter. A variable starts with an uppercase letter and can be unified with (take on the value and meaning of) an atom or function.

Additionally, there are some operators predefined, like ',', ';' to denote conjunction and disjunction respectively. As told before, the disjunction operator can also be used to ask an alternative unification for a variable for the answer of the query. There is also a symbol ('[]') to denote a (now empty) list, which can consist of multiple atoms.

3.2.2 Clauses

As told before, clauses are either facts or rules. Rules have a head an a body, while facts only consist of the head. See the example Algorithm 3.1 for how this looks in Prolog. You can read rules as the head being true if the body of the rule is found true.

Rules can also be recursive. In that case, the rule calls itself, with a slight modification: it solves a smaller instance of the same problem, after solv-

ing a basic case (see Algorithm 3.1 for the code of printList()).

A fact can also be a dynamic one, when defined as such (by typing ':-dynamic(predicate(arguments)).' (see Algorithm 3.1) where the arguments can be the underscore character '_' to not be specific about what can be used. Therefore, underscore is called the anonymous variable. Facts can be added by asserting them (assert(happy(finn)).) or deleted by retracting them (retract(happy(kim))), see Algorithm 3.1.

Algorithm 3.1 Examples of Prolog clauses

```
happy(kim).
%kim is happy, a fact

couple(kim, mike):- loves(kim, mike), loves(mike, kim).
%kim and mike are a couple if kim loves mike and vice versa, a rule

descend(X,Y):- child(X,Y).
descend(X,Y):- descend(Z,Y), child(X,Z).
%an example of a recursive rule

printList([]) :- nl.
printList([H—T]) :- write(H), printList(T).
%example of a recursive way to traverse a list and print it with the predicate 'write'.

:-dynamic(happy(_)).
%now one can add or delete knowledge about the predicate happy() with 1 argument.

assert(happy(finn)).
retract(happy(kim)).
assert(couple(pin,finn)).
```

3.2.3 Functions

There are some functions which write the output ('write()', 'nl', 'writeln()' where the first just writes what is in between the brackets, the second prints a new line, and the third does both). These output functions are used to produce the proof tree and the counter-examples.

There are also functions which can help to find several facts or rules in your code. These are 'findall(X,happy(X),List)', 'bagof(X,happy(X),List)', and 'setof(X,happy(X),List)'. The first argument will be searched for in places where that argument occurs in the second argument, and then it gets placed in the third argument (a list).

The differences between these three are that (1) 'findall' is equivalent to 'bagof' if all free variables are bound with the existential operator, except that then 'bagof' fails when the second argument has no solutions. The difference between 'bagof' and 'setoff' (2) is that 'setof' sorts the result to get a sorted list of alternatives without duplicates.

3.2.4 Negation

Prolog also has its own logic about negation. This form of negation, called 'negation by failure', means that a query cannot be found true if there is no evidence for it of being true (when there is no 'X' as a conclusion, one concludes 'not X'). This is quite problematic for (many-valued) logics such as FDE, K_3 and LP: it's difficult to specify the meaning of such programs, and queries involving free variables may behave unexpectedly. So, Fitting has suggested 'negation by refutation' (1989). In any case, in this paper the negation is implemented as a symbol rather than the logic meaning of it. The symbols '+ , - , * , / , < , > , = , : , . , & , ∼' are already used by Prolog itself, so in this paper 'not' is used (an atom in Prolog) as negation.

3.3 The program

For this system, a subset of the many-valued logics (K_3, LP, FDE) of (Priest, 2008) is used, subsequently making this prover sound, but not complete. There are certain restrictions to this system. For example, one cannot input a premise that may cause the tableau to branch, and there can only be one branching moment per proof tree. For the survey, all inferences were proven correct.

In general, this system is started by typing a query consisting of a function with a list, a '|' symbol, the logic (either fde, k3 or lp), and another list. The first list may contain the premises or it may be left empty (no premises beforehand). The list after the '|' symbol contains the to be proven conclusion.

In the program, that complete query input line triggers the first rule, which asserts the premises (being related to true) one by one into the knowledge base, prints them to the output, and prints

the conclusion (being not related to true). In Figure 3.2, you can see an example of an input for the systems, which corresponds to '$p \wedge q \vdash_{FDE} r \vee (p \wedge \neg p)$'. In Figure 3.3 the corresponding output is shown for the depth-first search system and in Figure 3.4 the corresponding output is shown for the breadth-first search system.

```
prove([p,'&',q], '|',fde, [r, 'V', '{',p, '&', not, p,'}']).
```

Figure 3.2: Possible input for the Prolog program.

In addition to asserting and printing the premises and conclusion, the first rule also calls upon other functions to try to prove the conclusion (e.g. make the tableau), and the function to show what the counter-examples may be. Each step in the tableau-making process is printed to the output, however the order of rules applied may not be top-down: systematically, the rules that do not make new branches are applied first, while the rules that make new branches are postponed until no other rules apply anymore.

```
SWI-Prolog (AMD64, Multi-threaded, version 7.6.3)
File  Edit  Settings  Run  Debug  Help
?- prove([p,'&',q], '|',fde, [r, 'V', '{',p, '&', not, p,'}']).
p&q,+
rV{p&notp},-

premises solving:
p&q,+
p,+
q,+

conclusion solving:
rV{p&notp},-
r,-
{p&notp},-
  /\p&notp,-
p,-

positive literals:
|p, + | q, + |
negative literals:
|r, - | p, - |
Closed branch fde has p,+ and p,-
true ;
\
notp,-

positive literals:
|p, + | q, + |
negative literals:
|r, - | not p, - |
fde branch is open, counter-example found fde:
p,+ q,+  r,- not p,-
p+? set pr1, notp+? set pr0
Set p related to true (p rho 1)
Set q related to true (q rho 1)
No other facts about rho obtain
true ;
false.

?-
```

Figure 3.3: Corresponding depth-first search output of the Prolog program of Figure 3.2.

```
SWI-Prolog (AMD64, Multi-threaded, version 7.6.3)
File  Edit  Settings  Run  Debug  Help
?- prove([p,'&',q], '|',fde, [r, 'V', '{',p, '&', not, p,'}']).
p&q,+
rV{p&notp},-

premises solving:
p&q,+
p,+
q,+

conclusion  solving:
rV{p&notp},-
r,-
{p&notp},-

//p&notp,-
p,-   OR   notp,-

positive literals:
|p, + | q, + |
negative literals:
|r, - | p, - |
Closed branch fde #1 has p,+ and p,-
true ;
positive literals:
|p, + | q, + |
negative literals:
|r, - | not p, - |
branch #2 fde is open, counter-example found fde:
p,+ q,+  r,- not p,-
p+? set pr1, notp+? set pr0
Set p related to true (p rho 1)
Set q related to true (q rho 1)
No other facts about rho obtain
true ;
false.

?-
```

Figure 3.4: Corresponding breadth-first search output of the Prolog program of Figure 3.2.

3.3.1 General and individual components

To get a general idea of the structure of the code used for this paper, see Figure 3.5. Note that this is not the way that the code actually looks like, but rather explains what happens in the Prolog code. For the full program, check: github.com/YvonneHoven/PrologTableauProver.

3.3.2 Design choices

For this system, the logic of Prolog itself has been avoided by using different symbols than the logical operators. This was considered necessary, since e.g. negation in Prolog does not work like the negation of our many-valued logics, and the logic of paradox has no modus ponens while Prolog does use that.

To print the full tableau in the breadth-first search, the use of the ';' operator in Prolog, which is used as a disjunction, was also rejected. Otherwise, one would print the other option of the disjunction not on the same line anymore. For depth-first search however, the disjunctive Prolog operator ';' was used, to print another branch every time another answer is requested.

The input should be given to the program in list form, so that (theoretically endless) inferences can be proven or disproven, as Prolog can use recursion

%%% helper functions.
assert([Premises]):- write(Premises, +),
assert([Premises with the rules applied to it]).

findall(Variables, Function in which the variable occurs, List):- putting all variables (that are in that function) into a list.

check(TrueP, FalseP):- check if there are any variables put into the list mentioned, e.g. whether there are branching rules which still need to be applied.

makecounterexamples(logic):-
printLiterals, printCounters(logic).
printLiterals:- prints all literals on the branch.
printCounters(logic):- prints for fde/k3/lp literals that caused the branch to close or gives related to true and false for atoms in counter-examples.

%%% The starting function, which calls upon other functions to solve the tableau.
prove([premises, |, logic, [conclusion]):-
assert(premises), write(conclusion, -),
proof(premises, +), proof(conclusion, -), findall([Y], toprove(Y, +), TP), findall([X], toprove(X, -), FP), check(TP, FP), makecounterexamples(logic).

%%% the logic rules
proof(A, &, B, +):- proof(A, +), nl, proof(B, +).

proof(A, V, B, +):- assert(toprove([A, V, B], +)).

prove([not, {, H, &, H2, }|T], '+'):- write(not, H, V, not, H2, +), proof((not, H), V, (not, H2), +), nl, prove(T, +).

Figure 3.5: Most important components of the Prolog program

with lists.

To be able to check for closure rules and thus closing of branches, assertions of literals are used to keep track of all literals on the branches. In the end, those literals are checked by the logic that was used (e.g. either FDE, K_3 or LP), to use the corresponding closure rules to close tableaux or to give the corresponding counter-examples.

3.4 About the survey

To measure the user-friendliness of the interface, a survey was held among 7 students of the Univiversity of Groningen, who have followed the course 'Advanced Logic'. All students used this Prolog tableau prover system in the same room, on the same computer, with the same SWI-prolog program, to cancel out confounding variables of noise and computer processors. In all cases, all the participants (students) had to use the same inputs of inferences, consequently leading to the same outputs.

First, all participating students had to make one tableau on paper, to determine whether they preferred a depth-first search (DFS) or breadth-first search (BFS) approach themselves. That tableau proof was: $\neg A \wedge \neg E \vdash_{FDE} \neg((A \wedge B) \vee (D \wedge E))$.

After that, a block of 3 (one for the logic FDE, one for K_3, and one for LP) DFS inferences had to be tested by the participants through the system, which was followed by a survey concerning the DFS approach. Then another block of almost the same 3 tests (using different variables, but the same operators) had to be tested for BFS, after which a survey was held regarding the BFS approach.

In addition, another block of both DFS and BFS inferences was done (with other inferences), with the same two corresponding surveys. This was done to give the participants a better grasp of the differences between the two systems, since they have to fill in the survey again when they have tested both systems already. Participants were told that they were going to work with and be asked about two different systems, D and B. It was made clear to them that each survey is only about the system of the most recent block.

These surveys were made in accordance with the System Usability Scale (SUS) (Brooke, 1996): 10 questions, with a score between 0 and 4 (the Likert scale: between strongly disagree and strongly agree). Questions that are phrased negatively, to avoid quick mindless answering, have the opposite score between 4 and 0.

This final score of 10 questions was then normalized to a scale between 0 and 100 (total score times 2.5). The survey was held twice for both the DFS

and BFS search strategies, and those two scores per search strategy were averaged per participant. Furthermore, the average scores per search strategy as a whole were compared to each other, in the way of the SUS: a higher score means a higher satisfaction. Furthermore, the scores are similar to grading systems (divided by ten): a six is sufficient, an eight is a good score.

The queries that were tested represent the possible diversities in the logic operators ('not', '&', 'V'), the diversity in the possible logics (FDE, K_3, LP), and represents both open branches with counter-examples and closed branches.

For the instructions given to the participants beforehand and the Prolog queries that were tested by the participants, see the appendix. The questions asked in the survey and the comments on the system are also included in the appendix.

4 Experiments and Results

Let us first take a look at the raw data that was acquired from this experiment. There were 7 participants, which all, except for one, used the breadth-first search strategy on paper.

All odd questions were phrased positively, while all even questions were phrased negatively.

Per participant, the total (implicit) points were derived by each survey (two times DFS and two times BFS), after which the points were normalized to system usability scores by multiplying the points by 2.5. The results of this can be seen in Table 4.1.

Furthermore, the average of both DFS surveys per participant, and the average of both BFS surveys per participant were calculated. This score was also used to derive the remaining graphs and figures.

In the first Figure (4.1), system usability scores per participant can be seen. This shows the (normalized) scores, averaged per search strategy. Since there are 7 participants, each line in the graph represents one participant. The scale of the y axis goes from 0 to 100, because that is the possible range of the normalized system usability scores. All lines except for one show a slight slope upwards.

As said before, the participants had to make one tableau proof on paper. The distribution of how many participants used which search strategy in that tableau proof is 1 participant for DFS versus 6 participants for BFS.

Table 4.1: Normalized raw data: system usability scores per search strategy. In the first column, the number of the participant is given, together with the search strategy that the participant used on paper to test the inference.

participant number and search strategy used	score DFS 1	score BFS 1	score DFS 2	score BFS 2
1 B	62.5	57.5	60	57.5
2 D	72.5	97.5	85	82.5
3 B	65	70	52.5	70
4 B	72.5	75	70	77.5
5 B	47.5	50	50	52.5
6 B	85	90	95	97.5
7 B	87.5	95	95	97.5

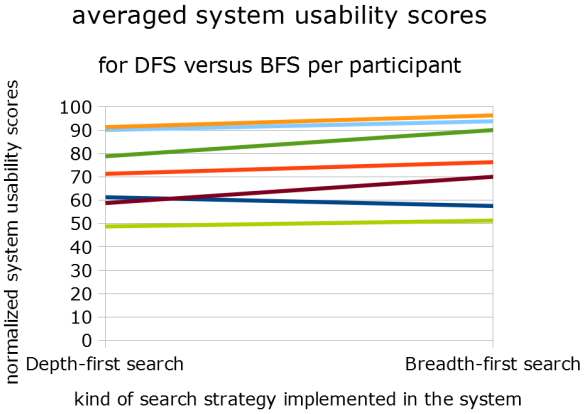

Figure 4.1: Comparison between the average system usability scores per search strategy. Each line represents the scores of a single participant.

Interestingly, the distribution of which search strategy got a higher score is the same, as 6 out of 7 lines in Figure 4.1 go upwards, while 1 goes down. In Figure 4.1, it is also visible that the difference between participants can be rather big (scores of 50 versus 90), while the difference within participants is rather small (for example: 92 versus 97), though existent.

Let us now look at the distribution of points given by the survey per search strategy, which can be seen in Figure 4.2. For all participants, for both DFS surveys, there was on average: 0.5 times given

a score of 0 points (which is the most unsatisfying option), 8 times score of 1 point, 11.5 times a score of 2 points (the neutral option), 27 times a score of 3 points, and 22.5 times a score of 4 points (which is the most satisfying option).

For the average of both BFS surveys over all participants, there was on average: 1 time a score of 0 points given (most negative option), 5.5 times a score of 1 point, 9.5 times a score of 2 points (neutral), 25.5 times a score of 3 points, and 28.5 times a score of 4 points (most positive option).

For BFS, the bar of 4 points is the highest bar, while for DFS that is the case for the bar of 3 points.

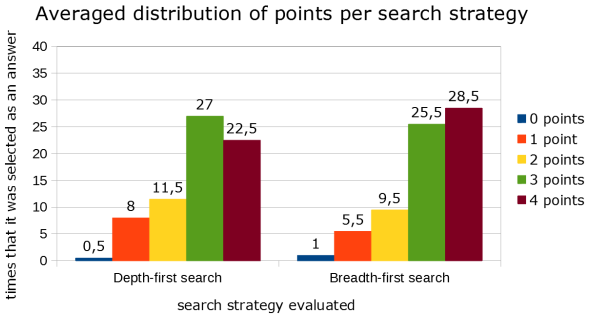

Figure 4.2: Averaged distribution of points per search strategy.

When taking into account that all points are added up to make up the normalized system usability scores, it is also interesting to look at the (total) points per survey and per search strategy. This can be seen in Figure 4.3. The total number of points for the first and second survey for DFS and the first and second survey for BFS are compared.

What is interesting here, is that DFS got more points in the second survey, compared to the first survey. This was expected, as participants would get to know the system better, after using it for a while. BFS scored overall more points than DFS.

When one takes the average of those points over all participants, without excluding participants as outliers, the averaged normalized system usability scores per survey per search strategy are derived. This is done by dividing the points in Figure 4.3 by 7 participants and multiplying by 2.5 to get the normalized version of the system usability scores, which can be seen in Figure 4.4 with standard deviations and in 4.6 without standard deviations.

The standard deviations in Figure 4.4 are high,

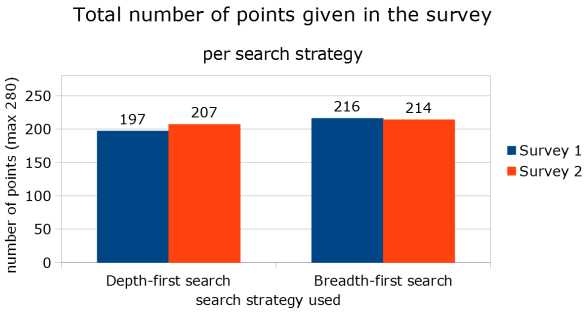

Figure 4.3: Number of total points given in the surveys per search strategy. The maximum score is 4 points for all 10 questions for all 7 participants (280).

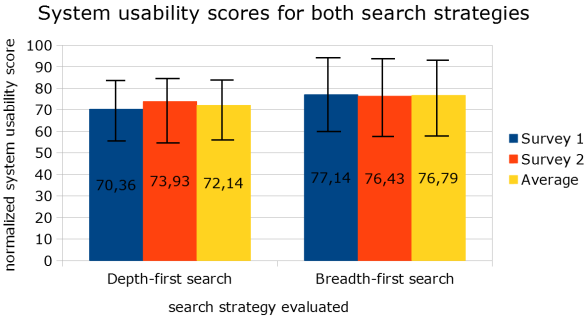

Figure 4.4: Normalized system usability scores per search strategy, averaged over all participants, with standard deviations.

because participants gave scores between 47.5 and 97.5, but since the important part is the difference between the search strategies (which can be seen more clearly in Figure 4.5) rather than in-between, this figure is also shown without standard deviations in Figure 4.6.

In Figure 4.5, the difference between system usability scores is taken as the breadth-first search strategy minus the depth-first strategy, to have as many positive (and therefore understandable) values as possible.

Although the standard deviations are high, the average and most of the standard deviation parts are above the null line, including the whole standard deviation of the average of both surveys per search strategy (which goes from 0 to 10). This means, since we took BFS − DFS as value, and since it is a positive value, that the BFS scores are really higher than the ones for DFS.

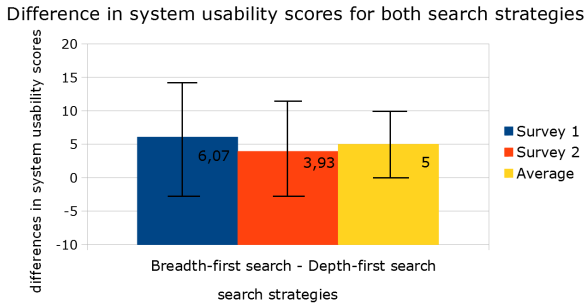

Figure 4.5: Differences between system usability scores of the breadth-first search strategy and depth-first search strategy.

What is interesting in Figure 4.6, is that the normalized system usability scores averaged over all participants for the BFS strategy are almost the same for survey 1 and 2, consequently being almost the same as the average of the two surveys for BFS.

Again, overall, the BFS scores are higher than the DFS scores, and DFS has a higher system usability score with the second survey, due to participants getting used to the two systems.

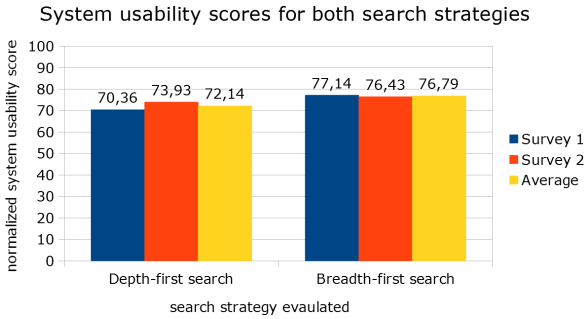

Figure 4.6: Normalized system usability scores per search strategy, averaged over all participants.

If the system usability scores for averaged DFS and averaged BFS would add up to 100, then 48.37% of the points given in the surveys would have gone to the DFS strategy, and 51.63% of the points given in the surveys would have gone to the BFS strategy. This can be seen in Figure 4.7.

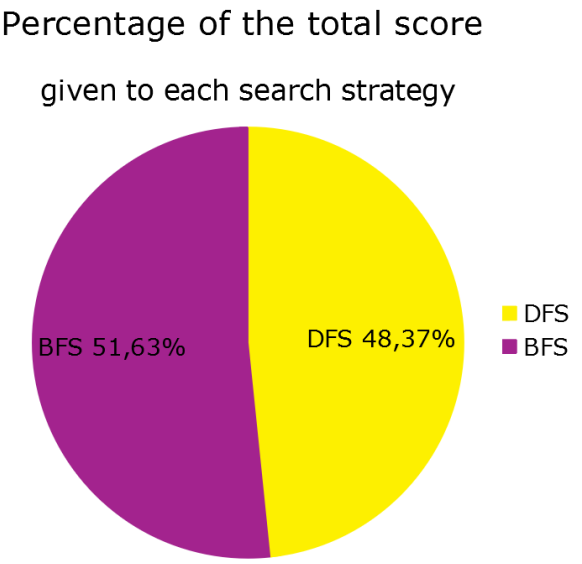

Figure 4.7: Percentage of the total score given to both search strategies.

5 Discussion and Conclusion

This research project started out with the research question whether it was possible to build an automatic semantic tableau prover for many-valued logics in Prolog with a likable interface for both depth-first and breadth-first search, and which one of them would be the preferred in terms of usability.

The usability of these two systems was tested by a System Usability Score in a survey with a Likert scale by 7 participants of the Univerity of Groningen, who had followed the course Advanced Logic.

As told before, there is not much research to find about many-valued logic provers in Prolog, let alone about the usability of such tableau provers. Therefore, my research cannot be compared to other research yet.

It became clear that it is possible to build a tableau prover for many-valued logics in Prolog, but that implementing a breadth-first search interface in Prolog's depth-first search structure requires some further thoughts.

Furthermore, it was apparent that the breadth-first search interface of the tableau prover was (slightly) preferred over the depth-first search interface by the participants of this experiment.

What was interesting in the results, is that for the depth-first search strategy there was a notice-

able increase in likability for the system in the second survey, as participants would get to know and get used to the two systems more, and therefore understand and like it more.

If converted to grades, the depth-first search would have gotten a 7.2 and the breadth-first search would have gotten a 7.7 by the 7 participants of this experiment.

The most informative graph is the one where each line represents one participant (Figure 4.1). This is the case because you can see the difference in likability between participants (which is large) and the difference in usability scores within participants (which is small). Subsequently, it it visible that the breadth-first search strategy of the system was preferred, as most of the lines go upwards.

Also, in the comments of the survey, it was stated multiple times that participants found that the different solutions to a proof (represented by branches of the tableau) were more clear in the B system (breadth-first search), than in the D system (depth-first search). One participant even noted that the proof output in the B system looks most similar to what students have to write on paper for the course Advanced Logic.

This reflects the point that for both surveys, the breadth-first search interface got a higher system usability score.

5.1 Critiques

When the results are critically evaluated, one comes across some assumptions and limitations of the system and the experiment.

The proof that participants had to make on paper was short: only one branch splitting was necessary. The automatic tableau prover also had this limitation.

This could have affected the results, as one could argue that a depth-first search strategy has a bigger advantage than breadth-first search strategy, as the tableau proofs grow big with many branches.

This is because with breadth-first search, one could lose sight of all formulas and literals on all different branches, while for depth-first search you continue on one branch till it is completed. This could mean that this research only confirmed that for small tableaux, the breadth-first search strategy was preferred both on paper and in the system.

Also, all 7 participants came form the University of Groningen, and had followed the Advanced Logic course there.

This could have affected the results, as professors of the University of Groningen have had an influence (by teaching in a certain way) on the preference of search strategies used on tableaux by students.

This could mean that students at other universities, with different professors, have other preferences for tableau prover interfaces.

5.2 Future research

Based on the above-mentioned limitations, it is highly suggested that for future research, (1) provers are made that can handle bigger proofs, (2) bigger proofs are tested, and (3) students across different universities participate in the experiments.

It could also be interesting to expand the tableau prover to different kinds of logics like classical logic or other many-valued logics like Ł$_3$, RM_3.

Finally, for future research on usability of automatic tableau provers building on this, it is advised to take some of the comments into account that were derived form this experiment, which can be seen in the Appendix. Some examples of student's suggestions are: make a manual for the system for how to interpret it, do not solve the premises and conclusion apart from each other, and leave the counter-model as it is now.

References

B. Beckert and J. Posegga. Leantap: lean tableau-based deduction. *Journal of Automated Reasoning*, 15:339–358, 1995.

B. Beckert, R. Hahnle, P. Oel, and M. Sulzmann. The tableau-based theorem prover 3tap - version 4.0. 1970. doi: 10.1007/3-540-61511-3_95.

B. Beckert, S. Gerberding, R. Hahnle, and W. Kernig. The tableau-based theorem prover 3tap for multiple-valued logics. In *Automated Deduction - CADE-11*, 1992. doi: 10.1007/3-540-55602-8_219.

N. D. Belnap. A useful four-valued logic. In J. M. Dunn and G. Epstein, editors, *Modern Uses*

of Multiple-Valued Logic, pages 5–37. D. Reidel, 1977.

P. Blackburn, J. Bos, and K. Striegnitz. *Learn Prolog Now!* London: College Publications, 2006. `http://www.learnPrologNow.org`.

I. Bratko. *Prolog Programming for Artificial Intelligence.* Wokingham: Addison-Wesley Publishing Company, 1990.

J. Brooke. SUS: a 'quick and dirty' usability scale. In P. W. Jordan, B. Thomas, B. A. Weerdmeester, and A. L. McClelland, editors, *Usability Evaluation in Industry*, pages 189–194. London: Taylor and Francis, 1996.

M. Fitting. Negation as refutation. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 63–70, 1989. doi: 10.1109/LICS.1989.39159.

M. Fitting. Bilattices in logic programming. In *Proceedings of the International Symposium on Multiple-Valued Logic*, pages 238–246, 1990. doi: 10.1109/ISMVL.1990.122627.

M. Fitting. Kleene's three valued logics and their children. *Fundamenta Informaticae*, 20:113–131, 1994.

M. Fitting. Lecture notes: First degree entailment. City University of New York, 2018. `http://melvinfitting.org/forclasses/phil76500spring2018/LectureNotes/FDE/FDEA.pdf`.

M. L. Ginsberg. Multi-valued logics. In *Fifth National Conference on Articial Intelligence*, pages 243–247, 1986.

M. L. Ginsberg. Multivalued logics: A uniform approach to inference in artificial intelligence. *Computational Intelligence*, 4:265–316, 1988.

W. H. Hanson. First-degree entailments and information. *Journal of Formal Logic*, 21:659–671, 1980.

B. Kooi and R. Verbrugge. Lecture notes: advanced logic. University of Groningen, 2018.

V. Kuznetsov. A Kripke-Kleene logic over general logic programs. In *Annual Conference on Artificial Intelligence*, pages 70–81. Springer, 1994. doi: 10.1007/3-540-58467-6_7.

T. van Loo. A tableau prover for GL provability logic. Bachelor's thesis, University of Groningen, Netherlands, 2017.

Y. Loyer, N. Spyratos, and D. Stamate. Hypothesis-based semantics of logic programs in multivalued logics. *ACM Transactions on Computational Logic*, 5:508–527, 2004.

G. Malinowski. Kleene logic and inference. *Bulletin of the Section of Logic*, 43:43–52, 2014.

R. Manthey and F. Bry. Satchmo: A theorem prover implemented in Prolog. In Lusk E. and Overbeek R., editors, *9th International Conference on Automated Deduction.*, pages 415–434. Berlin: Heidelberg, 1998.

L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

L. C. Paulson. Generic automatic proof tools. *Automated Reasoning and its Applications*, pages 23–47, 1996.

L. C. Paulson. A generic tableau prover and its integration with Isabelle. *Journal of Universal Computer Science*, 5:73–87, 1999.

Y. Petrukhin. Natural deduction for three-valued regular logics. *Logic and Logical Philosophy*, 26: 197–206, 2017.

G. Priest. *An Introduction to Non-Classical Logic.* Cambridge: Cambridge University Press, 2008.

G. Restall. First degree entailment, symmetry and paradox. *Logic and Logical Philosophy*, 26:3–18, 2017. doi: 10.12775/LLP.2016.028.

Y. Shramko, D. Zaitsev, and A. Belikov. First-degree entailment and its relatives. *Studia Logica*, 105:1291–1317, 2017.

M. E. Stickel. A Prolog technology theorem prover. *New Generation Computing*, 2:371–383, 1984.

M. E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4:353–380, 1988.

A Appendix

This prover works as follows: you type ‘prove([’ followed by the premises that you assume, then you type ‘], ‘|’, ’ followed by either fde, k3 or lp, representing the logics that this system can handle, followed by ‘, [’ and the conclusion that you want to test, followed by ‘]).’, like so:

prove([premises], ‘|’, fde/k3/lp, [conclusion]).

Here are some tableaux rules of those logics, to refresh your memory a bit.

see Figure 2.5.

Now I would like you to make this tableau complete, and you may use the tableaux rules from above:

$$\neg A \wedge \neg E \vdash_{FDE} \neg((A \wedge B) \vee (D \wedge E)$$
$$\neg A \wedge \neg E, +$$
$$\neg((A \wedge B) \vee (D \wedge E), -$$

Now the experimenter fills in D or B for the first question.

Now you may copy, one by one, the 3 first prove sentences into the prover system on your left, which represents the first system, system D. You may hit enter, and then ‘;’ until you see ‘false’. After that you do a survey for system D.
The next 3 prove sentences are for the prover system on your right, again followed by a corresponding survey. This whole thing is them once more repeated with the next prover sentences, after which you can get your homemade treat.

Figure A.1: Instructions given to the participants to do the experiment. The experiment consisted of two times: using the D system, followed by doing the D survey, using the B system, and doing the B survey.

```
system D first time

prove([not,not,a], '|',lp, ['{',b,'V',c,'}','V',a]).
(closed)
prove(['{',not,not,b,'&',not,e,'}',e], '|',k3, [not,not,c,'V',d,'V',not,f]).
(closed)
prove([not,'{',b,'V',not,d,'}'], '|',fde, [c,'V','{',b,'&',d,'}']).
(open)

system B first time

prove([not,not,x], '|',lp, ['{',z,'V',y,'}','V',x]).
(closed)
prove(['{',not,not,x,'&',not,y,'}'], '|',k3, [not,not,x,'V',z,'V',not,y]).
(closed)
prove([not,'{',p,'V',not,q,'}'], '|',fde, [r,'V','{',p,'&',q,'}']).
(open closed)

system D second time

prove([c,'&',not,f], '|',lp, [f]).
(open)
prove([not,not,'{',g,'&',h,'}'], '|',k3, [not,f,'&',not,not,g,'&',i]).
(open closed open)
prove([not,c,'&','{',e,'&',not,f,'}'], '|',fde, [not,not,'{',e,'&',not,f,'}']).
(closed closed)

system B second time

prove([not, s,'&',t], '|',lp, [not,t]).
(open)
prove([not,not,'{',x,'&',r,'}'], '|',k3, [not,r,'&',not,not,q,'&',p]).
(open open open)
prove([not,t,'&','{',not,w,'&',q,'}'], '|',fde, [not,not,'{',not,w,'&',q,'}']).
(closed closed)
```

Figure A.2: Prolog queries given to participants to test and use the two systems.

1. I think that I would like to use this system frequently in the future, when I have to make Logic proofs.
2. I thought the system was hard to use.
3. I found the various functions in this system were well integrated.
4. The system does not have a clean and simple presentation / navigation.
5. I found this system not too complex.
6. I thought there was not sufficient consistency in this system.
7. I would imagine that most students of Advanced Logic would learn to use this system very quickly.
8. The information in the system (the logic rules) is not credible / trustworthy.
9. I would recommend this system to a friend or a colleague who is interested in logic.
10. Overall, I didn't find the interface of the system to be user-friendly.
Comments:
What do you consider to be a (un)useful aspect of this D system? why?

Figure A.3: Questions that were asked in the survey. The scale of the answers goes from strongly disagree to strongly agree. Only the last 2 questions are open questions.

- I am missing explanation on the syntax.
- The interface is not really user-friendly, because the branch is not shown very clearly and the command is not very easily entered.
- I have found it slightly confusing that when there is a branching, the system first displays one part of the branch, then outputs information and then continues with the other part.
- You could insert at the beginning of the system a definition of how your sentence works with the premise, and conclusion that you have inserted, the type of logic, and an overall introduction on how to expect and interpret the functioning of the system. For someone with experience, it is very easy to follow, but someone who is new to logic might not be able to fully understand it at first glance.
- The heading for premises solving threw me off a bit, but it became clear later on.
- The interface could be more clear. It would be useful if it is directly clear what is going on and what the output means.
- When there is a branch splitting, it is not directly clear what happens (but after you have seen one, it is). Furthermore, in the 'prove-function', there are many commas and it is not entirely clear what they all mean.
- I find the system useful: It saves a lot of work for determining the solution of complex formula's and it makes no mistakes.
- The counter-model looks good.
- The system is useful because it gives a clear presentation of the tableaux and how it reaches a conclusion, so that people won't have to manually test everything, but also for students to check that their solutions are correct and teaching them how to solve them on their own.
- I find it useful that you don't have to draw your whole tree yourself.

Figure A.4: Comments of participants on the first survey on the depth-first search version of the system.

- The way that the branching was depicted kind of confused me, and it maybe would have been clearer for me if the premises and conclusion were not solved 'independently' like this.
- The branching looks a bit cleaner than in the B system.
- There are small partly solving the premises or conclusion before going to the premises solving or conclusion solving. It is also inconsistent between the premises and the conclusion.
- Getting the branches in depth-first is annoying! I want to see the whole tableau!

Figure A.5: Comments of participants on the second survey of the depth-first search version of the system. Note that the participant who made the last comment, already guessed that the D system was implemented as a depth-first search strategy.

- Again, I am missing an explanation on the syntax, and I am still unsure how to interpreted 'notzz', as brackets seem to be missing.
- I did not see much difference with the previous system.
- I found the branching in the B system a bit more clear than the D system.
- My comments are the same as for D system.
- I do not see much difference between system B and D at first glance.
- I found that the different solutions, relating to the different branches, were clear.
- I did like this system better than the D system, since the approach to branches that have to split is easier to read and follow.
- I thought the branching was depicted more clearly in this system.
- I found it useful hat you don't have to draw your whole tree yourself, and that the branches are better visualized than in the D system.

Figure A.6: Comments of participants on the first survey on the breadth-first search version of the system.

- I think the system would be really clear if there indeed was a manual. Apart from that, I am unsure about the consistency of bracketing etc.
- I think this system gives a more clear overview by directly showing the different branches. It looks like when you make trees on paper. If you are only interested in one solution, or if the statements become very lengthy, I think it can become less clear.
- I think that this system is more clear. When there is a formula that should be followed by a split branch, it should be re-written again, instead of starting with the branching symbol right-away
- The tableaux presentation: it's nice to see the whole tableaux! I also like that you added the closure rules.
- I liked the branching with the 'OR', but the /\ and // kind of confused me.
- Maybe it is harder to use the system when you have to type in the queries yourself, instead of copy-pasting them. The branches would be a bit more easy to read if the ORs were aligned. Lastly, maybe the number of the branch could be shown before the positive and negative literals so that you know to which branch they belong.

Figure A.7: Comments of participants on the second survey on the breadth-first search version of the system.